

MOSEK Optimizer API for Julia

Release 11.2.7

MOSEK ApS

23 October 2026

Contents

1 Introduction **1**
1.1 Why the Optimizer API for Julia? . . . 2

2 Contact Information **3**

3 License Agreement **4**
3.1 **MOSEK** end-user license agreement . . . 4
3.2 Third party licenses . . . 4

4 Installation **10**
4.1 Running Examples and Testing the Installation . . . 11

5 Design Overview **12**
5.1 Modeling . . . 12
5.2 "Hello World!" in **MOSEK** . . . 12

6 Optimization Tutorials **14**
6.1 Linear Optimization . . . 15
6.2 From Linear to Conic Optimization . . . 20
6.3 Conic Quadratic Optimization . . . 28
6.4 Power Cone Optimization . . . 32
6.5 Conic Exponential Optimization . . . 35
6.6 Geometric Programming . . . 38
6.7 Semidefinite Optimization . . . 42
6.8 Integer Optimization . . . 52
6.9 Disjunctive constraints . . . 56
6.10 Quadratic Optimization . . . 62
6.11 Problem Modification and Reoptimization . . . 68
6.12 Parallel optimization . . . 73
6.13 Retrieving infeasibility certificates . . . 74

7 Solver Interaction Tutorials **78**
7.1 Environment and task . . . 78
7.2 Accessing the solution . . . 79
7.3 Errors and exceptions . . . 82
7.4 Input/Output . . . 85
7.5 Setting solver parameters . . . 86
7.6 Retrieving information items . . . 87
7.7 Progress and data callback . . . 88
7.8 **MOSEK** OptServer . . . 90

8 Debugging Tutorials **94**
8.1 Understanding optimizer log . . . 95
8.2 Addressing numerical issues . . . 99
8.3 Debugging infeasibility . . . 101
8.4 Python Console . . . 106

9 Advanced Numerical Tutorials **109**
9.1 Solving Linear Systems Involving the Basis Matrix . . . 109
9.2 Computing a Sparse Cholesky Factorization . . . 115

10 Technical guidelines **119**
10.1 Memory management and garbage collection . . . 119
10.2 Names . . . 119
10.3 Multithreading . . . 119
10.4 Timing . . . 120
10.5 Efficiency . . . 120
10.6 The license system . . . 121
10.7 Deployment . . . 122

11 Case Studies **123**
11.1 Portfolio Optimization . . . 123
11.2 Logistic regression . . . 147
11.3 Concurrent optimizer . . . 151

12 Problem Formulation and Solutions **155**
12.1 Linear Optimization . . . 155
12.2 Conic Optimization . . . 158
12.3 Semidefinite Optimization . . . 161
12.4 Quadratic and Quadratically Constrained Optimization . . . 163

13 Optimizers **166**
13.1 Presolve . . . 166
13.2 Linear Optimization . . . 168
13.3 Conic Optimization - Interior-point optimizer . . . 175
13.4 The Optimizer for Mixed-Integer Problems . . . 179

14 Additional features **192**
14.1 Problem Analyzer . . . 192
14.2 Automatic Repair of Infeasible Problems . . . 193
14.3 Sensitivity Analysis . . . 197

15 API Reference **204**
15.1 API Conventions . . . 204
15.2 Functions grouped by topic . . . 209
15.3 Functions in alphabetical order . . . 219
15.4 Parameters grouped by topic . . . 394
15.5 Parameters (alphabetical list sorted by type) . . . 406
15.6 Response codes . . . 468
15.7 Enumerations . . . 492
15.8 Supported domains . . . 523
15.9 Environment variables . . . 525

16 Supported File Formats **526**
16.1 The LP File Format . . . 527
16.2 The MPS File Format . . . 531
16.3 The OPF Format . . . 543
16.4 The CBF Format . . . 554
16.5 The PTF Format . . . 571
16.6 The Task Format . . . 579
16.7 The JSON Format . . . 579
16.8 The Solution File Format . . . 585

17 List of examples **589**

18 Interface changes **591**
18.1 Important changes compared to version 10 . . . 591

18.2 Changes compared to version 10 591

Bibliography **596**

Symbol Index **597**

Index **615**

Chapter 1

Introduction

The **MOSEK** Optimization Suite 11.2.7 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 Optimizer API for Julia?

The Optimizer API for Julia provides low-level access to all functionalities of **MOSEK** based on a thin interface to the native C optimizer API. The overhead introduced by this mapping is minimal.

The Optimizer API for Julia 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) including Disjunctive Constraints (DJC)

as well as additional interfaces for:

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

Chapter 2

Contact Information

Phone	+45 7174 9373	Office
	+45 7174 5700	Sales
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
Linkedin	https://www.linkedin.com/company/mosek-aps
Youtube	https://www.youtube.com/channel/UCvIyectEVLP31NXeD5mIbEw

Chapter 3

License Agreement

3.1 MOSEK end-user license agreement

Before using the **MOSEK** software, please read the license agreement available in the distribution at `<MSKHOME>/mosek/11.2/mosek-eula.pdf` or on the **MOSEK** website https://mosek.com/products/license-agreement. By using **MOSEK** you agree to the terms of that license agreement.

3.2 Third party licenses

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

zlib

MOSEK uses 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 1.3.2, February 17th, 2026

Copyright (C) 1995-2026 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 uses the floating point formatting library developed by David M. Gay obtained from the netlib website. The license agreement for *fplib* is shown in Listing 3.2.

Listing 3.2: *fplib* license.

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

{fmt}

MOSEK uses the formatting library *{fmt}* developed by Victor Zverovich obtained form github/fmt and distributed under the MIT license. The license agreement fot *{fmt}* is shown in Listing 3.3.

Listing 3.3: *{fmt}* license.

```
Copyright (c) 2012 - present, Victor Zverovich

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

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

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

Zstandard

MOSEK uses the *Zstandard* library developed by Facebook obtained from github/zstd. The license agreement for *Zstandard* is shown in Listing 3.4.

Listing 3.4: *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.
```

OpenSSL

MOSEK uses the LibReSSL library, which is build on *OpenSSL*. OpenSSL is included under the OpenSSL license, Listing 3.5, and the LibReSSL additions are licensed under the ISC license, Listing 3.6.

Listing 3.5: *OpenSSL* license

```
====================================================================
Copyright (c) 1998-2011 The OpenSSL Project.  All rights reserved.

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

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

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
```

(continues on next page)

(continued from previous page)

```
   the documentation and/or other materials provided with the
   distribution.

3. All advertising materials mentioning features or use of this
   software must display the following acknowledgment:
   "This product includes software developed by the OpenSSL Project
   for use in the OpenSSL Toolkit. (http://www.openssl.org/)"

4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   endorse or promote products derived from this software without
   prior written permission. For written permission, please contact
   openssl-core@openssl.org.

5. Products derived from this software may not be called "OpenSSL"
   nor may "OpenSSL" appear in their names without prior written
   permission of the OpenSSL Project.

6. Redistributions of any form whatsoever must retain the following
   acknowledgment:
   "This product includes software developed by the OpenSSL Project
   for use in the OpenSSL Toolkit (http://www.openssl.org/)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
EXPRESSED 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 OpenSSL PROJECT OR
ITS 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.
====================================================================

This product includes cryptographic software written by Eric Young
(eay@cryptsoft.com).  This product includes software written by Tim
Hudson (tjh@cryptsoft.com).
```

Listing 3.6: *ISC* license

```
Copyright (C) 1994-2017 Free Software Foundation, Inc.
Copyright (c) 2014 Jeremie Courreges-Anglas <jca@openbsd.org>
Copyright (c) 2014-2015 Joel Sing <jsing@openbsd.org>
Copyright (c) 2014 Ted Unangst <tedu@openbsd.org>
Copyright (c) 2015-2016 Bob Beck <beck@openbsd.org>
Copyright (c) 2015 Marko Kreen <markokr@gmail.com>
Copyright (c) 2015 Reyk Floeter <reyk@openbsd.org>
Copyright (c) 2016 Tobias Pape <tobias@netshed.de>

Permission to use, copy, modify, and/or distribute this software for
any purpose with or without fee is hereby granted, provided that the
above copyright notice and this permission notice appear in all
copies.
```

(continues on next page)

(continued from previous page)

```
THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL
WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE
AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL
DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR
PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER
TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

mimalloc

MOSEK uses the *mimalloc* memory allocator library from github/mimalloc. The license agreement for *mimalloc* is shown in Listing 3.7.

Listing 3.7: *mimalloc* license.

```
MIT License

Copyright (c) 2019 Microsoft Corporation, Daan Leijen

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

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

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

BLASFEO

MOSEK uses the *BLASFEO* linear algebra library developed by Gianluca Frison, obtained from github/blasfeo. The license agreement for *BLASFEO* is shown in Listing 3.8.

Listing 3.8: *blasfeo* license.

```
BLASFEO -- BLAS For Embedded Optimization.
Copyright (C) 2019 by Gianluca Frison.
Developed at IMTEK (University of Freiburg) under the supervision of Moritz Diehl.
All rights reserved.

The 2-Clause BSD License

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

1. Redistributions of source code must retain the above copyright notice, this
```

(continues on next page)

(continued from previous page)

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

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

oneTBB

MOSEK uses the *oneTBB* parallelization library which is part of *oneAPI* developed by Intel, obtained from github/oneTBB, licensed under the Apache License 2.0. The license agreement for *oneTBB* can be found in https://github.com/oneapi-src/oneTBB/blob/master/LICENSE.txt .

Chapter 4

Installation

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

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 Optimizer API for Julia is compatible with Julia version 1.6 or later on 64bit platforms (Linux, Windows, macOS) supported by **MOSEK**.

The Optimizer API for Julia is not included in the **MOSEK** distribution, but should be installed using the built-in Julia package installation mechanism.

Locating files in the MOSEK Optimization Suite

The relevant files of the Optimizer API for Julia are organized as reported in Table 4.1.

Table 4.1: Relevant files for the Optimizer API for Julia.

Relative Path	Description	Label
`<MSKHOME>/mosek/11.2/tools/examples/julia`	Examples	`<EXDIR>`
`<MSKHOME>/mosek/11.2/tools/examples/data`	Additional data	`<MISCDIR>`

where

- `<MSKHOME>` is the folder in which the **MOSEK** Optimization Suite has been installed,
- `<PLATFORM>` is the actual platform among those supported by **MOSEK**, i.e. `win64x86`, `linux64x86`, `osxaarch64`, `linuxaarch64`.

Setting up paths

To install the Mosek Julia API from the Julia prompt, do

```
julia> import Pkg
julia> Pkg.add("Mosek")
```

The Mosek.jl package needs the MOSEK distro to be installed. It will do the following to locate the distro:

- If the environment variable `MOSEKJL_FORCE_DOWNLOAD` is set, it will attempt to download and install the MOSEK distro from the MOSEK web page, otherwise

- if the environment variable `MOSEKBINDIR` is set to point to the `bin` directory in the MOSEK distro installation, it will use that, otherwise
- it will look in the user's default home directory for the MOSEK distro installation.

4.1 Running Examples and Testing the Installation

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

```
using Mosek,SparseArrays

env = makeenv()
task = maketask(env=env)
```

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

```
julia lo1.jl
```

Chapter 5

Design Overview

5.1 Modeling

Optimizer API for Julia is an interface for specifying optimization problems directly in matrix form. It means that an optimization problem such as:

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

is specified by describing the matrix A, vectors b, c 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 Optimizer API incurs almost no overhead between the user's representation of the problem and **MOSEK**'s internal one.

Optimizer API for Julia 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 Optimizer API for Julia.

Creating an environment and task

Optionally, an interaction with **MOSEK** using Optimizer API for Julia can begin by creating a **MOSEK environment**. It coordinates the access to **MOSEK** from the current process.

In most cases the user does not interact directly with the environment, except for creating optimization **tasks**, which contain actual problem specifications and where optimization takes place. In this case the user can directly create tasks without invoking an environment, as we do here.

Defining tasks

After a task is created, the input data can be specified. An optimization problem consists of several components; objective, objective sense, constraints, variable bounds etc. See Sec. 6 for basic tutorials on how to specify and solve various types of optimization problems.

Retrieving the solutions

When the model is set up, the optimizer is invoked with the call to *optimize*. When the optimization is over, the user can check the results and retrieve numerical values. 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

```
##
#  Copyright: Copyright (c) MOSEK ApS, Denmark. All rights reserved.
#
#  File:    helloworld.jl
#
#  The most basic example of how to get started with MOSEK.
##


using Mosek

maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    appendvars(task, 1)                                 # 1 variable x
    putcj(task, 1, 1.0)                                 # c_0 = 1.0
    putvarbound(task, 1, MSK_BK_RA, 2.0, 3.0)           # 2.0 <= x <= 3.0
    putobjsense(task, MSK_OBJECTIVE_SENSE_MINIMIZE)     # minimize

    optimize(task)                                      # Optimize

    x = getxx(task, MSK_SOL_ITR)                        # Get solution
    println("Solution x = $(x[1])")                     # Print solution
end
```

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.

- **Model setup and linear optimization tutorial (LO)**
  - Sec. 6.1. Linear optimization tutorial, *recommended first reading for all users.* Apart from setting up a linear problem it also demonstrates how to work with an optimizer task: initialize it, add variables and constraints and retrieve the solution.
- **Conic optimization tutorials (CO)**
  - Sec. 6.2. A step by step introduction to programming with affine conic constraints (ACC). Explains all the steps required to input a conic problem. *Recommended first reading for users of the conic optimizer.*

  Further basic examples demonstrating various types of conic constraints:
  - Sec. 6.3. A basic example with a quadratic cone (CQO).
  - Sec. 6.4. A basic example with a power cone.
  - Sec. 6.5. A basic example with a exponential cone (CEO).
  - Sec. 6.6. A basic tutorial of geometric programming (GP).
- **Semidefinite optimization tutorial (SDO)**
  - Sec. 6.7. Examples showing how to solve semidefinite optimization problems with one or more semidefinite variables.
- **Mixed-integer optimization tutorials (MIO)**
  - Sec. 6.8. Shows how to declare integer variables for linear and conic problems and how to set an initial solution.
  - Sec. 6.9. Demonstrates how to create a problem with disjunctive constraints (DJC).
- **Quadratic optimization tutorial (QO, QCQO)**
  - Sec. 6.10. Examples showing how to solve a quadratic or quadratically constrained problem.
- **Reoptimization tutorials**
  - Sec. 6.11. Various techniques for modifying and reoptimizing a problem.
- **Parallel optimization tutorial**
  - Sec. 6.12. Shows how to optimize tasks in parallel.
- **Infeasibility certificates**
  - Sec. 6.13. Shows how to retrieve and analyze a primal infeasibility certificate for continuous problems.

6.1 Linear Optimization

The simplest optimization problem is a purely linear problem. A *linear optimization problem* (see also Sec. 12.1) is a problem of the following form:

Minimize or maximize the objective function

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

subject to the linear constraints

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

and the bounds

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

The problem description consists of the following elements:

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

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

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

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

- 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:

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

under the bounds

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

Solving the problem

To solve the problem above we go through the following steps:

1. (Optionally) Creating an environment.
2. Creating an optimization task.
3. Loading a problem into the task object.
4. Optimization.
5. Extracting the solution.

Below we explain each of these steps.

Creating an environment.

The user can start by creating a **MOSEK** environment, but it is not necessary if the user does not need access to other functionalities, license management, additional routines, etc. Therefore in this tutorial we don't create an explicit environment.

Creating an optimization task.

We create an empty task object. A task object represents all the data (inputs, outputs, parameters, information items etc.) associated with one optimization problem.

```
maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))
```

We also connect a call-back function to the task log stream. Messages related to the task are passed to the call-back function. In this case the stream call-back function writes its messages to the standard output stream. See Sec. 7.4.

Loading a problem into the task object.

Before any problem data can be set, variables and constraints must be added to the problem via calls to the functions *appendcons* and *appendvars*.

```
    # Append 'numcon' empty constraints.
    # The constraints will initially have no bounds.
    appendcons(task,numcon)
    for i=1:numcon
        putconname(task,i,@sprintf("c%02d",i))
    end

    # Append 'numvar' variables.
    # The variables will initially be fixed at zero (x=0).
    appendvars(task,numvar)
    for j=1:numvar
        putvarname(task,j,@sprintf("x%02d",j))
    end
```

New variables can now be referenced from other functions with indexes in $1,\ldots,$`numvar` and new constraints can be referenced with indexes in $1,\ldots,$`numcon`. More variables and/or constraints can be appended later as needed, these will be assigned indexes from `numvar`/`numcon` and up. Optionally one can add names.

Setting the objective.

Next step is to set the problem data. We first set the objective coefficients $c_j = \texttt{c[j]}$. This can be done with functions such as *putcj* or *putclist*.

```
putclist(task,[1,2,3,4], c)
```

Setting bounds on variables

For every variable we need to specify a bound key and two bounds according to Table 6.1.

Table 6.1: Bound keys as defined in the enum `boundkey`.

Bound key	Type of bound	Lower bound	Upper bound
MSK_BK_FX	$u_j = l_j$	Finite	Identical to the lower bound
MSK_BK_FR	Free	$-\infty$	$+\infty$
MSK_BK_LO	$l_j \leq \cdots$	Finite	$+\infty$
MSK_BK_RA	$l_j \leq \cdots \leq u_j$	Finite	Finite
MSK_BK_UP	$\cdots \leq u_j$	$-\infty$	Finite

For instance `bkx[0]= MSK_BK_LO` means that $x_0 \geq l_0^x$. Finally, the numerical values of the bounds on variables are given by

$$l_j^x = \texttt{blx[j]}$$

and

$$u_j^x = \texttt{bux[j]}.$$

Let us assume we have the bounds on variables stored in the arrays

```
# Bound keys for variables
bkx = [ MSK_BK_LO
        MSK_BK_RA
        MSK_BK_LO
        MSK_BK_LO ]

# Bound values for variables
blx = [   0.0,  0.0,    0.0,    0.0]
bux = [+Inf, 10.0, +Inf, +Inf]
```

Then we can set them using various functions such *putvarbound*, *putvarboundslice*, *putvarboundlist*, depending on what is most convenient in the given context. For instance:

```
putvarboundslice(task, 1, numvar+1, bkx,blx,bux)
```

Defining the linear constraint matrix.

Recall that in our example the A matrix is given by

$$A = \begin{bmatrix} 3 & 1 & 2 & 0 \\ 2 & 1 & 3 & 1 \\ 0 & 2 & 0 & 3 \end{bmatrix}.$$

This matrix is stored in sparse format:

```
# Below is the sparse representation of the A
# matrix stored by column.
A = sparse([1, 2, 1, 2, 3, 1, 2, 2, 3],
           [1, 1, 2, 2, 2, 3, 3, 4, 4],
           [3.0, 2.0, 1.0, 1.0, 2.0, 2.0, 3.0, 1.0, 3.0 ],
           numcon,numvar)
```

The matrix is stored as a standard sparse matrix in Julia, but other representations are also possible.

We now input the linear constraint matrix into the task. This can be done in many alternative ways, row-wise, column-wise or element by element in various orders. See functions such as *putarow*, *putarowlist*, *putaijlist*, *putacol* and similar.

```
putacolslice(task,1,numvar+1,A)
```

Setting bounds on constraints

Finally, the bounds on each constraint are set similarly to the variable bounds, using the bound keys as in Table 6.1. This can be done with one of the many functions *putconbound*, *putconboundslice*, *putconboundlist*, depending on the situation.

```
# Set the bounds on constraints.
# blc[i] <= constraint_i <= buc[i]
putconboundslice(task,1,numcon+1,bkc,blc,buc)
```

Optimization

After the problem is set-up the task can be optimized by calling the function *optimize*.

```
optimize(task)
```

Extracting the solution.

After optimizing the status of the solution is examined with a call to *getsolsta*.

```
solsta = getsolsta(task,MSK_SOL_BAS)
```

If the solution status is reported as *MSK_SOL_STA_OPTIMAL* the solution is extracted:

```
    xx = getxx(task,MSK_SOL_BAS)
```

The *getxx* function obtains the solution. **MOSEK** may compute several solutions depending on the optimizer employed. In this example the *basic solution* is requested by setting the first argument to *MSK_SOL_BAS*. For details about fetching solutions see Sec. 7.2.

Source code

The complete source code `lo1.jl` of this example appears below. See also `lo2.jl` for a version where the A matrix is entered row-wise.

Listing 6.1: Linear optimization example.

```
using Mosek
using Printf, SparseArrays

##############################
## Define problem data

bkc = [MSK_BK_FX
       MSK_BK_LO
       MSK_BK_UP]

# Bound values for constraints
blc = [30.0, 15.0, -Inf]
buc = [30.0, +Inf, 25.0]

# Bound keys for variables
```

(continues on next page)

(continued from previous page)

```
bkx = [ MSK_BK_LO
        MSK_BK_RA
        MSK_BK_LO
        MSK_BK_LO ]

# Bound values for variables
blx = [   0.0,  0.0,    0.0,    0.0]
bux = [+Inf, 10.0, +Inf, +Inf]

numvar = length(bkx)
numcon = length(bkc)

# Objective coefficients
c = [ 3.0, 1.0, 5.0, 1.0 ]

# Below is the sparse representation of the A
# matrix stored by column.
A = sparse([1, 2, 1, 2, 3, 1, 2, 2, 3],
           [1, 1, 2, 2, 2, 3, 3, 4, 4],
           [3.0, 2.0, 1.0, 1.0, 2.0, 2.0, 3.0, 1.0, 3.0 ],
           numcon,numvar)

############################

maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

    putobjname(task,"lo1")

    # Append 'numcon' empty constraints.
    # The constraints will initially have no bounds.
    appendcons(task,numcon)
    for i=1:numcon
        putconname(task,i,@sprintf("c%02d",i))
    end

    # Append 'numvar' variables.
    # The variables will initially be fixed at zero (x=0).
    appendvars(task,numvar)
    for j=1:numvar
        putvarname(task,j,@sprintf("x%02d",j))
    end

    putclist(task,[1,2,3,4], c)

    putacolslice(task,1,numvar+1,A)

    putvarboundslice(task, 1, numvar+1, bkx,blx,bux)

    # Set the bounds on constraints.
    # blc[i] <= constraint_i <= buc[i]
    putconboundslice(task,1,numcon+1,bkc,blc,buc)

    # Input the objective sense (minimize/maximize)
    putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)
```

(continues on next page)

(continued from previous page)

```
    # Solve the problem
    optimize(task)

    # Print a summary containing information
    # about the solution for debugging purposes
    solutionsummary(task,MSK_STREAM_MSG)

    # Get status information about the solution
    solsta = getsolsta(task,MSK_SOL_BAS)

    if solsta == MSK_SOL_STA_OPTIMAL
        xx = getxx(task,MSK_SOL_BAS)
        print("Optimal solution:")
        println(xx)

    elseif solsta in [ MSK_SOL_STA_DUAL_INFEAS_CER,
                       MSK_SOL_STA_PRIM_INFEAS_CER ]
        println("Primal or dual infeasibility certificate found.\n")
    elseif solsta == MSK_SOL_STA_UNKNOWN
        println("Unknown solution status")
    else
        @printf("Other solution status (%d)\n",solsta)
    end
end
```

6.2 From Linear to Conic Optimization

In Sec. 6.1 we demonstrated setting up the linear part of an optimization problem, that is the objective, linear bounds, linear equalities and inequalities. In this tutorial we show how to define conic constraints. We recommend going through this general conic tutorial before proceeding to examples with specific cone types.

MOSEK accepts conic constraints in the form

$$Fx + g \in \mathcal{D}$$

where

- $x \in \mathbb{R}^{n\text{‘}}$ is the optimization variable,
- $D \subseteq \mathbb{R}^k$ is a **conic domain** of some dimension k, representing *one of the cone types supported by MOSEK*,
- $F \in \mathbb{R}^{k \times n}$ and $g \in \mathbb{R}^k$ are data which constitute the sequence of k **affine expressions** appearing in the rows of $Fx + g$.

Constraints of this form will be called **affine conic constraints**, or **ACC** for short. Therefore in this section we show how to set up a problem of the form

$$\begin{array}{lccccc} \text{minimize} & & & c^T x + c^f & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \\ & & & Fx + g & \in & \mathcal{D}_1 \times \cdots \times \mathcal{D}_p, \end{array}$$

with some number p of affine conic constraints.

Note that conic constraints are a natural generalization of linear constraints to the general nonlinear case. For example, a typical linear constraint of the form

$$Ax + b \geq 0$$

can be also written as membership in the cone of nonnegative real numbers:

$$Ax + b \in \mathbb{R}^d_{\geq 0},$$

and that naturally generalizes to

$$Fx + g \in \mathcal{D}$$

for more complicated domains $\mathcal{D}$ from Sec. 15.8 of which $\mathcal{D} = \mathbb{R}^d_{\geq 0}$ is a special case.

6.2.1 Running example

In this tutorial we will consider a sample problem of the form

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & \sum_i x_i = 1, \\ & \gamma \geq \|Gx + h\|_2, \end{array} \tag{6.2}$$

where $x \in \mathbb{R}^n$ is the optimization variable and $G \in \mathbb{R}^{k \times n}$, $h \in \mathbb{R}^k$, $c \in \mathbb{R}^n$ and $\gamma \in \mathbb{R}$. We will use the following sample data:

$$n = 3, \quad k = 2, \quad x \in \mathbb{R}^3, \quad c = [2, 3, -1]^T, \quad \gamma = 0.03, \quad G = \begin{bmatrix} 1.5 & 0.1 & 0 \\ 0.3 & 0 & 2.1 \end{bmatrix}, \quad h = \begin{bmatrix} 0 \\ 0.1 \end{bmatrix}.$$

To be explicit, the problem we are going to solve is therefore:

$$\begin{array}{ll} \text{maximize} & 2x_0 + 3x_1 - x_2 \\ \text{subject to} & x_0 + x_1 + x_2 = 1, \\ & 0.03 \geq \sqrt{(1.5x_0 + 0.1x_1)^2 + (0.3x_0 + 2.1x_2 + 0.1)^2}. \end{array} \tag{6.3}$$

Consulting the *definition of a quadratic cone* $\mathcal{Q}$ we see that the conic form of this problem is:

$$\begin{array}{ll} \text{maximize} & 2x_0 + 3x_1 - x_2 \\ \text{subject to} & x_0 + x_1 + x_2 = 1, \\ & (0.03,\ 1.5x_0 + 0.1x_1,\ 0.3x_0 + 2.1x_2 + 0.1) \in \mathcal{Q}^3. \end{array} \tag{6.4}$$

The conic constraint has an affine conic representation $Fx + g \in \mathcal{D}$ as follows:

$$\begin{bmatrix} 0 & 0 & 0 \\ 1.5 & 0.1 & 0 \\ 0.3 & 0 & 2.1 \end{bmatrix} x + \begin{bmatrix} 0.03 \\ 0 \\ 0.1 \end{bmatrix} \in \mathcal{Q}^3. \tag{6.5}$$

Of course by the same logic in the general case the conic form of the problem (6.2) would be

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & \sum_i x_i = 1, \\ & (\gamma, Gx + h) \in \mathcal{Q}^{k+1} \end{array} \tag{6.6}$$

and the ACC representation of the constraint $(\gamma, Gx + h) \in \mathcal{Q}^{k+1}$ would be

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

Now we show how to add the ACC (6.5). This involves three steps:

- storing the affine expressions which appear in the constraint,
- creating a domain, and
- combining the two into an ACC.

6.2.2 Step 1: add affine expressions

To store affine expressions (**AFE** for short) **MOSEK** provides a matrix $\mathbf{F}$ and a vector $\mathbf{g}$ with the understanding that every row of

$$\mathbf{F}x + \mathbf{g}$$

defines one affine expression. The API functions with infix `afe` are used to operate on $\mathbf{F}$ and $\mathbf{g}$, add rows, add columns, set individual elements, set blocks etc. similarly to the methods for operating on the A matrix of linear constraints. The storage matrix $\mathbf{F}$ is a sparse matrix, therefore only nonzero elements have to be explicitly added.

Remark: the storage $\mathbf{F}, \mathbf{g}$ may, but does not have to be, equal to the pair F, g appearing in the expression $Fx + g$. It is possible to store the AFEs in different order than the order they will be used in F, g, as well as store some expressions only once if they appear multiple times in $Fx + g$. In this first turorial, however, we will for simplicity store all expressions in the same order we will later use them, so that $(\mathbf{F}, \mathbf{g}) = (F, g)$.

In our example we create only one conic constraint (6.5) with three (in general $k+1$) affine expressions

$$\begin{aligned} &0.03, \\ &1.5x_0 + 0.1x_1, \\ &0.3x_0 + 2.1x_2 + 0.1. \end{aligned}$$

Given the previous remark, we initialize the AFE storage as:

$$\mathbf{F} = \begin{bmatrix} 0 & 0 & 0 \\ 1.5 & 0.1 & 0 \\ 0.3 & 0 & 2.1 \end{bmatrix}, \quad \mathbf{g} = \begin{bmatrix} 0.03 \\ 0 \\ 0.1 \end{bmatrix}. \tag{6.7}$$

Initially $\mathbf{F}$ and $\mathbf{g}$ are empty (have 0 rows). We construct them as follows. First, we append a number of empty rows:

```
        # Append empty AFE rows for affine expression storage
        appendafes(task,k + 1)
```

We now have $\mathbf{F}$ and $\mathbf{g}$ with 3 rows of zeros and we fill them up to obtain (6.7).

```
        # G matrix in sparse form
        # Other data
        h     = Float64[0, 0.1]
        gamma = 0.03

        # Construct F matrix in sparse form
        Fsubi = Int32[i + 1 for i in Gsubi]    # G will be placed from row number 1 in␣
↪F
        Fsubj = Gsubj
        Fval  = Gval

        # Fill in F storage
        putafefentrylist(task,Fsubi, Fsubj, Fval)

       # Fill in g storage
        putafeg(task,1, gamma)
        putafegslice(task,2, k+2, h)
```

We have now created the matrices from (6.7). Note that at this point we have *not defined any ACC yet*. All we did was define some affine expressions and place them in a generic AFE storage facility to be used later.

6.2.3 Step 2: create a domain

Next, we create the domain to which the ACC belongs. Domains are created with functions with infix `domain`. In the case of (6.5) we need a quadratic cone domain of dimension 3 (in general $k+1$), which we create with:

```
        # Define a conic quadratic domain
        quadDom = appendquadraticconedomain(task,k + 1)
```

The function returns a domain index, which is just the position in the list of all domains (potentially) created for the problem. At this point the domain is just stored in the list of domains, but not yet used for anything.

6.2.4 Step 3: create the actual constraint

We are now in position to create the affine conic constraint. ACCs are created with functions with infix `acc`. The most basic variant, *appendacc* will append an affine conic constraint based on the following data:

- the list `afeidx` of indices of AFEs to be used in the constraint. These are the row numbers in $\mathbf{F}, \mathbf{g}$ which contain the required affine expressions.
- the index `domidx` of the domain to which the constraint belongs.

Note that number of AFEs used in `afeidx` must match the dimension of the domain.

In case of (6.5) we have already arranged $\mathbf{F}, \mathbf{g}$ in such a way that their (only) three rows contain the three affine expressions we need (in the correct order), and we already defined the quadratic cone domain of matching dimension 3. The ACC is now constructed with the following call:

```
        # Create the ACC
        appendaccseq(task,
                     quadDom,     # Domain index
                     1,           # Indices of AFE rows [0,...,k]
                     nothing)     # Ignored
```

This completes the setup of the affine conic constraint.

6.2.5 Example ACC1

We refer to Sec. 6.1 for instructions how to set up the objective and linear constraint $x_0 + x_1 + x_2 = 1$. All else that remains is to set up the **MOSEK** environment, task, add variables, call the solver with *optimize* and retrieve the solution with *getxx*. Since our problem contains a nonlinear constraint we fetch the interior-point solution. The full code solving problem (6.3) is shown below.

Listing 6.2: Full code of example ACC1.

```
using Mosek

# Define problem data
n, k = 3, 2


let Gsubi = Int64[1, 1, 2, 2],
    Gsubj = Int32[1, 2, 1, 3],
    Gval  = Float64[1.5, 0.1, 0.3, 2.1]

    # Make a MOSEK environment
    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        # Attach a printer to the task
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))
```

(continues on next page)

(continued from previous page)

```
        # Create n free variables
        appendvars(task,n)
        putvarboundsliceconst(task,1, n+1, MSK_BK_FR, -Inf, Inf)

        # Set up the objective
        putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)
        putclist(task,[1,2,3],[2.0,3.0,-1.0])

        # One linear constraint - sum(x) = 1
        appendcons(task,1)
        putarow(task,1, [1,2,3], [1.0,1.0,1.0])
        putconbound(task,1, MSK_BK_FX, 1.0, 1.0)

        # Append empty AFE rows for affine expression storage
        appendafes(task,k + 1)

        # G matrix in sparse form
        # Other data
        h     = Float64[0, 0.1]
        gamma = 0.03

        # Construct F matrix in sparse form
        Fsubi = Int32[i + 1 for i in Gsubi]    # G will be placed from row number 1 in␣
↪F
        Fsubj = Gsubj
        Fval  = Gval

        # Fill in F storage
        putafefentrylist(task,Fsubi, Fsubj, Fval)

       # Fill in g storage
        putafeg(task,1, gamma)
        putafegslice(task,2, k+2, h)

        # Define a conic quadratic domain
        quadDom = appendquadraticconedomain(task,k + 1)

        # Create the ACC
        appendaccseq(task,
                     quadDom,     # Domain index
                     1,           # Indices of AFE rows [0,...,k]
                     nothing)     # Ignored

        # Solve and retrieve solution
        optimize(task)
        writedata(task,"acc1.ptf")

        xx = getxx(task,MSK_SOL_ITR)
        println("Solution: $xx")
        # Demonstrate retrieving activity of ACC
        activity = evaluateacc(task,MSK_SOL_ITR,
                               1)      # ACC index
        println("Activity of ACC:: $activity")

        # Demonstrate retrieving the dual of ACC
```

(continues on next page)

(continued from previous page)

```
        doty = getaccdoty(task,MSK_SOL_ITR,
                          1)          # ACC index
        println("Dual of ACC: $doty")

    end
end
```

The answer is

```
[-0.07838011145615721, 1.1289128998004547, -0.0505327883442975]
```

The dual values $\dot{y}$ of an ACC can be obtained with *getaccdoty* if required.

```
        # Demonstrate retrieving the dual of ACC
        doty = getaccdoty(task,MSK_SOL_ITR,
                          1)          # ACC index
        println("Dual of ACC: $doty")
```

6.2.6 Example ACC2 - more conic constraints

Now that we know how to enter one affine conic constraint (ACC) we will demonstrate a problem with two ACCs. From there it should be clear how to add multiple ACCs. To keep things familiar we will reuse the previous problem, but this time cast it into a conic optimization problem with two ACCs as follows:

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & (\sum_i x_i - 1,\ \gamma,\ Gx + h) \in \{0\} \times \mathcal{Q}^{k+1} \end{array} \tag{6.8}$$

or, using the data from the example:

$$\begin{array}{lll} \text{maximize} & 2x_0 + 3x_1 - x_2 & \\ \text{subject to} & x_0 + x_1 + x_2 - 1 & \in \{0\}, \\ & (0.03, 1.5x_0 + 0.1x_1, 0.3x_0 + 2.1x_2 + 0.1) & \in \mathcal{Q}^3 \end{array}$$

In other words, we transformed the linear constraint into an ACC with the one-point zero domain.

As before, we proceed in three steps. First, we add the variables and create the storage $\mathbf{F}$, $\mathbf{g}$ containing all affine expressions that appear throughout all off the ACCs. It means we will require 4 rows:

$$\mathbf{F} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 1.5 & 0.1 & 0 \\ 0.3 & 0 & 2.1 \end{bmatrix}, \quad \mathbf{g} = \begin{bmatrix} -1 \\ 0.03 \\ 0 \\ 0.1 \end{bmatrix}. \tag{6.9}$$

```
    # Set AFE rows representing the linear constraint
    appendafes(task,1)
    putafefrow(task,1, [1:n...], ones(n))
    putafeg(task,1, -1.0)

    # Set AFE rows representing the quadratic constraint
    appendafes(task,k + 1)
    putafefrow(task,3,           # afeidx, row number
                    [1, 2],      # varidx, column numbers
                    [1.5, 0.1]) # values
    putafefrow(task,4,           # afeidx, row number
                    [1, 3],      # varidx, column numbers
                    [0.3, 2.1]) # values

    h     = [0, 0.1]
```

(continues on next page)

(continued from previous page)

```
    gamma = 0.03
    putafeg(task,2, gamma)
    putafegslice(task,3, k+2+1, h)
```

Next, we add the required domains: the zero domain of dimension 1, and the quadratic cone domain of dimension 3.

```
    # Define domains
    zeroDom = appendrzerodomain(task,1)
    quadDom = appendquadraticconedomain(task,k + 1)
```

Finally, we create both ACCs. The first ACCs picks the 0-th row of $\mathbf{F}, \mathbf{g}$ and places it in the zero domain:

```
    appendacc(task,zeroDom,     # Domain index
              [1],              # Indices of AFE rows
              nothing)          # Ignored
```

The second ACC picks rows $1, 2, 3$ in $\mathbf{F}, \mathbf{g}$ and places them in the quadratic cone domain:

```
    appendacc(task,quadDom,     # Domain index
              [2,3,4],          # Indices of AFE rows
              nothing)          # Ignored
```

The completes the construction and we can solve the problem like before:

Listing 6.3: Full code of example ACC2.

```
using Mosek


# Define problem data
n = 3
k = 2

# Create a task
maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    # Attach a printer to the task
    putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

    # Create n free variables
    appendvars(task,n)
    putvarboundsliceconst(task,1, n+1, MSK_BK_FR, -Inf, Inf)

    # Set up the objective
    c = Float64[2, 3, -1]
    putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)
    putclist(task,[1:n...], c)

    # Set AFE rows representing the linear constraint
    appendafes(task,1)
    putafefrow(task,1, [1:n...], ones(n))
    putafeg(task,1, -1.0)

    # Set AFE rows representing the quadratic constraint
    appendafes(task,k + 1)
    putafefrow(task,3,          # afeidx, row number
                    [1, 2],     # varidx, column numbers
```

(continues on next page)

(continued from previous page)

```
                    [1.5, 0.1]) # values
    putafefrow(task,4,          # afeidx, row number
                    [1, 3],     # varidx, column numbers
                    [0.3, 2.1]) # values

    h     = [0, 0.1]
    gamma = 0.03
    putafeg(task,2, gamma)
    putafegslice(task,3, k+2+1, h)

    # Define domains
    zeroDom = appendrzerodomain(task,1)
    quadDom = appendquadraticconedomain(task,k + 1)

    # Append affine conic constraints
    appendacc(task,zeroDom,     # Domain index
              [1],              # Indices of AFE rows
              nothing)          # Ignored
    appendacc(task,quadDom,     # Domain index
              [2,3,4],          # Indices of AFE rows
              nothing)          # Ignored

    # Solve and retrieve solution
    optimize(task)
    writedata(task,"acc2.ptf")
    @assert getsolsta(task,MSK_SOL_ITR) == MSK_SOL_STA_OPTIMAL
    xx = getxx(task,MSK_SOL_ITR)
    if getsolsta(task,MSK_SOL_ITR) == MSK_SOL_STA_OPTIMAL
        println("Solution: $xx")
    end

    # Demonstrate retrieving activity of ACC
    activity = evaluateacc(task,MSK_SOL_ITR,2)
    println("Activity of quadratic ACC:: $activity")

    # Demonstrate retrieving the dual of ACC
    doty = getaccdoty(task,MSK_SOL_ITR,2)
    println("Dual of quadratic ACC:: $doty")

end
```

We obtain the same result:

```
[-0.07838011145615721, 1.1289128998004547, -0.0505327883442975]
```

6.2.7 Summary and extensions

In this section we presented the most basic usage of the affine expression storage $\mathbf{F}, \mathbf{g}$ to input *affine expressions* used together with *domains* to create *affine conic constraints*. Now we briefly point out additional features of his interface which can be useful in some situations for more demanding users. They will be demonstrated in various examples in other tutorials and case studies in this manual.

- It is important to remember that $\mathbf{F}, \mathbf{g}$ has *only a storage function* and during the ACC construction we can pick an arbitrary list of row indices and place them in a conic domain. It means for example that:
  - It is not necessary to store the AFEs in the same order they will appear in ACCs.

  - The same AFE index can appear more than once in one and/or more conic constraints (this can be used to reduce storage if the same affine expression is used in multiple ACCs).
  - The $\mathbf{F}, \mathbf{g}$ storage can even include rows that are not presently used in any ACC.
- Domains can be reused: multiple ACCs can use the same domain. On the other hand the same type of domain can appear under many `domidx` positions. In this sense the list of created domains also plays only a *storage role*: the domains are only used when they enter an ACC.
- Affine expressions can also contain semidefinite terms, ie. the most general form of an ACC is in fact

  $$Fx + \langle \bar{F}, \overline{X} \rangle + g \in \mathcal{D}$$

  These terms are input into the rows of AFE storage using the functions with infix `afebarf`, creating an additional storage structure $\bar{\mathbf{F}}$.
- The same affine expression storage $\mathbf{F}, \mathbf{g}$ is shared between affine conic and disjunctive constraints (see Sec. 6.9).
- If, on the other hand, the user chooses to always store the AFEs one by one sequentially in the same order as they appear in ACCs then sequential functions such as *appendaccseq* and *appendaccsseq* make it easy to input one or more ACCs by just specifying the starting AFE index and dimension.
- It is possible to add a number of ACCs in one go using *appendaccs*.
- When defining an ACC an additional constant vector b can be provided to modify the constant terms coming from $\mathbf{g}$ but only for this particular ACC. This could be useful to reduce $\mathbf{F}$ storage space if, for example, many expressions $f^T x - b_i$ with the same linear part $f^T x$, but varying constant terms b_i, are to be used throughout ACCs.

6.3 Conic Quadratic Optimization

The structure of a typical conic optimization problem is

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

(see Sec. 12 for detailed formulations). We recommend Sec. 6.2 for a tutorial on how problems of that form are represented in MOSEK and what data structures are relevant. Here we discuss how to set-up problems with the **(rotated) quadratic cones**.

MOSEK supports two types of quadratic cones, namely:

- Quadratic cone:

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

- Rotated quadratic cone:

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

  For example, consider the following constraint:

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

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

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

For other types of cones supported by **MOSEK**, see Sec. 15.8 and the other tutorials in this chapter. Different cone types can appear together in one optimization problem.

6.3.1 Example CQO1

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

$$\begin{array}{lccc} \text{minimize} & x_4 + x_5 + x_6 & & \\ \text{subject to} & x_1 + x_2 + 2x_3 & = & 1, \\ & x_1, x_2, x_3 & \geq & 0, \\ & x_4 \geq \sqrt{x_1^2 + x_2^2}, & & \\ & 2x_5 x_6 \geq x_3^2 & & \end{array} \tag{6.10}$$

The two conic constraints can be expressed in the ACC form as shown in (6.11)

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \in \mathcal{Q}^3 \times \mathcal{Q}_r^3. \tag{6.11}$$

Setting up the linear part

The linear parts (constraints, variables, objective) are set up using exactly the same methods as for linear problems, and we refer to Sec. 6.1 for all the details. The same applies to technical aspects such as defining an optimization task, retrieving the solution and so on.

Setting up the conic constraints

In order to append the conic constraints we first input the matrix $\mathbf{F}$ and vector $\mathbf{g}$ appearing in (6.11). The matrix $\mathbf{F}$ is sparse and we input only its nonzeros using *putafefentrylist*. Since $\mathbf{g}$ is zero, nothing needs to be done about this vector.

Each of the conic constraints is appended using the function *appendacc*. In the first case we append the quadratic cone determined by the first three rows of $\mathbf{F}$ and then the rotated quadratic cone depending on the remaining three rows of $\mathbf{F}$.

```
# Input the cones
appendafes(task,6)
putafefentrylist(task,
                 [1, 2, 3, 4, 5, 6],          # Rows
                 [4, 1, 2, 5, 6, 3],          # Columns */
                 [1.0, 1.0, 1.0, 1.0, 1.0, 1.0])

# Quadratic cone (x(3),x(0),x(1)) \in QUAD_3
quadcone  = appendquadraticconedomain(task,3)
appendacc(task,
          quadcone,  # Domain
          [1, 2, 3], # Rows from F
          nothing)

# Rotated quadratic cone (x(4),x(5),x(2)) \in RQUAD_3
rquadcone = appendrquadraticconedomain(task,3)
appendacc(task,
          rquadcone, # Domain
          [4, 5, 6], # Rows from F
          nothing);
```

The first argument selects the domain, which must be appended before being used, and must have the dimension matching the number of affine expressions appearing in the constraint. Variants of this method are available to append multiple ACCs at a time. It is also possible to define the matrix $\mathbf{F}$ using

a variety of methods (row after row, column by column, individual entries, etc.) similarly as for the linear constraint matrix A.

For a more thorough exposition of the affine expression storage (AFE) matrix $\mathbf{F}$ and vector $\mathbf{g}$ see Sec. 6.2.

Source code

Listing 6.4: Source code solving problem (6.10).

```
using Mosek

printstream(msg::AbstractString) = print(msg)
callback(where,dinf,iinf,liinf) = 0

# Since the actual value of Infinity is ignores, we define it solely
# for symbolic purposes:

bkc = [ MSK_BK_FX ]
blc = [ 1.0 ]
buc = [ 1.0 ]

c   = [               0.0,              0.0,              0.0,
                      1.0,              1.0,              1.0 ]
bkx = [ MSK_BK_LO,MSK_BK_LO,MSK_BK_LO,
        MSK_BK_FR,MSK_BK_FR,MSK_BK_FR ]
blx = [               0.0,              0.0,              0.0,
                     -Inf,             -Inf,             -Inf ]
bux = [               Inf,              Inf,              Inf,
                      Inf,              Inf,              Inf ]

asub  = [ 1 ,2, 3 ]
aval  = [ 1.0, 1.0, 2.0 ]

numvar = length(bkx)
numcon = length(bkc)


# Create a task
maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    putstreamfunc(task,MSK_STREAM_LOG,printstream)
    putcallbackfunc(task,callback)

    # Append 'numcon' empty constraints.
    # The constraints will initially have no bounds.
    appendcons(task,numcon)

    #Append 'numvar' variables.
    # The variables will initially be fixed at zero (x=0).
    appendvars(task,numvar)

    # Set the linear term c_j in the objective.
    putclist(task,[1:6;],c)

    # Set the bounds on variable j
    # blx[j] <= x_j <= bux[j]
    putvarboundslice(task,1,numvar+1,bkx,blx,bux)
```

(continues on next page)

(continued from previous page)

```
    putarow(task,1,asub,aval)
    putconbound(task,1,bkc[1],blc[1],buc[1])

    # Input the cones
    appendafes(task,6)
    putafefentrylist(task,
                     [1, 2, 3, 4, 5, 6],          # Rows
                     [4, 1, 2, 5, 6, 3],          # Columns */
                     [1.0, 1.0, 1.0, 1.0, 1.0, 1.0])

    # Quadratic cone (x(3),x(0),x(1)) \in QUAD_3
    quadcone  = appendquadraticconedomain(task,3)
    appendacc(task,
              quadcone,  # Domain
              [1, 2, 3], # Rows from F
              nothing)

    # Rotated quadratic cone (x(4),x(5),x(2)) \in RQUAD_3
    rquadcone = appendrquadraticconedomain(task,3)
    appendacc(task,
              rquadcone, # Domain
              [4, 5, 6], # Rows from F
              nothing);

    # Input the objective sense (minimize/maximize)
    putobjsense(task,MSK_OBJECTIVE_SENSE_MINIMIZE)

    # Optimize the task
    #optimize(task,"mosek://solve.mosek.com:30080")
    optimize(task)
    writedata(task,"cqo1.ptf")
    # Print a summary containing information
    # about the solution for debugging purposes
    solutionsummary(task,MSK_STREAM_MSG)
    prosta = getprosta(task,MSK_SOL_ITR)
    solsta = getsolsta(task,MSK_SOL_ITR)

    if solsta == MSK_SOL_STA_OPTIMAL
        # Output a solution
        xx = getxx(task,MSK_SOL_ITR)
        println("Optimal solution: $xx")
    elseif solsta == MSK_SOL_STA_DUAL_INFEAS_CER
        println("Primal or dual infeasibility.\n")
    elseif solsta == MSK_SOL_STA_PRIM_INFEAS_CER
        println("Primal or dual infeasibility.\n")
    elseif solsta == MSK_SOL_STA_UNKNOWN
        println("Unknown solution status")
    else
        println("Other solution status")
    end

end
```

6.4 Power Cone Optimization

The structure of a typical conic optimization problem is

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

(see Sec. 12 for detailed formulations). Here we discuss how to set-up problems with the **primal/dual power cones**.

MOSEK supports the primal and dual power cones, defined as below:

- Primal power cone:

$$\mathcal{P}_n^{\alpha_k} = \left\{ x \in \mathbb{R}^n \ : \ \prod_{i=0}^{n_\ell - 1} x_i^{\beta_i} \geq \sqrt{\sum_{j=n_\ell}^{n-1} x_j^2}, \ x_0 \ldots, x_{n_\ell - 1} \geq 0 \right\}$$

  where $s = \sum_i \alpha_i$ and $\beta_i = \alpha_i / s$, so that $\sum_i \beta_i = 1$.

- Dual power cone:

$$(\mathcal{P}_n^{\alpha_k}) = \left\{ x \in \mathbb{R}^n \ : \ \prod_{i=0}^{n_\ell - 1} \left(\frac{x_i}{\beta_i} \right)^{\beta_i} \geq \sqrt{\sum_{j=n_\ell}^{n-1} x_j^2}, \ x_0 \ldots, x_{n_\ell - 1} \geq 0 \right\}$$

  where $s = \sum_i \alpha_i$ and $\beta_i = \alpha_i / s$, so that $\sum_i \beta_i = 1$.

Perhaps the most important special case is the three-dimensional power cone family:

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

which has the corresponding dual cone:

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 $x, y \geq 0$.

For other types of cones supported by **MOSEK**, see Sec. 15.8 and the other tutorials in this chapter. Different cone types can appear together in one optimization problem.

6.4.1 Example POW1

Consider the following optimization problem which involves powers of variables:

$$\begin{array}{lccc} \text{maximize} & x_0^{0.2} x_1^{0.8} + x_2^{0.4} - x_0 & & \\ \text{subject to} & x_0 + x_1 + \frac{1}{2} x_2 & = & 2, \\ & x_0, x_1, x_2 & \geq & 0. \end{array} \tag{6.12}$$

We convert (6.12) into affine conic form using auxiliary variables as bounds for the power expressions:

$$\begin{array}{lccc} \text{maximize} & x_3 + x_4 - x_0 & & \\ \text{subject to} & x_0 + x_1 + \frac{1}{2} x_2 & = & 2, \\ & (x_0, x_1, x_3) & \in & \mathcal{P}_3^{0.2, 0.8}, \\ & (x_2, 1.0, x_4) & \in & \mathcal{P}_3^{0.4, 0.6}. \end{array} \tag{6.13}$$

The two conic constraints shown in (6.13) can be expressed in the ACC form as shown in (6.14):

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \in \mathcal{P}_3^{0.2, 0.8} \times \mathcal{P}_3^{0.4, 0.6}. \tag{6.14}$$

Setting up the linear part

The linear parts (constraints, variables, objective) are set up using exactly the same methods as for linear problems, and we refer to Sec. 6.1 for all the details. The same applies to technical aspects such as defining an optimization task, retrieving the solution and so on.

Setting up the conic constraints

In order to append the conic constraints we first input the matrix $\mathbf{F}$ and vector $\mathbf{g}$ which together determine all the six affine expressions appearing in the conic constraints of (6.13)

```
    # Input the cones
    pc1 = appendprimalpowerconedomain(task,3,[0.2, 0.8])
    pc2 = appendprimalpowerconedomain(task,3,[4.0, 6.0])

    appendafes(task,6)
    putafefentrylist(task,
                     [1, 2, 3, 4, 6], # Rows
                     [1, 2, 4, 3, 5],  #Columns
                     [1.0, 1.0, 1.0, 1.0, 1.0])
    putafeg(task,5,1.0)

    # Append the two conic constraints
    appendacc(task,
              pc1,           # Domain
              [1, 2, 3],     # Rows from F
              nothing)
    appendacc(task,
              pc2,           # Domain
              [4, 5, 6],     # Rows from F
              nothing)
```

Following that, each of the affine conic constraints is appended using the function *appendacc*. The first argument selects the domain, which must be appended before being used, and must have the dimension matching the number of affine expressions appearing in the constraint. In the first case we append the power cone determined by the first three rows of $\mathbf{F}$ and $\mathbf{g}$ while in the second call we use the remaining three rows of $\mathbf{F}$ and $\mathbf{g}$.

Variants of this method are available to append multiple ACCs at a time. It is also possible to define the matrix $\mathbf{F}$ using a variety of methods (row after row, column by column, individual entries, etc.) similarly as for the linear constraint matrix A.

For a more thorough exposition of the affine expression storage (AFE) matrix $\mathbf{F}$ and vector $\mathbf{g}$ see Sec. 6.2.

Source code

Listing 6.5: Source code solving problem (6.12).

```
using Mosek
printstream(msg::AbstractString) = print(msg)


csub = [ 4, 5, 1 ]
cval = [ 1.0, 1.0, -1.0]
asub = [ 1, 2, 3]
aval = [ 1.0, 1.0, 0.5]
numvar = 5
numcon = 1
```

(continues on next page)

(continued from previous page)

```
# Create a task
maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

    appendcons(task,numcon)
    appendvars(task,numvar)

    # Set up the linear part of the problem
    putclist(task,csub, cval)
    putarow(task,1, asub, aval)
    putconbound(task,1, MSK_BK_FX, 2.0, 2.0)

    putvarboundsliceconst(task,1, numvar+1,MSK_BK_FR,-Inf,Inf)

    # Input the cones
    pc1 = appendprimalpowerconedomain(task,3,[0.2, 0.8])
    pc2 = appendprimalpowerconedomain(task,3,[4.0, 6.0])

    appendafes(task,6)
    putafefentrylist(task,
                     [1, 2, 3, 4, 6], # Rows
                     [1, 2, 4, 3, 5],  #Columns
                     [1.0, 1.0, 1.0, 1.0, 1.0])
    putafeg(task,5,1.0)

    # Append the two conic constraints
    appendacc(task,
              pc1,           # Domain
              [1, 2, 3],     # Rows from F
              nothing)
    appendacc(task,
              pc2,           # Domain
              [4, 5, 6],     # Rows from F
              nothing)

    # Input the objective sense (minimize/maximize)
    putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)

    # Optimize the task
    optimize(task)
    writedata(task,"pow1.ptf")
    # Print a summary containing information
    # about the solution for debugging purposes
    solutionsummary(task,MSK_STREAM_MSG)
    prosta = getprosta(task,MSK_SOL_ITR)
    solsta = getsolsta(task,MSK_SOL_ITR)

    if solsta == MSK_SOL_STA_OPTIMAL
        # Output a solution
        xx = getxx(task,MSK_SOL_ITR)
        println("Optimal solution: $(xx[1:3])")
    elseif solsta == MSK_SOL_STA_DUAL_INFEAS_CER
        println("Primal or dual infeasibility.")
    elseif solsta == MSK_SOL_STA_PRIM_INFEAS_CER
        println("Primal or dual infeasibility.")
```

(continues on next page)

(continued from previous page)

```
    elseif solsta == MSK_SOL_STA_UNKNOWN
        println("Unknown solution status")
    else
        println("Other solution status")
    end
end
```

6.5 Conic Exponential Optimization

The structure of a typical conic optimization problem is

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

(see Sec. 12 for detailed formulations). We recommend Sec. 6.2 for a tutorial on how problems of that form are represented in MOSEK and what data structures are relevant. Here we discuss how to set-up problems with the **primal/dual exponential cones**.

MOSEK supports two exponential cones, namely:

- Primal exponential cone:

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

- Dual exponential cone:

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

For example, consider the following constraint:

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

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

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

For other types of cones supported by **MOSEK**, see Sec. 15.8 and the other tutorials in this chapter. Different cone types can appear together in one optimization problem.

6.5.1 Example CEO1

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

$$\begin{array}{lrcl} \text{minimize} & x_0 + x_1 & & \\ \text{subject to} & x_0 + x_1 + x_2 & = & 1, \\ & x_0 & \geq & x_1 \exp(x_2/x_1), \\ & x_0, x_1 & \geq & 0. \end{array} \tag{6.15}$$

The affine conic form of (6.15) is:

$$\begin{array}{lrcl} \text{minimize} & x_0 + x_1 & & \\ \text{subject to} & x_0 + x_1 + x_2 & = & 1, \\ & Ix & \in & K_{\text{exp}}, \\ & x & \in & \mathbb{R}^3. \end{array} \tag{6.16}$$

where I is the 3×3 identity matrix.

Setting up the linear part

The linear parts (constraints, variables, objective) are set up using exactly the same methods as for linear problems, and we refer to Sec. 6.1 for all the details. The same applies to technical aspects such as defining an optimization task, retrieving the solution and so on.

Setting up the conic constraints

In order to append the conic constraints we first input the sparse identity matrix $\mathbf{F}$ as indicated by (6.16).

The affine conic constraint is then appended using the function *appendacc*, with the primal exponential domain and the list of $\mathbf{F}$ rows, in this case consisting of all rows in their natural order.

```
    # Create a 3x3 identity matrix F
    appendafes(task,3)
    putafefentrylist(task,
                     [1, 2, 3],          # Rows
                     [1, 2, 3],          # Columns
                     ones(3))

    # Exponential cone (x(0),x(1),x(2)) \in EXP
    expdomain = appendprimalexpconedomain(task)
    appendacc(task,
              expdomain,                 # Domain
              [1, 2, 3],                 # Rows from F
              nothing)                   # Unused
```

The first argument selects the domain, which must be appended before being used, and must have the dimension matching the number of affine expressions appearing in the constraint. Variants of this method are available to append multiple ACCs at a time. It is also possible to define the matrix $\mathbf{F}$ using a variety of methods (row after row, column by column, individual entries, etc.) similarly as for the linear constraint matrix A.

For a more thorough exposition of the affine expression storage (AFE) matrix $\mathbf{F}$ and vector $\mathbf{g}$ see Sec. 6.2.

Source code

Listing 6.6: Source code solving problem (6.15).

```
using Mosek
using Printf, SparseArrays



# Create a task
maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    # Attach a printer to the task
    putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

    c = [1.0, 1.0, 0.0]
    a = [1.0, 1.0, 1.0]
    numvar = 3
    numcon = 1

    # Append 'numcon' empty constraints.
    # The constraints will initially have no bounds.
    appendcons(task,numcon)
```

(continues on next page)

(continued from previous page)

```
    # Append 'numvar' variables.
    # The variables will initially be fixed at zero (x=0).
    appendvars(task,numvar)

    # Set up the linear part of the problem
    putcslice(task,1, numvar+1, c)
    putarow(task,1, [1, 2, 3], a)
    putvarboundsliceconst(task,1, numvar+1, MSK_BK_FR, -Inf, Inf)
    putconbound(task,1, MSK_BK_FX, 1.0, 1.0)
    # Add a conic constraint
    # Create a 3x3 identity matrix F
    appendafes(task,3)
    putafefentrylist(task,
                     [1, 2, 3],          # Rows
                     [1, 2, 3],          # Columns
                     ones(3))

    # Exponential cone (x(0),x(1),x(2)) \in EXP
    expdomain = appendprimalexpconedomain(task)
    appendacc(task,
              expdomain,                 # Domain
              [1, 2, 3],                 # Rows from F
              nothing)                   # Unused

    # Input the objective sense (minimize/maximize)
    putobjsense(task,MSK_OBJECTIVE_SENSE_MINIMIZE)

    # Optimize the task
    optimize(task)
    # Print a summary containing information
    # about the solution for debugging purposes
    solutionsummary(task,MSK_STREAM_MSG)
    prosta = getprosta(task,MSK_SOL_ITR)
    solsta = getsolsta(task,MSK_SOL_ITR)

    # Output a solution
    xx = getxx(task,MSK_SOL_ITR)

    if solsta == MSK_SOL_STA_OPTIMAL
        println("Optimal solution: $xx")
    elseif solsta == MSK_SOL_STA_DUAL_INFEAS_CER
        println("Primal or dual infeasibility.")
    elseif solsta == MSK_SOL_STA_PRIM_INFEAS_CER
        println("Primal or dual infeasibility.")
    elseif solsta == MSK_SOL_STA_UNKNOWN
        println("Unknown solution status")
    else
        println("Other solution status")
    end


end
```

6.6 Geometric Programming

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

$$\begin{array}{lll} \text{minimize} & f_0(x) & \\ \text{subject to} & f_i(x) \leq 1, & i = 1, \ldots, m, \\ & x_j > 0, & j = 1, \ldots, n, \end{array} \tag{6.17}$$

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

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

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

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

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

$$\log(\sum_k \exp(a_k^T y + b_k)) \leq 0 \tag{6.18}$$

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.

6.6.1 Example GP1

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

$$\begin{array}{lrl} \text{maximize} & hwd & \\ \text{subject to} & 2(hw + hd) & \leq A_{\text{wall}}, \\ & wd & \leq A_{\text{floor}}, \\ & \alpha & \leq h/w \leq \beta, \\ & \gamma & \leq d/w \leq \delta. \end{array} \tag{6.19}$$

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 (6.19) becomes

$$\begin{array}{ll} \text{maximize} & x + y + z \\ \text{subject to} & \log(\exp(x + y + \log(2/A_{\text{wall}})) + \exp(x + z + \log(2/A_{\text{wall}}))) \leq 0, \\ & y + z \leq \log(A_{\text{floor}}), \\ & \log(\alpha) \leq x - y \leq \log(\beta), \\ & \log(\gamma) \leq z - y \leq \log(\delta). \end{array} \tag{6.20}$$

Next, we demonstrate how to implement a log-sum-exp constraint (6.18). It can be written as:

$$\begin{array}{l} u_k \geq \exp(a_k^T y + b_k), \quad (\text{equiv. } (u_k, 1, a_k^T y + b_k) \in K_{\exp}), \\ \sum_k u_k = 1. \end{array} \tag{6.21}$$

This presentation requires one extra variable u_k for each monomial appearing in the original posynomial constraint. In this case the affine conic constraints (ACC, see Sec. 6.2) take the form:

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

As a matter of demonstration we will also add the constraint

$$u_1 + u_2 - 1 = 0$$

as an affine conic constraint. It means that to define the all the ACCs we need to produce the following affine expressions (AFE) and store them:

$$u_1,\ u_2,\ x + y + \log(2/A_{\text{wall}}),\ x + z + \log(2/A_{\text{wall}}),\ 1.0,\ u_1 + u_2 - 1.0.$$

We implement it by adding all the affine expressions (AFE) and then picking the ones required for each ACC:

Listing 6.7: Implementation of log-sum-exp as in (6.21).

```
        putvarboundsliceconst(task,1, numvar+1, MSK_BK_FR, -Inf, Inf)

        appendcons(task,3)
        # s0+s1 < 1 <=> log(s0+s1) < 0
        putaijlist(task,
                   [1,1,2,2,3,3],
                   [y, z, x, y, z, y],
                   [1.0, 1.0, 1.0, -1.0, 1.0, -1.0])

        putconbound(task,1,MSK_BK_UP,-Inf,log(Af))
        putconbound(task,2,MSK_BK_RA,log(alpha),log(beta))
        putconbound(task,3,MSK_BK_RA,log(gamma),log(delta))


        let afei = getnumafe(task)+1,
            u1 = getnumvar(task)+1,
            u2 = u1+1,
            afeidx = [1, 2, 3, 3, 4, 4, 6, 6],
            varidx = [u1, u2, x, y, x, z, u1, u2],
            fval   = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
            gfull  = [0.0, 0.0, log(2.0/Aw), log(2.0/Aw), 1.0, -1.0]

            appendvars(task,2)
            appendafes(task,6)

            putvarboundsliceconst(task,u1, u1+2, MSK_BK_FR, -Inf, Inf)

            # Affine expressions appearing in affine conic constraints
            # in this order:
            # u1, u2, x+y+log(2/Awall), x+z+log(2/Awall), 1.0, u1+u2-1.0
            putafefentrylist(task,afeidx, varidx, fval)
            putafegslice(task,afei, afei+6, gfull)

            let dom = appendprimalexpconedomain(task)
```

(continues on next page)

(continued from previous page)

```
                # (u1, 1, x+y+log(2/Awall)) \in EXP
                appendacc(task,dom, [1, 5, 3], nothing)

                # (u2, 1, x+z+log(2/Awall)) \in EXP
                appendacc(task,dom, [2, 5, 4], nothing)
            end
            let dom = appendrzerodomain(task,1)
                # The constraint u1+u2-1 \in \ZERO is added also as an ACC
                appendacc(task,dom, [6], nothing)
            end
        end
```

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 6.8: Source code solving problem (6.20).

```
function max_volume_box(Aw    :: Float64,
                        Af    :: Float64,
                        alpha :: Float64,
                        beta  :: Float64,
                        gamma :: Float64,
                        delta :: Float64)
    numvar = 3 # Variables in original problem
        # Create the optimization task.

    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        # Add variables and constraints
        appendvars(task,numvar)

        x = 1
        y = 2
        z = 3

        # Objective is the sum of three first variables
        putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)
        putcslice(task,1, numvar+1, [1.0,1.0,1.0])

        putvarboundsliceconst(task,1, numvar+1, MSK_BK_FR, -Inf, Inf)

        appendcons(task,3)
        # s0+s1 < 1 <=> log(s0+s1) < 0
        putaijlist(task,
                   [1,1,2,2,3,3],
                   [y, z, x, y, z, y],
                   [1.0, 1.0, 1.0, -1.0, 1.0, -1.0])

        putconbound(task,1,MSK_BK_UP,-Inf,log(Af))
        putconbound(task,2,MSK_BK_RA,log(alpha),log(beta))
        putconbound(task,3,MSK_BK_RA,log(gamma),log(delta))


        let afei = getnumafe(task)+1,
            u1 = getnumvar(task)+1,
```

(continues on next page)

(continued from previous page)

```
            u2 = u1+1,
            afeidx = [1, 2, 3, 3, 4, 4, 6, 6],
            varidx = [u1, u2, x, y, x, z, u1, u2],
            fval   = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
            gfull  = [0.0, 0.0, log(2.0/Aw), log(2.0/Aw), 1.0, -1.0]

            appendvars(task,2)
            appendafes(task,6)

            putvarboundsliceconst(task,u1, u1+2, MSK_BK_FR, -Inf, Inf)

            # Affine expressions appearing in affine conic constraints
            # in this order:
            # u1, u2, x+y+log(2/Awall), x+z+log(2/Awall), 1.0, u1+u2-1.0
            putafefentrylist(task,afeidx, varidx, fval)
            putafegslice(task,afei, afei+6, gfull)

            let dom = appendprimalexpconedomain(task)

                # (u1, 1, x+y+log(2/Awall)) \in EXP
                appendacc(task,dom, [1, 5, 3], nothing)

                # (u2, 1, x+z+log(2/Awall)) \in EXP
                appendacc(task,dom, [2, 5, 4], nothing)
            end
            let dom = appendrzerodomain(task,1)
                # The constraint u1+u2-1 \in \ZERO is added also as an ACC
                appendacc(task,dom, [6], nothing)
            end
        end

        optimize(task)
        writedata(task,"gp1.ptf")

        exp.(getxxslice(task,MSK_SOL_ITR, 1, numvar+1))
    end # maketask
end # max_volume_box
```

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

Listing 6.9: Sample data for problem (6.19).

```
hwd = let Aw    = 200.0,
    Af    = 50.0,
    alpha = 2.0,
    beta  = 10.0,
    gamma = 2.0,
    delta = 10.0

    max_volume_box(Aw, Af, alpha, beta, gamma, delta)
end
println("h=$(hwd[1]) w=$(hwd[2]) d=$(hwd[3])\n");
```

6.7 Semidefinite Optimization

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

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

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

MOSEK can solve semidefinite optimization problems stated in the **primal** form,

$$\begin{array}{lccccc} \text{minimize} & & \sum_{j=0}^{p-1} \langle \overline{C}_j, \overline{X}_j \rangle + \sum_{j=0}^{n-1} c_j x_j + c^f & & & \\ \text{subject to} & l^c_i & \leq \sum_{j=0}^{p-1} \langle \overline{A}_{ij}, \overline{X}_j \rangle + \sum_{j=0}^{n-1} a_{ij} x_j & \leq & u^c_i, & i = 0, \ldots, m-1, \\ & & \sum_{j=0}^{p-1} \langle \overline{F}_{ij}, \overline{X}_j \rangle + \sum_{j=0}^{n-1} f_{ij} x_j + g_i & \in & \mathcal{K}_i, & i = 0, \ldots, q-1, \\ & l^x_j & \leq x_j & \leq & u^x_j, & j = 0, \ldots, n-1, \\ & & x \in \mathcal{K}, \overline{X}_j \in \mathcal{S}^{r_j}_+, & & & j = 0, \ldots, p-1 \end{array} \tag{6.22}$$

where the problem has p symmetric positive semidefinite variables $\overline{X}_j \in \mathcal{S}^{r_j}_+$ of dimension r_j. The symmetric coefficient matrices $\overline{C}_j \in \mathcal{S}^{r_j}$ and $\overline{A}_{i,j} \in \mathcal{S}^{r_j}$ are used to specify PSD terms in the linear objective and the linear constraints, respectively. The symmetric coefficient matrices $\overline{F}_{i,j} \in \mathcal{S}^{r_j}$ are used to specify PSD terms in the affine conic constraints. Note that q ((6.22)) is the total dimension of all the cones, i.e. $q = \dim(\mathcal{K}_1 \times \ldots \times \mathcal{K}_k)$, given there are k ACCs. We use standard notation for the matrix inner product, i.e., for $A, B \in \mathbb{R}^{m \times n}$ we have

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

In addition to the primal form presented above, semidefinite problems can be expressed in their **dual** form. Constraints in this form are usually called **linear matrix inequalities** (LMIs). LMIs can be easily specified in **MOSEK** using the vectorized positive semidefinite cone which is defined as:

- Vectorized semidefinite domain:

$$\mathcal{S}^{d,\text{vec}}_+ = \left\{ (x_1, \ldots, x_{d(d+1)/2}) \in \mathbb{R}^n \ : \ \text{sMat}(x) \in \mathcal{S}^d_+ \right\},$$

where $n = d(d+1)/2$ and,

$$\text{sMat}(x) = \begin{bmatrix} x_1 & x_2/\sqrt{2} & \cdots & x_d/\sqrt{2} \\ x_2/\sqrt{2} & x_{d+1} & \cdots & x_{2d-1}/\sqrt{2} \\ \cdots & \cdots & \cdots & \cdots \\ x_d/\sqrt{2} & x_{2d-1}/\sqrt{2} & \cdots & x_{d(d+1)/2} \end{bmatrix},$$

or equivalently

$$\mathcal{S}^{d,\text{vec}}_+ = \left\{ \text{sVec}(X) \ : \ X \in \mathcal{S}^d_+ \right\},$$

where

$$\mathrm{sVec}(X) = (X_{11}, \sqrt{2}X_{21}, \ldots, \sqrt{2}X_{d1}, X_{22}, \sqrt{2}X_{32}, \ldots, X_{dd}).$$

In other words, the domain consists of vectorizations of the lower-triangular part of a positive semidefinite matrix, with the non-diagonal elements additionally rescaled. LMIs can be expressed by restricting appropriate affine expressions to this cone type.

For other types of cones supported by **MOSEK**, see Sec. 15.8 and the other tutorials in this chapter. Different cone types can appear together in one optimization problem.

We demonstrate the setup of semidefinite variables and their coefficient matrices in the following examples:

- Sec. 6.7.1: A problem with one semidefinite variable and linear and conic constraints.
- Sec. 6.7.2: A problem with two semidefinite variables with a linear constraint and bound.
- Sec. 6.7.3: A problem with linear matrix inequalities and the vectorized semidefinite domain.

6.7.1 Example SDO1

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

$$\begin{array}{lll}
\text{minimize} & \left\langle \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, \overline{X} \right\rangle + x_0 & \\
\text{subject to} & \left\langle \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \overline{X} \right\rangle + x_0 & = \quad 1, \\
& \left\langle \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \overline{X} \right\rangle + x_1 + x_2 & = \quad 1/2, \\
& x_0 \geq \sqrt{{x_1}^2 + {x_2}^2}, & \overline{X} \succeq 0,
\end{array} \tag{6.23}$$

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

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

and an affine conic constraint (ACC) $(x_0, x_1, x_2) \in \mathcal{Q}^3$. The objective is to minimize

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

subject to the two linear constraints

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

Setting up the linear and conic part

The linear and conic parts (constraints, variables, objective, ACC) are set up using the methods described in the relevant tutorials; Sec. 6.1, Sec. 6.2. Here we only discuss the aspects directly involving semidefinite variables.

Appending semidefinite variables

First, we need to declare the number of semidefinite variables in the problem, similarly to the number of linear variables and constraints. This is done with the function *appendbarvars*.

```
    # Append matrix variables of sizes in 'BARVARDIM'.
    # The variables will initially be fixed at zero.
    appendbarvars(task,barvardim)
```

Appending coefficient matrices

Coefficient matrices $\overline{C}_j$ and $\overline{A}_{ij}$ are constructed as weighted combinations of sparse symmetric matrices previously appended with the function *appendsparsesymmat*.

```
    symc  = appendsparsesymmat(task,barvardim[1],
                               barci,
                               barcj,
                               barcval)

    syma0 = appendsparsesymmat(task,barvardim[1],
                               barai[1],
                               baraj[1],
                               baraval[1])

    syma1 = appendsparsesymmat(task,barvardim[1],
                               barai[2],
                               baraj[2],
                               baraval[2])
```

The arguments specify the dimension of the symmetric matrix, followed by its description in the sparse triplet format. Only lower-triangular entries should be included. The function produces a unique index of the matrix just entered in the collection of all coefficient matrices defined by the user.

After one or more symmetric matrices have been created using *appendsparsesymmat*, we can combine them to set up the objective matrix coefficient $\overline{C}_j$ using *putbarcj*, which forms a linear combination of one or more symmetric matrices. In this example we form the objective matrix directly, i.e. as a weighted combination of a single symmetric matrix.

```
    putbarcj(task,1, [symc], [1.0])
```

Similarly, a constraint matrix coefficient $\overline{A}_{ij}$ is set up by the function *putbaraij*.

```
    putbaraij(task,1, 1, [syma0], [1.0])
    putbaraij(task,2, 1, [syma1], [1.0])
```

Retrieving the solution

After the problem is solved, we read the solution using *getbarxj*:

```
        xx = getxx(task,MSK_SOL_ITR)
        barx = getbarxj(task,MSK_SOL_ITR, 1)
```

The function returns the half-vectorization of $\overline{X}_j$ (the lower triangular part stacked as a column vector), where the semidefinite variable index j is passed as an argument.

Source code

Listing 6.10: Source code solving problem (6.23).

```
using Mosek
using Printf, SparseArrays

printstream(msg::String) = print(msg)

# Bound keys for constraints
bkc = [ MSK_BK_FX
        MSK_BK_FX]

# Bound values for constraints
blc = [1.0, 0.5]
buc = [1.0, 0.5]


A = sparse( [1,2,2],[1,2,3],[1.0, 1.0, 1.0])
conesub = [1, 2, 3]

barci = [1, 2, 2, 3, 3]
barcj = [1, 1, 2, 2, 3]
barcval = [2.0, 1.0, 2.0, 1.0, 2.0]

barai   = Any[ [1, 2, 3],
               [1, 2, 3, 2, 3, 3] ]
baraj   = Any[ [1, 2, 3],
               [1, 1, 1, 2, 2, 3] ]
baraval = Any[ [1.0, 1.0, 1.0],
               [1.0, 1.0, 1.0, 1.0, 1.0, 1.0] ]

numvar = 3
numcon = length(bkc)
barvardim = [3]

# Create a task object and attach log stream printer
maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    putstreamfunc(task,MSK_STREAM_LOG,printstream)

    # Append 'numvar' variables.
    # The variables will initially be fixed at zero (x=0).
    appendvars(task,numvar)

    # Append 'numcon' empty constraints.
    # The constraints will initially have no bounds.
    appendcons(task,numcon)

    # Append matrix variables of sizes in 'BARVARDIM'.
    # The variables will initially be fixed at zero.
    appendbarvars(task,barvardim)

    # Set the linear term c_0 in the objective.
    putcj(task, 1, 1.0)

    # Set the bounds on variable j
```

(continues on next page)

(continued from previous page)

```
    # blx[j] <= x_j <= bux[j]
    putvarboundsliceconst(task,1,numvar+1,
                          MSK_BK_FR,
                          -Inf,
                          +Inf)

    # Set the bounds on constraints.
    # blc[i] <= constraint_i <= buc[i]
    putconboundslice(task,1,numcon+1, bkc,blc,buc)



    # Append the conic quadratic cone
    let afei = getnumafe(task)+1,
        dom = appendquadraticconedomain(task,3)

        appendafes(task,3)
        putafefentrylist(task,[1,2,3],
                             [1,2,3],
                             [1.0,1.0,1.0])
        appendaccseq(task,dom,afei,nothing)
    end


    # Input row i of A
    putacolslice(task,1,numvar+1,
                 A.colptr[1:numvar], A.colptr[2:numvar+1],
                 A.rowval,A.nzval)


    symc  = appendsparsesymmat(task,barvardim[1],
                               barci,
                               barcj,
                               barcval)

    syma0 = appendsparsesymmat(task,barvardim[1],
                               barai[1],
                               baraj[1],
                               baraval[1])

    syma1 = appendsparsesymmat(task,barvardim[1],
                               barai[2],
                               baraj[2],
                               baraval[2])

    putbarcj(task,1, [symc], [1.0])

    putbaraij(task,1, 1, [syma0], [1.0])
    putbaraij(task,2, 1, [syma1], [1.0])

    # Input the objective sense (minimize/maximize)
    putobjsense(task,MSK_OBJECTIVE_SENSE_MINIMIZE)

    # Solve the problem and print summary
    optimize(task)
    writedata(task,"sdo1.ptf")
```

(continues on next page)

(continued from previous page)

```
    solutionsummary(task,MSK_STREAM_MSG)

    # Get status information about the solution
    prosta = getprosta(task,MSK_SOL_ITR)
    solsta = getsolsta(task,MSK_SOL_ITR)


    if solsta == MSK_SOL_STA_OPTIMAL
        # Output a solution
        xx = getxx(task,MSK_SOL_ITR)
        barx = getbarxj(task,MSK_SOL_ITR, 1)

        @printf("Optimal solution: \n  xx = %s\n  barx = %s\n", xx',barx')
    elseif solsta == MSK_SOL_STA_DUAL_INFEAS_CER
        println("Primal or dual infeasibility.\n")
    elseif solsta == MSK_SOL_STA_PRIM_INFEAS_CER
        println("Primal or dual infeasibility.\n")
    elseif  solsta == MSK_SOL_STA_UNKNOWN
        println("Unknown solution status")
    else
        println("Other solution status")
    end
end
```

6.7.2 Example SDO2

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

$$\begin{array}{lrcl} \text{minimize} & \langle C_1, \overline{X}_1\rangle + \langle C_2, \overline{X}_2\rangle & & \\ \text{subject to} & \langle A_1, \overline{X}_1\rangle + \langle A_2, \overline{X}_2\rangle & = & b, \\ & (\overline{X}_2)_{01} & \leq & k, \\ & \overline{X}_1, \overline{X}_2 & \succeq & 0. \end{array} \tag{6.24}$$

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

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

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

are constant symmetric matrices.

Note that this problem does not contain any scalar variables, but they could be added in the same fashion as in Sec. 6.7.1.

Other than in Sec. 6.7.1 we don't append coefficient matrices separately but we directly input all nonzeros in each constraint and all nonzeros in the objective at once. Every term of the form $(\overline{A}_{i,j})_{k,l}(\overline{X}_j)_{k,l}$ is determined by four indices (i, j, k, l) and a coefficient value $v = (\overline{A}_{i,j})_{k,l}$. Here i is the number of the constraint in which the term appears, j is the index of the semidefinite variable it involves and (k, l) is the position in that variable. This data is passed in the call to *putbarablocktriplet*. Note that only the lower triangular part should be specified explicitly, that is one always has $k \geq l$. Semidefinite terms $(\overline{C}_j)_{k,l}(\overline{X}_j)_{k,l}$ of the objective are specified in the same way in *putbarcblocktriplet* but only include (j, k, l) and v.

For explanations of other data structures used in the example see Sec. 6.7.1.

The code representing the above problem is shown below.

Listing 6.11: Implementation of model (6.24).

```
using Mosek

# Input data
let numcon    = 2,              # Number of constraints.
    numbarvar = 2,
    dimbarvar = Int32[3, 4],         # Dimension of semidefinite variables

    # Objective coefficients concatenated
    Cj = Int32[ 1, 1, 2, 2, 2, 2 ],    # Which symmetric variable (j)
    Ck = Int32[ 1, 3, 1, 2, 2, 3 ],    # Which entry (k,l)->v
    Cl = Int32[ 1, 3, 1, 1, 2, 3 ],
    Cv = [ 1.0, 6.0, 1.0, -3.0, 2.0, 1.0 ],

    # Equality constraints coefficients concatenated
    Ai = Int32[ 1, 1, 1, 1, 1, 1 ],    # Which constraint (i = 0)
    Aj = Int32[ 1, 1, 1, 2, 2, 2 ],    # Which symmetric variable (j)
    Ak = Int32[ 1, 3, 3, 2, 2, 4 ],    # Which entry (k,l)->v
    Al = Int32[ 1, 1, 3, 1, 2, 4 ],
    Av = [ 1.0, 1.0, 2.0, 1.0, -1.0, -3.0 ],

    # The second constraint - one-term inequality
    A2i = Int32[ 2 ],                          # Which constraint (i = 1)
    A2j = Int32[ 2 ],                          # Which symmetric variable (j = 1)
    A2k = Int32[ 2 ],                          # Which entry A(1,0) = A(0,1) = 0.5
    A2l = Int32[ 1 ],
    A2v = [ 0.5 ],

    bkc = [ MSK_BK_FX,
            MSK_BK_UP ],
    blc = [ 23.0, 0.0 ],
    buc = [ 23.0, -3.0 ]

    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        # Append numcon empty constraints.
        # The constraints will initially have no bounds.
        appendcons(task,numcon)

        # Append numbarvar semidefinite variables.
        appendbarvars(task,dimbarvar)

        # Set objective (6 nonzeros).
        putbarcblocktriplet(task,Cj, Ck, Cl, Cv)

        # Set the equality constraint (6 nonzeros).
        putbarablocktriplet(task,Ai, Aj, Ak, Al, Av)

        # Set the inequality constraint (1 nonzero).
        putbarablocktriplet(task,A2i, A2j, A2k, A2l, A2v)

        # Set constraint bounds
        putconboundslice(task,1, 3, bkc, blc, buc)
```

(continues on next page)

(continued from previous page)

```
        # Run optimizer
        optimize(task)
        solutionsummary(task,MSK_STREAM_MSG)

        solsta = getsolsta(task,MSK_SOL_ITR)

        if solsta == MSK_SOL_STA_OPTIMAL
            # Retrieve the soution for all symmetric variables
            println("Solution (lower triangular part vectorized):")
            for i in 1:numbarvar
                barx = getbarxj(task,MSK_SOL_ITR, i)
                println("X$i: $barx")
            end
        elseif solsta == MSK_SOL_STA_DUAL_INFEAS_CER || if solsta == MSK_SOL_STA_PRIM_
↪INFEAS_CER
            println("Primal or dual infeasibility certificate found.")
        elseif solsta == MSK_SOL_STA_UNKNOWN
            println("The status of the solution could not be determined.")
        else
            println("Other solution status.")
        end
        end
    end
end
```

6.7.3 Example SDO_LMI: Linear matrix inequalities and the vectorized semidefinite domain

The standard form of a semidefinite problem is usually either based on semidefinite variables (primal form) or on linear matrix inequalities (dual form). However, **MOSEK** allows mixing of these two forms, as shown in (6.25)

$$\begin{array}{lll} \text{minimize} & \left\langle \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \overline{X} \right\rangle + x_0 + x_1 + 1 & \\ \text{subject to} & \left\langle \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \overline{X} \right\rangle - x_0 - x_1 & \in \mathbb{R}^1_{\geq 0}, \\ & x_0 \begin{bmatrix} 0 & 1 \\ 1 & 3 \end{bmatrix} + x_1 \begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \succeq 0, \\ & \overline{X} \succeq 0. & \end{array} \tag{6.25}$$

The first affine expression is restricted to a linear domain and could also be modelled as a linear constraint (instead of an ACC). The lower triangular part of the linear matrix inequality (second constraint) can be vectorized and restricted to the *MSK_DOMAIN_SVEC_PSD_CONE*. This allows us to express the constraints in (6.25) as the affine conic constraints shown in (6.26).

$$\begin{array}{lll} \left\langle \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \overline{X} \right\rangle + \begin{bmatrix} -1 & -1 \end{bmatrix} x + \begin{bmatrix} 0 \end{bmatrix} & \in \mathbb{R}^1_{\geq 0}, \\ \begin{bmatrix} 0 & 3 \\ \sqrt{2} & \sqrt{2} \\ 3 & 0 \end{bmatrix} x + \begin{bmatrix} -1 \\ 0 \\ -1 \end{bmatrix} & \in \mathcal{S}^{3,\text{vec}}_+ \end{array} \tag{6.26}$$

Vectorization of the LMI is performed as explained in Sec. 15.8.

Setting up the linear part

The linear parts (objective, constraints, variables) and the semidefinite terms in the linear expressions are defined exactly as shown in the previous examples.

Setting up the affine conic constraints with semidefinite terms

To define the affine conic constraints, we first set up the affine expressions. The F matrix and the g vector are defined as usual. Additionally, we specify the coefficients for the semidefinite variables. The semidefinite coefficients shown in (6.26) are setup using the function *putafebarfblocktriplet*.

```
        # barF block triplets
        putafebarfblocktriplet(task,barf_i, barf_j, barf_k, barf_l, barf_v)
```

These affine expressions are then included in their corresponding domains to construct the affine conic constraints. Lastly, the ACCs are appended to the task.

```
        # Append R+ domain and the corresponding ACC
        appendacc(task,appendrplusdomain(task,1), [1], nothing)
        # Append SVEC_PSD domain and the corresponding ACC
        appendacc(task,appendsvecpsdconedomain(task,3), [2,3,4], nothing)
```

Source code

Listing 6.12: Source code solving problem (6.25).

```
using Mosek

let numafe      = 4,  # Number of affine expressions.
    numvar      = 2,  # Number of scalar variables
    dimbarvar = [2],          # Dimension of semidefinite cone
    lenbarvar = [2 * (2 + 1) / 2], # Number of scalar SD variables
    barc_j  = [1, 1],
    barc_k  = [1, 2],
    barc_l  = [1, 2],
    barc_v  = [1.0, 1.0],

    afeidx  = [1, 1, 2, 3, 3, 4],
    varidx  = [1, 2, 2, 1, 2, 1],
    f_val   = Float64[-1, -1, 3, sqrt(2.0), sqrt(2.0), 3],
    g       = Float64[0, -1, 0, -1],

    barf_i = [1, 1],
    barf_j = [1, 1],
    barf_k = [1, 2],
    barf_l = [1, 1],
    barf_v = [0.0, 1.0]

    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        # Append 'NUMAFE' empty affine expressions.
        appendafes(task,numafe)

        # Append 'NUMVAR' variables.
        # The variables will initially be fixed at zero (x=0).
        appendvars(task,numvar)

        # Append 'NUMBARVAR' semidefinite variables.
```

(continues on next page)

(continued from previous page)

```
        appendbarvars(task,dimbarvar)

        # Optionally add a constant term to the objective.
        putcfix(task,1.0)

        # Set the linear term c_j in the objective.
        putcj(task,1, 1.0)
        putcj(task,2, 1.0)

        for j in 1:numvar
            putvarbound(task,j, MSK_BK_FR, -0.0, 0.0)
        end
        # Set the linear term barc_j in the objective.
        putbarcblocktriplet(task, barc_j, barc_k, barc_l, barc_v)

        # Set up the affine conic constraints

        # Construct the affine expressions
        # F matrix
        putafefentrylist(task,afeidx, varidx, f_val)
        # g vector
        putafegslice(task,1, 5, g)

        # barF block triplets
        putafebarfblocktriplet(task,barf_i, barf_j, barf_k, barf_l, barf_v)

        # Append R+ domain and the corresponding ACC
        appendacc(task,appendrplusdomain(task,1), [1], nothing)
        # Append SVEC_PSD domain and the corresponding ACC
        appendacc(task,appendsvecpsdconedomain(task,3), [2,3,4], nothing)

        # Run optimizer
        optimize(task)

        # Print a summary containing information
        # about the solution for debugging purposes
        solutionsummary(task,MSK_STREAM_MSG)

        solsta = getsolsta(task,MSK_SOL_ITR)

        if solsta == MSK_SOL_STA_OPTIMAL
            xx = getxx(task,MSK_SOL_ITR)
            barx = getbarxj(task,MSK_SOL_ITR,1);    # Request the interior solution.
            println("Optimal primal solution, x = $xx, barx = $barx")
        elseif solsta == MSK_SOL_STA_PRIM_INFEAS_CER || solsta == MSK_SOL_STA_DUAL_
↪INFEAS_CER
            println("Primal or dual infeasibility certificate found.")
        elseif solsta == MSK_SOL_STA_UNKNOWN
            println("The status of the solution could not be determined.")
        else
            println("Other solution status.")
        end
    end
end
```

6.8 Integer Optimization

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

6.8.1 Example MILO1

We use the example

$$\begin{array}{lllll} \text{maximize} & x_0 + 0.64x_1 & & & \\ \text{subject to} & 50x_0 + 31x_1 & \leq & 250, & \\ & 3x_0 - 2x_1 & \geq & -4, & \\ & x_0, x_1 \geq 0 & & \text{and integer} & \end{array} \tag{6.27}$$

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

First, the integrality constraints are imposed using the function *putvartype* or one of its bulk analogues:

```
    # Define variables to be integers
    putvartypelist(task,[ 1, 2 ],
                   [ MSK_VAR_TYPE_INT, MSK_VAR_TYPE_INT ])
```

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

```
    # Set max solution time
    putdouparam(task,MSK_DPAR_MIO_MAX_TIME, 60.0)
```

The complete source for the example is listed Listing 6.13. Please note that when we fetch the solution then the integer solution is requested by using *MSK_SOL_ITG*. No dual solution is defined for integer optimization problems.

Listing 6.13: Source code implementing problem (6.27).

```
using Mosek
using Printf, SparseArrays

# Define a stream printer to grab output from MOSEK
printstream(msg::String) = print(msg)


bkc = [ MSK_BK_UP, MSK_BK_LO  ]
blc = [      -Inf,      -4.0  ]
buc = [     250.0,       Inf  ]

bkx = [ MSK_BK_LO, MSK_BK_LO  ]
blx = [       0.0,       0.0  ]
bux = [       Inf,       Inf  ]

c   = [       1.0,      0.64 ]

A    = sparse( [ 1, 1, 2, 2],
               [ 1, 2, 1, 2],
               [ 50.0, 31.0,
                 3.0, -2.0] )
```

(continues on next page)

(continued from previous page)

```
numvar = length(bkx)
numcon = length(bkc)


# Create a task
maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    # Attach a printer to the task
    putstreamfunc(task,MSK_STREAM_LOG,printstream)

    # Append 'numcon' empty constraints.
    # The constraints will initially have no bounds.
    appendcons(task,numcon)

    #Append 'numvar' variables.
    # The variables will initially be fixed at zero (x=0).
    appendvars(task,numvar)

    # Set the linear term c_j in the objective.
    putclist(task,[1:numvar;],c)

    # Set the bounds on variables
    # blx[j] <= x_j <= bux[j]
    putvarboundslice(task,1,numvar+1,bkx,blx,bux)

    # Input columns of A
    putacolslice(task,1,numvar+1, A.colptr[1:numvar],A.colptr[2:numvar+1],A.rowval,A.
↪nzval)

    putconboundslice(task,1,numcon+1,bkc,blc,buc)

    # Input the objective sense (minimize/maximize)
    putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)

    # Define variables to be integers
    putvartypelist(task,[ 1, 2 ],
                   [ MSK_VAR_TYPE_INT, MSK_VAR_TYPE_INT ])

    # Set max solution time
    putdouparam(task,MSK_DPAR_MIO_MAX_TIME, 60.0)

    # Optimize the task
    optimize(task)

    writedata(task,"milo1.ptf")

    # Print a summary containing information
    # about the solution for debugging purposes
    solutionsummary(task,MSK_STREAM_MSG)

    prosta = getprosta(task,MSK_SOL_ITG)
    solsta = getsolsta(task,MSK_SOL_ITG)

    if solsta == MSK_SOL_STA_INTEGER_OPTIMAL
        # Output a solution
```

(continues on next page)

(continued from previous page)

```
        xx = getxx(task,MSK_SOL_ITG)
        @printf("Optimal solution: %s\n", xx')
    elseif solsta == MSK_SOL_STA_UNKNOWN
        println("Unknown solution status")
    else
        println("Other solution status")
    end

end
```

6.8.2 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.

There are two modes for **MOSEK** to utilize an initial solution.

- **A complete solution.** **MOSEK** will first try to check if the current value of the primal variable solution is a feasible point. The solution can either come from a previous solver call or can be entered by the user, however the full solution with values for all variables (both integer and continuous) must be provided. This check is always performed and does not require any extra action from the user. The outcome of this process can be inspected via information items *MSK_IINF_MIO_INITIAL_FEASIBLE_SOLUTION* and *MSK_DINF_MIO_INITIAL_FEASIBLE_SOLUTION_OBJ*, and via the `Initial feasible solution objective` entry in the log.
- **A partial integer solution.** **MOSEK** can also try to construct a feasible solution by fixing integer variables to the values provided by the user (rounding if necessary) and optimizing over the remaining continuous variables. In this setup the user must provide initial values for all integer variables. This action is only performed if the parameter *MSK_IPAR_MIO_CONSTRUCT_SOL* is switched on. 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.

In the following example we focus on inputting a partial integer solution.

$$\begin{array}{ll} \text{maximize} & 7x_0 + 10x_1 + x_2 + 5x_3 \\ \text{subject to} & x_0 + x_1 + x_2 + x_3 \leq 2.5 \\ & x_0, x_1, x_2 \in \mathbb{Z} \\ & x_0, x_1, x_2, x_3 \geq 0 \end{array} \tag{6.28}$$

Solution values can be set using *putxx*, *putxxslice* or similar .

Listing 6.14: Implementation of problem (6.28) specifying an initial solution.

```
        # Assign values to integer variables
        # We only set that slice of xx
        putxxslice(task,MSK_SOL_ITG, 1, 4, [1.0, 1.0, 0.0])

        # Request constructing the solution from integer variable values
        putintparam(task,MSK_IPAR_MIO_CONSTRUCT_SOL, MSK_ON)
```

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

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

The same information can be obtained from the API:

Listing 6.15: Retrieving information about usage of initial solution

```
constr = getintinf(task,MSK_IINF_MIO_CONSTRUCT_SOLUTION)
constrVal = getdouinf(task,MSK_DINF_MIO_CONSTRUCT_SOLUTION_OBJ)
println("Construct solution utilization: $constr")
println("Construct solution objective: $constrVal")
```

6.8.3 Example MICO1

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

$$\begin{array}{ll} \text{minimize} & x^2 + y^2 \\ \text{subject to} & x \geq e^y + 3.8, \\ & x, y \text{ integer}. \end{array} \tag{6.29}$$

The canonical conic formulation of (6.29) suitable for Optimizer API for Julia is

$$\begin{array}{lll} \text{minimize} & t & \\ \text{subject to} & (t, x, y) \in \mathcal{Q}^3 & (t \geq \sqrt{x^2 + y^2}) \\ & (x - 3.8, 1, y) \in K_{\exp} & (x - 3.8 \geq e^y) \\ & x, y \text{ integer}, & \\ & t \in \mathbb{R}. & \end{array} \tag{6.30}$$

Listing 6.16: Implementation of problem (6.30).

```
maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    # Directs the log task stream to the user specified
    # method task_msg_obj.stream
    appendvars(task,3);    # x, y, t
    x=1
    y=2
    t=3
    putvarboundsliceconst(task,1, 4, MSK_BK_FR, -Inf, Inf)

    # Integrality constraints for x, y
    putvartypelist(task,
                   [x,y],
                   [MSK_VAR_TYPE_INT,MSK_VAR_TYPE_INT])

    # Set up the affine expressions
    # x, x-3.8, y, t, 1.0
    appendafes(task,5)
    putafefentrylist(task,
                     [1,2,3,4],
                     [x,x,y,t],
                     [1,1,1,1])
    putafegslice(task,1, 6, Float64[0, -3.8, 0, 0, 1.0])

    # Add constraint (x-3.8, 1, y) \in \EXP
    appendacc(task,appendprimalexpconedomain(task), Int64[2, 5, 3], nothing)

    # Add constraint (t, x, y) \in \QUAD
    appendacc(task,appendquadraticconedomain(task,3), Int64[4, 1, 3], nothing)
```

(continues on next page)

(continued from previous page)

```
    # Objective
    putobjsense(task,MSK_OBJECTIVE_SENSE_MINIMIZE)
    putcj(task,t, 1.0)

    # Optimize the task
    optimize(task)
    solutionsummary(task,MSK_STREAM_MSG)

    xx = getxxslice(task,MSK_SOL_ITG, 1, 3)
    println("x = $(xx[1]) y = $(xx[2])")


end
```

Error and solution status handling were omitted for readability.

6.9 Disjunctive constraints

A **disjunctive constraint (DJC)** involves of a number of affine conditions combined with the logical operators or ($\vee$) and optionally and ($\wedge$) into a formula in *disjunctive normal form*, that is a disjunction of conjunctions. Specifically, a disjunctive constraint has the form of a disjunction

$$T_1 \text{ or } T_2 \text{ or} \cdots \text{ or } T_t \tag{6.31}$$

where each T_i is written as a conjunction

$$T_i = T_{i,1} \text{ and } T_{i,2} \text{ and} \cdots \text{ and } T_{i,s_i} \tag{6.32}$$

and each $T_{i,j}$ is an affine condition (affine equation or affine inequality) of the form $D_{ij}x + d_{ij} \in \mathcal{D}_{ij}$ with $\mathcal{D}_{ij}$ being one of the affine domains from Sec. 15.8.1. A disjunctive constraint (DJC) can therefore be succinctly written as

$$\bigvee_{i=1}^{t} \bigwedge_{j=1}^{s_i} T_{i,j} \tag{6.33}$$

where each $T_{i,j}$ is an affine condition.

Each T_i is called a **term** or **clause** of the disjunctive constraint and t is the number of terms. Each condition $T_{i,j}$ is called a **simple term** and s_i is called the **size** of the i-th term.

A disjunctive constraint is satisfied if at least one of its terms (clauses) is satisfied. A term (clause) is satisfied if all of its constituent simple terms are satisfied. A problem containing DJCs will be solved by the mixed-integer optimizer.

Note that nonlinear cones are not allowed as one of the domains $\mathcal{D}_{ij}$ inside a DJC.

6.9.1 Applications

Disjunctive constraints are a convenient and expressive syntactical tool. Then can be used to phrase many constructions appearing especially in mixed-integer modelling. Here are some examples.

- **Complementarity.** The condition $xy = 0$, where x, y are scalar variables, is equivalent to

  $$x = 0 \text{ or } y = 0.$$

  It is a DJC with two terms, each of size 1.

- **Semicontinuous variable.** A semicontinuous variable is a scalar variable which takes values in $\{0\} \cup [a, +\infty]$. This can be expressed as

  $$x = 0 \text{ or } x \geq a.$$

  It is again a DJC with two terms, each of size 1.

- **Exact absolute value.** The constraint $t = |x|$ is not convex, but can be written as

$$(x \geq 0 \text{ and } t = x) \text{ or } (x \leq 0 \text{ and } t = -x)$$

  It is a DJC with two terms, each of size 2.

- **Indicator.** Suppose z is a Boolean variable. Then we can write the indicator constraint $z = 1 \implies a^T x \leq b$ as

$$(z = 1 \text{ and } a^T x \leq b) \text{ or } (z = 0)$$

  which is a DJC with two terms, of sizes, respectively, 2 and 1.

- **Piecewise linear functions.** Suppose $a_1 \leq \cdots \leq a_{k+1}$ and $f : [a_1, a_{k+1}] \to \mathbb{R}$ is a piecewise linear function, given on the i-th of k intervals $[a_i, a_{i+1}]$ by a different affine expression $f_i(x)$. Then we can write the constraint $y = f(x)$ as

$$\bigvee_{i=1}^{k} (a_i \leq y \text{ and } y \leq a_{i+1} \text{ and } y - f_i(x) = 0)$$

  making it a DJC with k terms, each of size 3.

On the other hand most DJCs are equivalent to a mixed-integer linear program through a big-M reformulation. In some cases, when a suitable big-M is known to the user, writing such a formulation directly may be more efficient than formulating the problem as a DJC. See Sec. 13.4.5 for a discussion of this topic.

Disjunctive constraints can be added to any problem which includes linear constraints, affine conic constraints (without semidefinite domains) or integer variables.

6.9.2 Example DJC1

In this tutorial we will consider the following sample demonstration problem:

$$\begin{array}{ll} \text{minimize} & 2x_0 + x_1 + 3x_2 + x_3 \\ \text{subject to} & x_0 + x_1 + x_2 + x_3 \geq -10, \\ & \begin{pmatrix} x_0 - 2x_1 \leq -1 \\ \text{and} \\ x_2 = x_3 = 0 \end{pmatrix} \text{ or } \begin{pmatrix} x_2 - 3x_3 \leq -2 \\ \text{and} \\ x_0 = x_1 = 0 \end{pmatrix}, \\ & x_i = 2.5 \text{ for at least one } i \in \{0, 1, 2, 3\}. \end{array} \tag{6.34}$$

The problem has two DJCs: the first one has 2 terms. The second one, which we can write as $\bigvee_{i=0}^{3}(x_i = 2.5)$, has 4 terms (clauses).

We begin by expressing problem (6.34) in the format where all simple terms are of the form $D_{ij}x+d_{ij} \in \mathcal{D}_{ij}$, that is of the form *a sequence of affine expressions belongs to a linear domain*:

$$\begin{array}{ll} \text{minimize} & 2x_0 + x_1 + 3x_2 + x_3 \\ \text{subject to} & x_0 + x_1 + x_2 + x_3 \geq -10, \\ & \begin{pmatrix} x_0 - 2x_1 + 1 \in \mathbb{R}^1_{\leq 0} \\ \text{and} \\ (x_2, x_3) \in 0^2 \end{pmatrix} \text{ or } \begin{pmatrix} x_2 - 3x_3 + 2 \in \mathbb{R}^1_{\leq 0} \\ \text{and} \\ (x_0, x_1) \in 0^2 \end{pmatrix}, \\ & (x_0 - 2.5 \in 0^1) \text{ or } (x_1 - 2.5 \in 0^1) \text{ or } (x_2 - 2.5 \in 0^1) \text{ or } (x_3 - 2.5 \in 0^1), \end{array} \tag{6.35}$$

where 0^n denotes the n-dimensional zero domain and $\mathbb{R}^n_{\leq 0}$ denotes the n-dimensional nonpositive orthant, as in Sec. 15.8.

Now we show how to add the two DJCs from (6.35). This involves three steps:

- storing the affine expressions which appear in the DJCs,
- creating the required domains, and
- combining the two into the description of the DJCs.

Readers familiar with Sec. 6.2 will find that the process is completely analogous to the process of adding affine conic constraints (ACCs). In fact we would recommend Sec. 6.2 as a means of familiarizing with the structures used here at a slightly lower level of complexity.

6.9.3 Step 1: add affine expressions

In the first step we need to store all affine expressions appearing in the problem, that is the rows of the expressions $D_{ij}x + d_{ij}$. In problem (6.35) the disjunctive constraints contain altogether the following affine expressions:

$$\begin{array}{ll} (0) & x_0 - 2x_1 + 1 \\ (1) & x_2 - 3x_3 + 2 \\ (2) & x_0 \\ (3) & x_1 \\ (4) & x_2 \\ (5) & x_3 \\ (6) & x_0 - 2.5 \\ (7) & x_1 - 2.5 \\ (8) & x_2 - 2.5 \\ (9) & x_3 - 2.5 \end{array} \tag{6.36}$$

To store affine expressions (**AFE** for short) **MOSEK** provides a matrix $\mathbf{F}$ and a vector $\mathbf{g}$ with the understanding that every row of

$$\mathbf{F}x + \mathbf{g}$$

defines one affine expression. The API functions with infix `afe` are used to operate on $\mathbf{F}$ and $\mathbf{g}$, add rows, add columns, set individual elements, set blocks etc. similarly to the methods for operating on the A matrix of linear constraints. The storage matrix $\mathbf{F}$ is a sparse matrix, therefore only nonzero elements have to be explicitly added.

Remark: the storage $\mathbf{F}, \mathbf{g}$ may, but does not have to be, kept in the same order in which the expressions enter DJCs. In fact in (6.36) we have chosen to list the linear expressions in a different, convenient order. It is also possible to store some expressions only once if they appear multiple times in DJCs.

Given the list (6.36), we initialize the AFE storage as (only nonzeros are listed and for convenience we list the content of (6.36) alongside in the leftmost column):

$$\begin{array}{ll} (0) & x_0 - 2x_1 + 1 \\ (1) & x_2 - 3x_3 + 2 \\ (2) & x_0 \\ (3) & x_1 \\ (4) & x_2 \\ (5) & x_3 \\ (6) & x_0 - 2.5 \\ (7) & x_1 - 2.5 \\ (8) & x_2 - 2.5 \\ (9) & x_3 - 2.5 \end{array} \quad \mathbf{F} = \begin{bmatrix} 1 & -2 & & \\ & & 1 & -3 \\ 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \\ 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}, \quad \mathbf{g} = \begin{bmatrix} 1 \\ 2 \\ \\ \\ \\ \\ -2.5 \\ -2.5 \\ -2.5 \\ -2.5 \end{bmatrix}. \tag{6.37}$$

Initially $\mathbf{F}$ and $\mathbf{g}$ are empty (have 0 rows). We construct them as follows. First, we append a number of empty rows:

```
numafe = 10
appendafes(task,numafe)
```

We now have $\mathbf{F}$ and $\mathbf{g}$ with 10 rows of zeros and we fill them up to obtain (6.37).

```
fafeidx = Int64[1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10]
fvaridx = Int32[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
fval    = Float64[1.0, -2.0, 1.0, -3.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
g       = Float64[1.0, 2.0, 0.0, 0.0, 0.0, 0.0, -2.5, -2.5, -2.5, -2.5]

putafefentrylist(task,fafeidx, fvaridx, fval)
putafegslice(task,1, numafe+1, g)
```

We have now created the matrices from (6.37). Note that at this point we have *not defined any DJCs yet*. All we did was define some affine expressions and place them in a generic AFE storage facility to be used later.

6.9.4 Step 2: create domains

Next, we create all the domains $\mathcal{D}_{ij}$ appearing in all the simple terms of all DJCs. Domains are created with functions with infix `domain`. In the case of (6.35) there are three different domains appearing:

$$0^1,\ 0^2,\ \mathbb{R}^1_{\leq 0}.$$

We create them with the corresponding functions:

```
zero1   = appendrzerodomain(task,1)
zero2   = appendrzerodomain(task,2)
rminus1 = appendrminusdomain(task,1)
```

The function returns a domain index, which is just the position in the list of all domains (potentially) created for the problem. At this point the domains are just stored in the list of domains, but not yet used for anything.

6.9.5 Step 3: create the actual disjunctive constraints

We are now in position to create the disjunctive constraints. DJCs are created with functions with infix `djc`. The function *`appenddjcs`* will append a number of initially empty DJCs to the task:

```
numdjc = 2
appenddjcs(task,numdjc)
```

We can then define each disjunction with the method *`putdjc`*. It will require the following data:

- the list `termsizelist` of the sizes of all terms of the DJC,
- the list `afeidxlist` of indices of AFEs to be used in the constraint. These are the row numbers in $\mathbf{F}, \mathbf{g}$ which contain the required affine expressions.
- the list `domidxlist` of the domains for all the simple terms.

For example, consider the first DJC of (6.35). Below we format this DJC by replacing each affine expression with the index of that expression in (6.37) and each domain with its index we obtained in Step 2:

$$\begin{array}{ccccc} (x_0 - 2x_1 + 1 \in \mathbb{R}^1_{\leq 0} \text{ and } (x_2, x_3) \in 0^2) & \text{or} & (x_2 - 3x_3 + 2 \in \mathbb{R}^1_{\leq 0} \text{ and } (x_0, x_1) \in 0^2) \\ \underbrace{((0) \in \texttt{rminus1} \text{ and } ((4),(5)) \in \texttt{zero2})}_{\texttt{term of size 2}} & \text{or} & \underbrace{((1) \in \texttt{rminus1} \text{ and } ((2),(3)) \in \texttt{zero2})}_{\texttt{term of size 2}} \end{array} \tag{6.38}$$

It implies that the DJC will be represented by the following data:

- `termsizelist = [2, 2]`,
- `afeidxlist = [0, 4, 5, 1, 2, 3]`,
- `domidxlist = [rminus1, zero2, rminus1, zero2]`.

The code adding this DJC will therefore look as follows:

```
putdjc(task,1,                                          # DJC index
            [rminus1, zero2, rminus1, zero2],           # Domains     (domidxlist)
            [1, 5, 6, 2, 3, 4],                         # AFE indices (afeidxlist)
            nothing,                                    # Unused
            [2, 2] )                                    # Term sizes
↪(termsizelist)
```

Note that number of AFEs used in `afeidxlist` must match the sum of dimensions of all the domains (here: $6 == 1+2+1+2$) and the number of domains must match the sum of all term sizes (here: $4 == 2+2$).

For similar reasons the second DJC of problem (6.35) will have the description:

$$\begin{array}{cccccccc} x_0 - 2.5 \in 0^1 & \text{or} & x_1 - 2.5 \in 0^1 & \text{or} & x_2 - 2.5 \in 0^1 & \text{or} & x_3 - 2.5 \in 0^1 \\ \underbrace{(6) \in \texttt{zero1}}_{\texttt{term of size 1}} & \text{or} & \underbrace{(7) \in \texttt{zero1}}_{\texttt{term of size 1}} & \text{or} & \underbrace{(8) \in \texttt{zero1}}_{\texttt{term of size 1}} & \text{or} & \underbrace{(9) \in \texttt{zero1}}_{\texttt{term of size 1}} \end{array} \tag{6.39}$$

- `termsizelist = [1, 1, 1, 1]`,
- `afeidxlist = [6, 7, 8, 9]`,
- `domidxlist = [zero1, zero1, zero1, zero1]`.

```
putdjc(task,2,                                  # DJC index
            [zero1, zero1, zero1, zero1],       # Domains     (domidxlist)
            [7, 8, 9, 10],                      # AFE indices (afeidxlist)
            nothing,                            # Unused
            [1, 1, 1, 1] )                      # Term sizes  (termidxlist)
```

This completes the setup of the disjunctive constraints.

6.9.6 Example DJC1 full code

We refer to Sec. 6.1 for instructions how to initialize a **MOSEK** session, add variables and set up the objective and linear constraints. All else that remains is to call the solver with *optimize* and retrieve the solution with *getxx*. Since our problem contains a DJC, and thus is solved by the mixed-integer optimizer, we fetch the integer solution. The full code solving problem (6.34) is shown below.

Listing 6.17: Full code of example DJC1.

```
using Mosek


maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    # Append free variables
    numvar = 4
    appendvars(task,numvar)
    putvarboundsliceconst(task,1, numvar+1, MSK_BK_FR, -Inf, Inf)

    # The linear part: the linear constraint
    appendcons(task,1)
    putarow(task,1,[1:numvar...], ones(numvar))
    putconbound(task,1,MSK_BK_LO, -10.0, -10.0)

    # The linear part: objective
    putobjsense(task,MSK_OBJECTIVE_SENSE_MINIMIZE)
    putclist(task,[1:numvar...], Float64[2, 1, 3, 1])

    # Fill in the affine expression storage F, g
    numafe = 10
    appendafes(task,numafe)

    fafeidx = Int64[1, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    fvaridx = Int32[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
    fval    = Float64[1.0, -2.0, 1.0, -3.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
    g       = Float64[1.0, 2.0, 0.0, 0.0, 0.0, 0.0, -2.5, -2.5, -2.5, -2.5]

    putafefentrylist(task,fafeidx, fvaridx, fval)
    putafegslice(task,1, numafe+1, g)

    # Create domains
    zero1   = appendrzerodomain(task,1)
    zero2   = appendrzerodomain(task,2)
    rminus1 = appendrminusdomain(task,1)
```

(continues on next page)

(continued from previous page)

```
    # Append disjunctive constraints
    numdjc = 2
    appenddjcs(task,numdjc)

    # First disjunctive constraint
    putdjc(task,1,                                      # DJC index
                [rminus1, zero2, rminus1, zero2],       # Domains     (domidxlist)
                [1, 5, 6, 2, 3, 4],                     # AFE indices (afeidxlist)
                nothing,                                # Unused
                [2, 2] )                                # Term sizes
↪(termsizelist)

    # Second disjunctive constraint
    putdjc(task,2,                                      # DJC index
                [zero1, zero1, zero1, zero1],           # Domains     (domidxlist)
                [7, 8, 9, 10],                          # AFE indices (afeidxlist)
                nothing,                                # Unused
                [1, 1, 1, 1] )                          # Term sizes  (termidxlist)

    # Useful for debugging
    writedata(task,"djc1.ptf")                      # Write file in human-readable
↪format

    # Solve the problem
    optimize(task)

    # Print a summary containing information
    # about the solution for debugging purposes
    solutionsummary(task,MSK_STREAM_MSG)

    # Get status information about the solution
    sta = getsolsta(task,MSK_SOL_ITG)

    println(sta)
    @assert sta == MSK_SOL_STA_INTEGER_OPTIMAL

    xx = getxx(task,MSK_SOL_ITG)

    println("Optimal solution:")
    for i in 1:numvar
        println("x[$i]=$(xx[i])")
    end
    @assert maximum(abs.(xx - [0.0, 0.0, -12.5, 2.5])) < 1e-7
```

The answer is

```
[0, 0, -12.5, 2.5]
```

6.9.7 Summary and extensions

In this section we presented the most basic usage of the affine expression storage $\mathbf{F}, \mathbf{g}$ to input *affine expressions* used together with *domains* to create *disjunctive constraints* (DJC). Now we briefly point out additional features of his interface which can be useful in some situations for more demanding users. They will be demonstrated in various examples in other tutorials and case studies in this manual.

- It is important to remember that $\mathbf{F}, \mathbf{g}$ has *only a storage function* and during the DJC construction we can pick an arbitrary list of row indices and place them in a domain. It means for example that:
  - It is not necessary to store the AFEs in the same order they will appear in DJCs.
  - The same AFE index can appear more than once in one and/or more conic constraints (this can be used to reduce storage if the same affine expression is used in multiple DJCs).
  - The $\mathbf{F}, \mathbf{g}$ storage can even include rows that are not presently used in any DJC.
- Domains can be reused: multiple DJCs can use the same domain. On the other hand the same type of domain can appear under many `domidx` positions. In this sense the list of created domains also plays only a *storage role*: the domains are only used when they enter a DJC.
- The same affine expression storage $\mathbf{F}, \mathbf{g}$ is shared between disjunctive constraints and affine conic constraints (ACCs, see Sec. 6.2).
- When defining an DJC an additional constant vector b can be provided to modify the constant terms coming from $\mathbf{g}$ but only for this particular DJC. This could be useful to reduce $\mathbf{F}$ storage space if, for example, many expressions $D^T x - b_i$ with the same linear part $D^T x$, but varying constant terms b_i, are to be used throughout DJCs.

6.10 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:

$$\begin{array}{lccccll} \text{minimize} & & & \frac{1}{2}x^T Q^o x + c^T x + c^f & & & \\ \text{subject to} & l_k^c & \leq & \frac{1}{2}x^T Q^k x + \sum_{j=0}^{n-1} a_{k,j} x_j & \leq & u_k^c, & k = 0, \ldots, m-1, \\ & l_j^x & \leq & x_j & \leq & u_j^x, & j = 0, \ldots, n-1. \end{array} \tag{6.40}$$

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 kth constraint must be of the form

$$l_k^c \leq \frac{1}{2} x^T Q^k x + \sum_{j=0}^{n-1} a_{k,j} x_j \tag{6.41}$$

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.

6.10.1 Example: Quadratic Objective

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

$$\begin{array}{lll} \text{minimize} & & x_1^2 + 0.1x_2^2 + x_3^2 - x_1x_3 - x_2 \\ \text{subject to} & 1 \leq & x_1 + x_2 + x_3 \\ & 0 \leq & x. \end{array} \tag{6.42}$$

The matrix formulation of (6.42) 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 (6.40) 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 (6.42).

Setting up the linear part

The linear parts (constraints, variables, objective) are set up using exactly the same methods as for linear problems, and we refer to Sec. 6.1 for all the details. The same applies to technical aspects such as defining an optimization task, retrieving the solution and so on.

Setting up the quadratic objective

The quadratic objective is specified using the function *putqobj*. Since Q^o is symmetric only the lower triangular part of Q^o is inputted. In fact entries from above the diagonal may *not* appear in the input.

The lower triangular part of the matrix Q^o is specified using an unordered sparse triplet format (for details, see Sec. 15.1.4):

```
# Set up and input quadratic objective
qsubi = [ 1,   2,    3,   3   ]
qsubj = [ 1,   2,    1,   3   ]
qval  = [ 2.0, 0.2, -1.0, 2.0 ]
```

Please note that

- only non-zero elements are specified (any element not specified is 0 by definition),
- the order of the non-zero elements is insignificant, and
- *only* the lower triangular part should be specified.

Finally, this definition of Q^o is loaded into the task:

```
putqobj(task,qsubi,qsubj,qval)
```

Source code

Listing 6.18: Source code implementing problem (6.42).

```
using Mosek
using Printf, SparseArrays

# Define a stream printer to grab output from MOSEK
```

(continues on next page)

(continued from previous page)

```
bkc   = [ MSK_BK_LO ]
blc   = [ 1.0 ]
buc   = [ Inf ]

bkx   = [ MSK_BK_LO, MSK_BK_LO, MSK_BK_LO ]
blx   = [ 0.0,  0.0, 0.0 ]
bux   = [ Inf,  Inf, Inf ]
numvar = length(bkx)
numcon = length(bkc)

c     = [ 0.0, -1.0, 0.0 ]
A     = sparse( [ 1, 1, 1 ],
                [ 1, 2, 3 ],
                [ 1.0, 1.0, 1.0 ],
                numcon, numvar )

maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

    # Append 'numcon' empty constraints.
    # The constraints will initially have no bounds.
    appendcons(task,numcon)

    # Append 'numvar' variables.
    # The variables will initially be fixed at zero (x=0).
    appendvars(task,numvar)

    # Set the linear term c_j in the objective.
    putclist(task,[1:numvar;],c)

    # Set the bounds on variable j
    # blx[j] <= x_j <= bux[j]
    putvarboundslice(task,1,numvar+1,bkx,blx,bux)

    putacolslice(task,1,numvar+1,
                 A.colptr[1:numvar],A.colptr[2:numvar+1],
                 A.rowval,A.nzval)

    # Set up and input quadratic objective
    qsubi = [ 1,   2,    3,   3   ]
    qsubj = [ 1,   2,    1,   3   ]
    qval  = [ 2.0, 0.2, -1.0, 2.0 ]

    putqobj(task,qsubi,qsubj,qval)

    putobjsense(task,MSK_OBJECTIVE_SENSE_MINIMIZE)

    # Optimize
    r = optimize(task)
    # Print a summary containing information
    # about the solution for debugging purposes
    solutionsummary(task,MSK_STREAM_MSG)

    prosta = getprosta(task,MSK_SOL_ITR)
```

(continues on next page)

(continued from previous page)

```
    solsta = getsolsta(task,MSK_SOL_ITR)

    if solsta == MSK_SOL_STA_OPTIMAL
        xx = getxx(task,MSK_SOL_ITR)
        println("Optimal solution:")
        println(xx)
    elseif solsta in [ MSK_SOL_STA_DUAL_INFEAS_CER,
                       MSK_SOL_STA_PRIM_INFEAS_CER ]
        println("Primal or dual infeasibility certificate found.\n")
    elseif solsta == MSK_SOL_STA_UNKNOWN
        println("Unknown solution status")
    else
        @printf("Other solution status (%d)\n",solsta)
    end

end
```

6.10.2 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 (6.41).

Consider the problem:

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

This is equivalent to

$$\begin{array}{lcc} \text{minimize} & & \frac{1}{2}x^T Q^o x + c^T x \\ \text{subject to} & \frac{1}{2}x^T Q^0 x + Ax & \geq \quad b, \\ & x \geq 0, & \end{array} \tag{6.43}$$

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

To add quadratic terms to the constraints we use the function *putqconk*.

```
    qsubi = [  1,    2,    3,   3   ]
    qsubj = [  1,    2,    3,   1   ]
    qval  = [ -2.0, -2.0, -0.2, 0.2 ]

    # put Q^0 in constraint with index 0.

    putqconk(task,1, qsubi,qsubj, qval)
```

While *putqconk* adds quadratic terms to a specific constraint, it is also possible to input all quadratic terms in one chunk using the *putqcon* function.

Source code

Listing 6.19: Implementation of the quadratically constrained problem (6.43).

```
using Mosek
using Printf
# Since the actual value of Infinity is ignores, we define it solely
# for symbolic purposes:


# Set up and input bounds and linear coefficients
bkc   = [ MSK_BK_LO ]
blc   = [ 1.0 ]
buc   = [ Inf ]

bkx   = [ MSK_BK_LO
          MSK_BK_LO
          MSK_BK_LO ]
blx   = [ 0.0,  0.0, 0.0 ]
bux   = [ Inf,  Inf, Inf ]

c     = [ 0.0, -1.0, 0.0 ]

asub  = [ 1 ,2, 3 ]
aval  = [ 1.0, 1.0, 1.0 ]

numvar = length(bkx)
numcon = length(bkc)


# Create a task
maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    # Append 'numcon' empty constraints.
    # The constraints will initially have no bounds.
    appendcons(task,numcon)

    #Append 'numvar' variables.
    # The variables will initially be fixed at zero (x=0).
    appendvars(task,numvar)

    #Optionally add a constant term to the objective.
    putcfix(task,0.0)
    # Set the linear term c_j in the objective.
    putclist(task,[1:numvar;],c)

    # Set the bounds on variable j
    # blx[j] <= x_j <= bux[j]
    putvarboundslice(task,1,numvar+1,bkx,blx,bux)
    # Input column j of A
    putarow(task,1,asub,aval)

    putconbound(task,1,bkc[1],blc[1],buc[1])

    # Set up and input quadratic objective
```

(continues on next page)

(continued from previous page)

```
    qsubi = [ 1,   2,    3,   3   ]
    qsubj = [ 1,   2,    1,   3   ]
    qval  = [ 2.0, 0.2, -1.0, 2.0 ]

    putqobj(task,qsubi,qsubj,qval)

    # The lower triangular part of the Q^0
    # matrix in the first constraint is specified.
    # This corresponds to adding the term
    # - x0^2 - x1^2 - 0.1 x2^2 + 0.2 x0 x2

    qsubi = [  1,    2,    3,   3   ]
    qsubj = [  1,    2,    3,   1   ]
    qval  = [ -2.0, -2.0, -0.2, 0.2 ]

    # put Q^0 in constraint with index 0.

    putqconk(task,1, qsubi,qsubj, qval)

    # Input the objective sense (minimize/maximize)
    putobjsense(task,MSK_OBJECTIVE_SENSE_MINIMIZE)

    # Optimize the task
    optimize(task)
    # Print a summary containing information
    # about the solution for debugging purposes
    solutionsummary(task,MSK_STREAM_MSG)
    prosta = getprosta(task,MSK_SOL_ITR)
    solsta = getsolsta(task,MSK_SOL_ITR)

    if solsta == MSK_SOL_STA_OPTIMAL
        # Output a solution
        xx = getxx(task,MSK_SOL_ITR)
        @printf("Optimal solution: %s\n", xx')
    elseif solsta == MSK_SOL_STA_DUAL_INFEAS_CER
        println("Primal or dual infeasibility.\n")
    elseif solsta == MSK_SOL_STA_PRIM_INFEAS_CER
        println("Primal or dual infeasibility.\n")
    elseif solsta == MSK_SOL_STA_UNKNOWN
        println("Unknown solution status")
    else
        println("Other solution status")
    end
end
```

6.11 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 [Chvatal83].

Parameter settings (see Sec. 7.5) can also be changed between optimizations.

6.11.1 Example: Production Planning

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

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

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

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

$$\begin{array}{lrcrcrcl} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & & \\ \text{subject to} & 2x_0 & + & 4x_1 & + & 3x_2 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & \leq & 60000, \end{array} \tag{6.44}$$

and

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

Code in Listing 6.20 loads and solves this problem.

Listing 6.20: Setting up and solving problem (6.44)

```
let numcon = 3,
    numvar = 3,
    c      = [1.5, 2.5, 3.0 ],
    bkc    = [ MSK_BK_UP,
               MSK_BK_UP,
               MSK_BK_UP
               ],
    blc    = [ -Inf,
               -Inf,
               -Inf ],
    buc    = [ 100000.0,
               50000.0,
               60000.0
             ],
    bkx    = [ MSK_BK_LO,
               MSK_BK_LO,
               MSK_BK_LO ],
    blx    = [ 0.0, 0.0, 0.0 ],
    bux    = [ +Inf,
               +Inf,
               +Inf ],
    aptrb  = Int64[ 1,4,7 ],
    aptre  = Int64[ 4,7,10 ],
    asub   = Int32[ 1, 2, 3,
                    1, 2, 3,
                    1, 2, 3 ],
    aval   = [ 2.0, 3.0, 2.0,
               4.0, 2.0, 3.0,
               3.0, 3.0, 2.0 ]

    maketask() do task          # 126
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))
        # Append the constraints.
        appendcons(task,numcon)

        # Append the variables.
        appendvars(task,numvar)

        # Put C.
        for j in 1:numvar
            putcj(task,j, c[j])
        end

        # Put constraint bounds
        for i in 1:numcon
            putconbound(task,i, bkc[i], blc[i], buc[i])
        end

        # Put variable bounds.
        for j in 1:numvar
            putvarbound(task,j, bkx[j], blx[j], bux[j])
        end

        # Put A.
```

(continues on next page)

(continued from previous page)

```
putacolslice(task,1,numvar+1,aptrb,aptre,asub,aval)

# A maximization problem
putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)
# Solve the problem
optimize(task)

xx = getxx(task,MSK_SOL_BAS) # Request the basic solution.

println("x = $xx")
```

6.11.2 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 calling the function *putaij* as shown below.

```
putaij(task,1,1, 3.0)
```

The problem now has the form:

$$\begin{array}{lccccccc} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & & \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & \leq & 60000, \end{array} \tag{6.45}$$

and

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

After this operation we can reoptimize the problem.

6.11.3 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 6.21

Listing 6.21: How to add a new variable (column)

```
# Get index of new variable.
varidx = getnumvar(task)+1

# Append a new variable x_3 to the problem
appendvars(task,1)
numvar += 1

# Set bounds on new varaible
putvarbound(task,
            varidx,
            MSK_BK_LO,
            0.0,
            Inf);
```

(continues on next page)

(continued from previous page)

```
        # Change objective
        putcj(task,varidx, 1.0)

        # Put new values in the A matrix
        let acolsub = Int32[1,3],
            acolval = Float64[4.0, 1.0]

            putacol(task,varidx, # column index
                    acolsub,
                    acolval)
        end
```

After this operation the new problem is:

$$\begin{array}{lccccccccc} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & + & 1.0x_3 & & \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & + & 4x_3 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & & & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & + & 1x_3 & \leq & 60000, \end{array} \tag{6.46}$$

and

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

6.11.4 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 6.22: Adding a new constraint.

```
        # Get index of new constraint.
        conidx = getnumcon(task)+1

        # Append a new constraint
        appendcons(task,1)
        numcon += 1;

        # Set bounds on new constraint
        putconbound(task,
                    conidx,
                    MSK_BK_UP,
                    -Inf,
                    30000.0)

        # Put new values in the A matrix
```

(continues on next page)

(continued from previous page)

```
        let arowsub = [1,   2,   3,   4  ],
            arowval = [1.0, 2.0, 1.0, 1.0]

            putarow(task,conidx, # row index
                    arowsub,
                    arowval)
        end
```

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

6.11.5 Changing bounds

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

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

That means we would like to solve the problem:

$$\begin{array}{lrcrcrcrcl} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & + & 1.0x_3 & & \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & + & 4x_3 & \leq & 80000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & & & \leq & 40000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & + & 1x_3 & \leq & 50000, \\ & x_0 & + & 2x_1 & + & x_2 & + & x_3 & \leq & 22000. \end{array} \tag{6.47}$$

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

Listing 6.23: Change constraint bounds.

```
        let newbkc  = [MSK_BK_UP,
                       MSK_BK_UP,
                       MSK_BK_UP,
                       MSK_BK_UP],
            newblc = [ -Inf,
                       -Inf,
                       -Inf,
                       -Inf],
            newbuc = [ 80000.0, 40000.0, 50000.0, 22000.0 ]

            putconboundslice(task,1, numcon+1, newbkc, newblc, newbuc)
        end
```

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

6.11.6 Advanced hot-start

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

```
Optimizer  - hotstart               : yes
```

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

6.12 Parallel optimization

In this section we demonstrate the method *optimizebatch* which is a parallel optimization mechanism built-in in **MOSEK**. It has the following features:

- One license token checked out by the environment will be shared by the tasks.
- It allows to fine-tune the balance between the total number of threads in use by the parallel solver and the number of threads used for each individual task.
- It is very efficient for optimizing a large number of task of similar size, for example tasks obtained by cloning an initial task and changing some coefficients.

In the example below we simply load a few different tasks and optimize them together. When all tasks complete we access the response codes, solutions and other information in the standard way, as if each task was optimized separately.

Listing 6.24: Calling the parallel optimizer.

```
using Mosek

# Example of how to use env.optimizebatch().
# Optimizes tasks whose names were read from command line.
if length(ARGS) < 2
    println("Usage: parallel FILENAME FILENAME [ FILENAME ... ]")
else
    n = length(ARGS)
    makeenv() do env
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        tasks = [ maketask(filename=f) for f in ARGS ]
```

(continues on next page)

(continued from previous page)

```
        # Size of thread pool available for all tasks
        threadpoolsize = 6

        for t in tasks
            putintparam(t,MSK_IPAR_NUM_THREADS, 2)
        end

        # Optimize all the given tasks in parallel
        (trm,res) = optimizebatch(env,
                                  false,          # No race
                                  -1.0,           # No time limit
                                  threadpoolsize,
                                  tasks)          # Array of tasks to optimize

        for (i,t) in enumerate(tasks)
            println("Task $i res $(res[i]) trm $(trm[i]) obj_val $(getdouinf(t,
↪MSK_DINF_INTPNT_PRIMAL_OBJ)) time $(getdouinf(t,MSK_DINF_OPTIMIZER_TIME))")
        end
    end
end
```

Another, slightly more advanced application of the parallel optimizer is presented in Sec. 11.3.

6.13 Retrieving infeasibility certificates

When a continuous problem is declared as primal or dual infeasible, **MOSEK** provides a Farkas-type infeasibility certificate. If, as it happens in many cases, the problem is infeasible due to an unintended mistake in the formulation or because some individual constraint is too tight, then it is likely that infeasibility can be isolated to a few linear constraints/bounds that mutually contradict each other. In this case it is easy to identify the source of infeasibility. The tutorial in Sec. 8.3 has instructions on how to deal with this situation and debug it **by hand**. We recommend Sec. 8.3 as an introduction to infeasibility certificates and how to deal with infeasibilities in general.

Some users, however, would prefer to obtain the infeasibility certificate using Optimizer API for Julia, for example in order to repair the issue automatically, display the information to the user, or perhaps simply because the infeasibility was one of the intended outcomes that should be analyzed in the code.

In this tutorial we show how to obtain such an infeasibility certificate with Optimizer API for Julia in the most typical case, that is when the linear part of a problem is primal infeasible. A Farkas-type primal infeasibility certificate consists of the dual values of linear constraints and bounds. The names of duals corresponding to various parts of the problem are defined in Sec. 12.1.2. Each of the dual values (multipliers) indicates that a certain multiple of the corresponding constraint should be taken into account when forming the collection of mutually contradictory equalities/inequalities.

6.13.1 Example PINFEAS

For the purpose of this tutorial we use the same example as in Sec. 8.3, that is the primal infeasible problem

$$
\begin{array}{lllccccccccccccc}
\text{minimize} & & x_0 & + & 2x_1 & + & 5x_2 & + & 2x_3 & + & x_4 & + & 2x_5 & + & x_6 & & \\
\text{subject to} & s_0: & x_0 & + & x_1 & & & & & & & & & & \leq & 200, & \\
& s_1: & & & & & x_2 & + & x_3 & & & & & & \leq & 1000, & \\
& s_2: & & & & & & & & & x_4 & + & x_5 & + & x_6 \leq & 1000, & \\
& d_0: & x_0 & & & & & & & + & x_4 & & & & = & 1100, & (6.48) \\
& d_1: & & & x_1 & & & & & & & & & & = & 200, & \\
& d_2: & & & & & x_2 & + & & & & & x_5 & & = & 500, & \\
& d_3: & & & & & & & x_3 & + & & & & x_6 & = & 500, & \\
& & & & & & & & & & & & & x_i & \geq & 0. &
\end{array}
$$

Checking infeasible status and adjusting settings

After the model has been solved we check that it is indeed infeasible. If yes, then we choose a threshold for when a certificate value is considered as an important contributor to infeasibility (ideally we would like to list all nonzero duals, but just like an optimal solution, an infeasibility certificate is also subject to floating-point rounding errors). All these steps are demonstrated in the snippet below:

```
    # Check problem status, we use the interior point solution
    if getprosta(task,MSK_SOL_ITR) == MSK_PRO_STA_PRIM_INFEAS
        # Set the tolerance at which we consider a dual value as essential
        eps = 1e-7
```

Going through the certificate for a single item

We can define a fairly generic function which takes an array of lower and upper dual values and all other required data and prints out the positions of those entries whose dual values exceed the given threshold. These are precisely the values we are interested in:

```
"""
Analyzes and prints infeasibility contributing elements
sl - dual values for lower bounds
su - dual values for upper bounds
eps - tolerance for when a nunzero dual value is significant
"""
function analyzeCertificate(sl :: Vector{Float64}, su :: Vector{Float64}, eps ::␣
↪Float64)
    for i in 1:length(sl)
        if abs(sl[i]) > eps
            println("#$i, lower,  dual = $(sl[i])")
        end
        if abs(su[i]) > eps
            println("#$i, upper,  dual = $(su[i])")
        end
    end
end
```

Full source code

All that remains is to call this function for all variable and constraint bounds for which we want to know their contribution to infeasibility. Putting all these pieces together we obtain the following full code:

Listing 6.25: Demonstrates how to retrieve a primal infeasibility certificate.

```
using Mosek

# Set up a simple linear problem from the manual for test purposes
function testProblem(func :: Function)
    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        appendvars(task,7)
        appendcons(task,7);
        putclist(task,
                 Int32[1,2,3,4,5,6,7],
                 Float64[1,2,5,2,1,2,1])
        putaijlist(task,
                   Int32[1,1,2,2,3,3,3,4,4,5,6,6,7,7],
                   Int32[1,2,3,4,5,6,7,1,5,2,3,6,4,7],
```

(continues on next page)

(continued from previous page)

```
                    Float64[1,1,1,1,1,1,1,1,1,1,1,1,1,1])
        putconboundslice(task,
                         1, 8,
                         [MSK_BK_UP,MSK_BK_UP,MSK_BK_UP,MSK_BK_FX,MSK_BK_FX,MSK_BK_FX,
↪MSK_BK_FX],
                         Float64[-Inf, -Inf, -Inf, 1100, 200, 500, 500],
                         Float64[200, 1000, 1000, 1100, 200, 500, 500])
        putvarboundsliceconst(task,1, 8, MSK_BK_LO, 0.0, +Inf)

        func(task)
    end
end

"""
Analyzes and prints infeasibility contributing elements
sl - dual values for lower bounds
su - dual values for upper bounds
eps - tolerance for when a nunzero dual value is significant
"""
function analyzeCertificate(sl :: Vector{Float64}, su :: Vector{Float64}, eps ::␣
↪Float64)
    for i in 1:length(sl)
        if abs(sl[i]) > eps
            println("#$i, lower,  dual = $(sl[i])")
        end
        if abs(su[i]) > eps
            println("#$i, upper,  dual = $(su[i])")
        end
    end
end

# In this example we set up a simple problem
# One could use any task or a task read from a file
testProblem() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    # Useful for debugging
    writedata(task,"pinfeas.ptf");                          # Write file in human-
↪readable format
    # Attach a log stream printer to the task
    putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

    # Perform the optimization.
    optimize(task)
    solutionsummary(task,MSK_STREAM_LOG)

    # Check problem status, we use the interior point solution
    if getprosta(task,MSK_SOL_ITR) == MSK_PRO_STA_PRIM_INFEAS
        # Set the tolerance at which we consider a dual value as essential
        eps = 1e-7
        println("Variable bounds important for infeasibility: ");
        analyzeCertificate(getslx(task,MSK_SOL_ITR), getsux(task,MSK_SOL_ITR), eps)

        println("Constraint bounds important for infeasibility: ")
        analyzeCertificate(getslc(task,MSK_SOL_ITR), getsuc(task,MSK_SOL_ITR), eps)
    else
        println("The problem is not primal infeasible, no certificate to show")
```

(continues on next page)

(continued from previous page)

```
    end
end
```

Running this code will produce the following output:

```
Variable bounds important for infeasibility:
#6: lower, dual = 1.000000e+00
#7: lower, dual = 1.000000e+00
Constraint bounds important for infeasibility:
#1: upper, dual = 1.000000e+00
#3: upper, dual = 1.000000e+00
#4: lower, dual = 1.000000e+00
#5: lower, dual = 1.000000e+00
```

indicating the positions of bounds which appear in the infeasibility certificate with nonzero values. For a more in-depth treatment see the following sections:

- Sec. 11 for more advanced and complicated optimization examples.
- Sec. 11.1 for examples related to portfolio optimization.
- Sec. 12 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 Environment and task

All interaction with Optimizer API for Julia proceeds through one of two entry points: the **MOSEK tasks** and, to a lesser degree the **MOSEK environment** .

7.1.1 Task

The **MOSEK** task provides a representation of one optimization problem. It is the main interface through which all optimization is performed. Many tasks can be created and disposed of in one process.

A typical scenario for working with a task is shown below:

```
maketask() do task
  # Define and solve an optimization problem here
  # ...
```

7.1.2 Environment

The **MOSEK** environment coordinates access to **MOSEK** from the current process. It provides various general functionalities, in particular those related to license management, linear algebra, parallel optimization and certain other auxiliary functions. All tasks are explicitly or implicitly attached to some environment. It is recommended to have at most one environment per process.

Creating an environment is optional and only recommended for those users who will require some of the features it provides. Most users will **NOT need their own environment** and can skip this object. In this case **MOSEK** will internally create a global environment transparently for the user. The user can access this global environment by using the static variants (with no environment argument) of any function that otherwise belongs to the environment.

A typical scenario for working with **MOSEK** through an explicit environment is shown below:

```
makeenv() do env
  # Create one or more tasks
  maketask(env=env) do task
    # Define and solve an optimization problem here
    # ...
```

7.2 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.

7.2.1 Solver termination

The optimizer provides two status codes relevant for error handling:

- **Response code** of type *rescode*. It indicates if any unexpected error (such as an out of memory error, licensing error etc.) has occurred. The expected value for a successful optimization is *MSK_RES_OK*.
- **Termination code**: It provides information about why the optimizer terminated, for instance if a predefined time limit has been reached. These are not errors, but ordinary events that can be expected (depending on parameter settings and the type of optimizer used).

If the optimization was successful then the method *optimize* returns normally and its output is the termination code. If an error occurs then the method throws an exception, which contains the response code. See Sec. 7.3 for how to access it.

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. 7.4.

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

7.2.2 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	*MSK_SOL_BAS*	*MSK_SOL_ITR*	
Nonlinear continuous problem		*MSK_SOL_ITR*	
Problem with integer variables			*MSK_SOL_ITG*

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.

7.2.3 Problem and solution status

Assuming that the optimization terminated without errors, the next important step is to check the problem and solution status and availability of solutions. 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 for each solution can be retrieved with *getprosta* and *getsolsta*, respectively.

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	*MSK_PRO_STA_PRIM_AND_DUAL_FE*	*MSK_SOL_STA_OPTIMAL*
Primal infeasible	*MSK_PRO_STA_PRIM_INFEAS*	*MSK_SOL_STA_PRIM_INFEAS_CER*
Dual infeasible (unbounded)	*MSK_PRO_STA_DUAL_INFEAS*	*MSK_SOL_STA_DUAL_INFEAS_CER*
Uncertain (stall, numerical issues, etc.)	*MSK_PRO_STA_UNKNOWN*	*MSK_SOL_STA_UNKNOWN*

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

Outcome	Problem status	Solution status
Integer optimal	*MSK_PRO_STA_PRIM_FEAS*	*MSK_SOL_STA_INTEGER_OPTIMAL*
Infeasible	*MSK_PRO_STA_PRIM_INFEAS*	*MSK_SOL_STA_UNKNOWN*
Integer feasible point	*MSK_PRO_STA_PRIM_FEAS*	*MSK_SOL_STA_PRIM_FEAS*
No conclusion	*MSK_PRO_STA_UNKNOWN*	*MSK_SOL_STA_UNKNOWN*

7.2.4 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:

- *getprimalobj*, *getdualobj* — the primal and dual objective value.
- *getxx* — solution values for the variables.
- *getsolution* — a full solution with primal and dual values

and many more specialized methods, see the *API reference*.

7.2.5 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.

```
using Mosek


cqo1_ptf = "
Task ''
Objective obj
    Minimize + x4 + x5 + x6
Constraints
    c1 [1] + x1 + x2 + 2 x3
    k1 [QUAD(3)]
        @ac1: + x4
        @ac2: + x1
        @ac3: + x2
    k2 [RQUAD(3)]
        @ac4: + x5
        @ac5: + x6
        @ac6: + x3
Variables
    x4
    x1 [0;+inf]
    x2 [0;+inf]
    x5
    x6
    x3 [0;+inf]
"

try
    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        if length(ARGS) < 1
            readptfstring(task,cqo1_ptf)
        else
            readdata(task,ARGS[1])
        end

        # (Optional) uncomment to see what happens when solution status is unknown
        # putintparam(task,MSK_IPAR_INTPNT_MAX_ITERATIONS, 1)
```

(continues on next page)

(continued from previous page)

```
        # Optimize
        trmcode = optimize(task)
        solutionsummary(task,MSK_STREAM_LOG)

        # We expect solution status OPTIMAL
        solsta = getsolsta(task,MSK_SOL_ITR)

        if solsta == MSK_SOL_STA_OPTIMAL
            # Optimal solution. Fetch and print it.
            println("An optimal interior-point solution is located.")
            numvar = getnumvar(task)
            xx = getxx(task,MSK_SOL_ITR)
            println("x = $xx")
        elseif solsta == MSK_SOL_STA_DUAL_INFEAS_CER
            println("Dual infeasibility certificate found.")
        elseif solsta == MSK_SOL_STA_PRIM_INFEAS_CER
            println("Primal infeasibility certificate found.")
        elseif solsta == MSK_SOL_STA_UNKNOWN
            # The solutions status is unknown. The termination code
            # indicates why the optimizer terminated prematurely.
            println("The solution status is unknown.")
            (symname, desc) = getcodedesc(trmcode)
            println("   Termination code: $symname $desc")
        else
            println("An unexpected solution status $solsta is obtained.")
        end
    end
catch e
    println("En error occurred: $(e.rcode), $(e.msg)")
end
```

7.3 Errors and exceptions

Exceptions

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

- referencing a nonexisting variable (for example with too large index),
- defining an invalid value for a parameter,
- accessing an undefined solution,
- repeating a variable name, etc.

It is therefore a good idea to catch errors. The one case where it is *extremely important* to do so is when *optimize* is invoked. We will say more about this in Sec. 7.2.

The error is contains a *response code* (element of the enum *rescode*) and short diagnostic messages. They can be accessed as in the following example.

```
try
    putdouparam(task,MSK_DPAR_INTPNT_CO_TOL_REL_GAP, -1.0e-7)
catch e
    println("Response code $(e.rcode)")
```

(continues on next page)

(continued from previous page)

```
    println("Message       $(e.msg)")
end
```

It will produce as output:

```
Error in call to put_dou_param: (1216) "The parameter value -1e-07 is too small for␣
↪parameter 'MSK_DPAR_INTPNT_CO_TOL_REL_GAP'.\0"
```

Another way to obtain a human-readable string corresponding to a response code is the method *getcodedesc*. A full list of exceptions, as well as response codes, can be found in the *API reference*.

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. 7.4). 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.

```
using Mosek


cqo1_ptf = "
Task ''
Objective obj
    Minimize + x4 + x5 + x6
Constraints
    c1 [1] + x1 + x2 + 2 x3
    k1 [QUAD(3)]
        @ac1: + x4
        @ac2: + x1
        @ac3: + x2
    k2 [RQUAD(3)]
        @ac4: + x5
        @ac5: + x6
        @ac6: + x3
Variables
    x4
    x1 [0;+inf]
    x2 [0;+inf]
    x5
    x6
    x3 [0;+inf]
"

try
    maketask() do task
```

(continues on next page)

(continued from previous page)

```
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        if length(ARGS) < 1
            readptfstring(task,cqo1_ptf)
        else
            readdata(task,ARGS[1])
        end

        # (Optional) uncomment to see what happens when solution status is unknown
        # putintparam(task,MSK_IPAR_INTPNT_MAX_ITERATIONS, 1)


        # Optimize
        trmcode = optimize(task)
        solutionsummary(task,MSK_STREAM_LOG)

        # We expect solution status OPTIMAL
        solsta = getsolsta(task,MSK_SOL_ITR)

        if solsta == MSK_SOL_STA_OPTIMAL
            # Optimal solution. Fetch and print it.
            println("An optimal interior-point solution is located.")
            numvar = getnumvar(task)
            xx = getxx(task,MSK_SOL_ITR)
            println("x = $xx")
        elseif solsta == MSK_SOL_STA_DUAL_INFEAS_CER
            println("Dual infeasibility certificate found.")
        elseif solsta == MSK_SOL_STA_PRIM_INFEAS_CER
            println("Primal infeasibility certificate found.")
        elseif solsta == MSK_SOL_STA_UNKNOWN
            # The solutions status is unknown. The termination code
            # indicates why the optimizer terminated prematurely.
            println("The solution status is unknown.")
            (symname, desc) = getcodedesc(trmcode)
            println("   Termination code: $symname $desc")
        else
            println("An unexpected solution status $solsta is obtained.")
        end
    end
catch e
    println("En error occurred: $(e.rcode), $(e.msg)")
end
```

7.4 Input/Output

The logging and I/O features are provided mainly by the **MOSEK** task and to some extent by the **MOSEK** environment objects.

7.4.1 Stream logging

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

The log messages are partitioned in three streams:

- messages, *MSK_STREAM_MSG*
- warnings, *MSK_STREAM_WRN*
- errors, *MSK_STREAM_ERR*

These streams are aggregated in the *MSK_STREAM_LOG* stream. A stream handler can be defined for each stream separately.

A stream handler is simply a function which accepts a string. It is attached to the stream using *putstreamfunc* as follows:

```
putstreamfunc(task, MSK_STREAM_LOG, msg -> print(msg))
```

The stream can be detached by calling

```
clearstreamfunc(task,MSK_STREAM_LOG)
```

A log stream can also be redirected to a file:

```
linkfiletostream(task,MSK_STREAM_LOG,"mosek.log",0);
```

After optimization is completed an additional short summary of the solution and optimization process can be printed to any stream using the method *solutionsummary*.

7.4.2 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*.

Each parameter controls the output level of a specific functionality or algorithm. The main switch is *MSK_IPAR_LOG* which affect 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 produce 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.

7.4.3 Saving a problem to a file

An optimization problem can be dumped to a file using the method *writedata*. The file format will be determined from the extension of the filename. 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 a human-readable PTF file (see Sec. 16.5) with

```
writedata(task,"data.ptf")
```

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

```
writedata(task,"data.task.gz")
```

Some remarks:

- 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.

7.4.4 Reading a problem from a file

A problem saved in any of the supported file formats can be read directly into a task using *readdata*. The task must be created in advance. Afterwards the problem can be optimized, modified, etc. If the file contained solutions, then are also imported, but the status of any solution will be set to *MSK_SOL_STA_UNKNOWN* (solutions can also be read separately using *readsolution*). If the file contains parameters, they will be set accordingly.

```
task = maketask(env=env)
try
    readdata(task,"file.task.gz")
    optimize(task)
catch e
    print("Problem reading the file")
end
```

7.5 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. There are three types of parameters depending on the values they take:

- Integer parameters. They take either either simple integer values or values from an enumeration provided for readability and compatibility of the code. Set with *putintparam*.
- Double (floating point) parameters. Set with *putdouparam*.
- String parameters. Set with *putstrparam*.

There are also parameter setting functions which operate fully on symbolic strings containing generic command-line style names of parameters and their values. See the example below. The optimizer will try to convert the given argument to the exact expected type, and will error if that fails.

If an incorrect value is provided then the parameter is left unchanged.

For example, the following piece of code sets up some parameters before solving a problem.

Listing 7.3: Parameter setting example.

```
# Set log level (integer parameter)
putintparam(task,MSK_IPAR_LOG, 1)
# Select interior-point optimizer... (integer parameter)
putintparam(task,MSK_IPAR_OPTIMIZER, MSK_OPTIMIZER_INTPNT)
# ... without basis identification (integer parameter)
putintparam(task,MSK_IPAR_INTPNT_BASIS,MSK_BI_NEVER)
# Set relative gap tolerance (double parameter)
putdouparam(task,MSK_DPAR_INTPNT_CO_TOL_REL_GAP, 1.0e-7)

# The same using explicit string names
putparam(task,"MSK_DPAR_INTPNT_CO_TOL_REL_GAP", "1.0e-7");
putnadouparam(task,"MSK_DPAR_INTPNT_CO_TOL_REL_GAP",  1.0e-7 )

# Incorrect value
try
    putdouparam(task,MSK_DPAR_INTPNT_CO_TOL_REL_GAP, -1.0)
catch
    println("Wrong parameter value")
end
```

Reading parameter values

The functions *getintparam*, *getdouparam*, *getstrparam* can be used to inspect the current value of a parameter, for example:

```
param = getdouparam(task,MSK_DPAR_INTPNT_CO_TOL_REL_GAP)
println("Current value for parameter intpnt_co_tol_rel_gap = $param")
```

7.6 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. 7.7 for details.

Remark

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

Retrieving the values

Values of information items are fetched using one of the methods

- *getdouinf* for a double information item,
- *getintinf* for an integer information item,
- *getlintinf* for a long integer information item.

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

Listing 7.4: Information items example.

```
tm = getdouinf(task,MSK_DINF_OPTIMIZER_TIME)
iter = getintinf(task,MSK_IINF_INTPNT_ITER)
```

7.7 Progress and data callback

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

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

Warning

The callbacks functions *must not* invoke any functions of the solver, environment or task. Otherwise the state of the solver and its outcome are undefined. The only exception is the possibility to retrieve an integer solution, see below.

Retrieving mixed-integer solutions

If the mixed-integer optimizer is used, the callback will take place, in particular, every time an improved integer solution is found. In that case it is possible to retrieve the current values of the best integer solution from within the callback function. It can be useful for implementing complex termination criteria for integer optimization.

7.7.1 Data callback

In the data callback **MOSEK** passes a callback code and values of all information items to a user-defined function. The callback function is called, in particular, at the beginning of each iteration of the interior-point optimizer. For the simplex optimizers *MSK_IPAR_LOG_SIM_FREQ* controls how frequently the call-back is called.

The callback is set by calling the method *putcallbackfunc* with a user-defined function which takes the callback code and values of information items.

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

7.7.2 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 callback(task    :: Mosek.Task,
                  maxtime :: Float64,
                  caller  :: Callbackcode,
                  douinf  :: Vector{Float64},
                  intinf  :: Vector{Int32},
                  lintinf :: Vector{Int64})
        opttime = 0.0

        if caller == MSK_CALLBACK_BEGIN_INTPNT
            println("Starting interior-point optimizer")
        elseif caller == MSK_CALLBACK_INTPNT
            itrn    = intinf[MSK_IINF_INTPNT_ITER]
            pobj    = douinf[MSK_DINF_INTPNT_PRIMAL_OBJ]
            dobj    = douinf[MSK_DINF_INTPNT_DUAL_OBJ]
            stime   = douinf[MSK_DINF_INTPNT_TIME]
            opttime = douinf[MSK_DINF_OPTIMIZER_TIME]

            println("Iterations: $itrn")
            @printf("  Elapsed time: %6.2f(%.2f) ",opttime, stime)
            @printf("  Primal obj.: %-18.6e  Dual obj.: %-18.6e",pobj, dobj)
        elseif caller == MSK_CALLBACK_END_INTPNT
            println("Interior-point optimizer finished.")
        elseif caller == MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX
            println("Primal simplex optimizer started.")
        elseif caller == MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX
            itrn = intinf[MSK_IINF_SIM_PRIMAL_ITER]
            pobj = douinf[MSK_DINF_SIM_OBJ]
            stime = douinf[MSK_DINF_SIM_TIME]
            opttime = douinf[MSK_DINF_OPTIMIZER_TIME]

            println("Iterations: %-3d", itrn)
            println("  Elapsed time: %6.2f(%.2f)",opttime, stime)
            println("  Obj.: %-18.6e", pobj)
        elseif caller == MSK_CALLBACK_END_PRIMAL_SIMPLEX
            println("Primal simplex optimizer finished.")
        elseif caller == MSK_CALLBACK_BEGIN_DUAL_SIMPLEX
            println("Dual simplex optimizer started.")
        elseif caller == MSK_CALLBACK_UPDATE_DUAL_SIMPLEX
            itrn = intinf[MSK_IINF_SIM_DUAL_iter]
            pobj = douinf[MSK_DINF_SIM_OBJ]
            stime = douinf[MSK_DINF_SIM_TIME]
```

(continues on next page)

(continued from previous page)

```
            opttime = douinf[MSK_DINF_OPTIMIZER_TIME]
            println("Iterations: %-3d",itrn)
            println("  Elapsed time: %6.2f(%.2f)",opttime, stime)
            println("  Obj.: %-18.6e",pobj)
        elseif caller == MSK_CALLBACK_END_DUAL_SIMPLEX
            println("Dual simplex optimizer finished.")
        elseif caller == MSK_CALLBACK_NEW_INT_MIO
            println("New integer solution has been located.")
            xx = getxx(task,MSK_SOL_ITG)
            println("  x = $xx")
            println("Obj.: %f", douinf[MSK_DINF_MIO_OBJ_INT])
        end

        if opttime >= maxtime
            # mosek is spending too much time. Terminate it.
            println("Terminating.")
            1
        else
            0
        end
end
```

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

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

```
        putcallbackfunc(task,(caller,dinf,iinf,linf) -> callback(task,0.05,caller,
↪dinf,iinf,linf))
```

7.8 MOSEK OptServer

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

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

The `URL` of the remote server required in all client-side calls should be a string of the form `http://host:port` or `https://host:port`.

7.8.1 Synchronous Remote Optimization

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

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

Source code example

Listing 7.7: Using the OptServer in synchronous mode.

```
using Mosek

if length(ARGS) < 2
    println("Missing argument, syntax is:")
    println("  opt_server_sync inputfile addr [certpath]")
else
    maketask() do task
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        inputfile  = ARGS[1]
        serveraddr = ARGS[2]
        tlscert    = if length(ARGS) < 3 Nothing else ARGS[3] end

        # We assume that a problem file was given as the first command
        # line argument (received in `argv')
        readdata(task,inputfile)

        # Set OptServer URL
        putoptserverhost(task,serveraddr)

        # Path to certificate, if any
        if tlscert !== Nothing
            putstrparam(task,MSK_SPAR_REMOTE_TLS_CERT_PATH, tlscert)

            # Solve the problem remotely, no access token
            trm = optimize(task)

            # Print a summary of the solution
            solutionsummary(task,MSK_STREAM_LOG)
        end
    end
end
```

7.8.2 Asynchronous Remote Optimization

In asynchronous mode the client sends a job to the remote server and the execution of the client code continues. In particular, it is the client's responsibility to periodically check the optimization status and, when ready, fetch the results. The client can also interrupt optimization. The most relevant methods are:

- *asyncoptimize* : Offload the optimization task to a solver server.
- *asyncpoll* : Request information about the status of the remote job.
- *asyncgetresult* : Request the results from a completed remote job.
- *asyncstop* : Terminate a remote job.

Source code example

In the example below the program enters in a polling loop that regularly checks whether the result of the optimization is available.

Listing 7.8: Using the OptServer in asynchronous mode.

```
using Mosek

if length(ARGS) < 2
    println("Missing argument, syntax is:")
    println("  opt_server_async inputfile host:port numpolls [cert]")
else
    filename   = ARGS[1]
    serveraddr = ARGS[2]
    numpolls   = parse(Int,ARGS[3])
    cert       = (if length(ARGS) > 3
                      ARGS[4]
                  else
                      Nothing
                  end)

    token = maketask() do task
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        token = Nothing

        print("reading task from file")
        readdata(task,filename)

        if cert !== Nothing
            putstrparam(task,MSK_SPAR_REMOTE_TLS_CERT_PATH,cert)
        end

        println("Solve the problem remotely (async)")
        asyncoptimize(task,serveraddr,"")
    end

    println("Task token: '$token'")

    maketask() do task
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))
        readdata(task,filename)

        if cert !== Nothing
            putstrparam(task,MSK_SPAR_REMOTE_TLS_CERT_PATH,cert)
        end


        i = 0
        while i < numpolls
            sleep(0.1)

            println("poll $i...")
            respavailable, res, trm = asyncpoll(task, serveraddr, "", token)

            println("done!")
```

(continues on next page)

(continued from previous page)

```
            if respavailable
                println("solution available!")

                respavailable, res, trm = asyncgetresult(task, serveraddr, "", token)

                solutionsummary(task,MSK_STREAM_LOG)
                break
            end
            i = i + 1

            if i == numpolls
                println("max number of polls reached, stopping host.")
                asyncstop(task,serveraddr,"", token)
            end
        end
    end
end
```

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:

- Enable log output. See Sec. 7.4.1 for how to do it. In the simplest case:

  Attach a log handler function to the log stream:

  ```
  putstreamfunc(task, MSK_STREAM_LOG, msg -> print(msg))
  ```

  and include solution summary after the optimization:

  ```
  optimize(task)
  solutionsummary(task,MSK_STREAM_LOG)
  ```

- 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. 7.4.3.
  - use a human-readable text format, preferably `*.ptf` if you want to check the problem structure by hand. Assign names to variables and constraints to make them easier to identify.

    ```
    writedata(task,"data.ptf")
    ```

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

    ```
    writedata(task,"data.task.gz")
    ```

- 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. 13.4.3.

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        : minimize
  Type                   : CONIC (conic optimization problem)
  Constraints            : 234
  Affine conic cons.     : 5348 (6444 rows)
  Disjunctive cons.      : 0
  Cones                  : 0
  Scalar variables       : 20693
  Matrix variables       : 1 (scalarized: 45)
  Integer variables      : 0
```

This can be consulted to eliminate simple errors: wrong objective sense, wrong number of variables etc. Note that some modeling tools can introduce additional variables and constraints to the model and perturb the model even further (such as by dualizing). In most **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. It can also be done directly with the function *analyzeproblem*. 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

** Affine conic constraints (ACC)
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
```

(continues on next page)

(continued from previous page)

```
|F|               nnz: 612301       min=8.29e-06      max=9.31e+01
|g|               nnz: 1203         min=5.00e-03      max=1.00e+00
```

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

- Wrong type of conic constraint was used or it has wrong dimension.
- The bounds for variables or constraints are incorrect or incomplete. Check if you defined bound keys for all variables. A variable for which no bound was defined is by default fixed at 0.
- 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 PTF (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. 7.3 for more details.

8.1.2 Solution summary

The last item in the log is the solution summary. In the Optimizer API it is only printed by invoking the function *solutionsummary*.

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    ␣
↪acc: 3e-11
Dual.    obj: 8.7560521345e+01    nrm: 1e+00    Viol.  con: 5e-09    var: 9e-11    ␣
↪acc: 0e+00
```

It contains the following elements:

- Problem and solution status. For details see Sec. 7.2.3.
- A summary of the primal solution: objective value, infinity norm of the solution vector and maximal violations of variables and constraints of different types. 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 is the distance to the cone.
- 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    ␣
↪acc: 0e+00
Dual.    obj: 3.0119004098e-01    nrm: 5e+07    Viol.  con: 4e-16    var: 1e-01    ␣
↪acc: 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    ␣
↪acc: 0e+00
Dual.    obj: 3.4529720645e+04    nrm: 8e+03    Viol.  con: 1e-04    var: 2e-04    ␣
↪acc: 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    ␣
↪acc: 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    ␣
↪acc: 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.

Some information about the numerical properties of the data 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. It can also be done directly with the function *analyzeproblem*.

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:

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

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

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

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

When solving an optimization problem one typically expects to get an optimal solution, but in some cases, either by design, or (most frequently) due to an error in the formulation, the problem may become infeasible (have no solution at all).

This section

- describes the intuitions behind infeasibility,
- helps to debug (unexpectedly) infeasible problems using the command line tool and by inspecting infeasibility reports and problem data by hand,
- gives some hints for how to modify the formulation to identify the reasons for infeasibility.

If, instead, you want to fetch an infeasibility certificate directly using Optimizer API for Julia, see the tutorial in Sec. 6.13.

An infeasibility certificate is only available for continuous problems, however the hints in Sec. 8.3.4 apply to a large extent also to mixed-integer problems.

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.

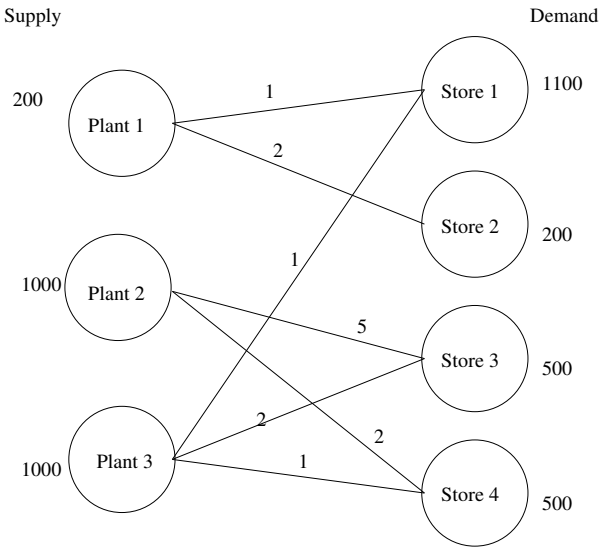


Fig. 8.1: Supply, demand and cost of transportation.

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:

$$\begin{array}{lllrcrcrcrcrcrcl}
\text{minimize} & & x_{11} & + & 2x_{12} & + & 5x_{23} & + & 2x_{24} & + & x_{31} & + & 2x_{33} & + & x_{34} & & \\
\text{subject to} & s_0: & x_{11} & + & x_{12} & & & & & & & & & & & \leq & 200, \\
& s_1: & & & & & x_{23} & + & x_{24} & & & & & & & \leq & 1000, \\
& s_2: & & & & & & & & & x_{31} & + & x_{33} & + & x_{34} & \leq & 1000, \\
& d_0: & x_{11} & & & & & & & + & x_{31} & & & & & = & 1100, \\
& d_1: & & & x_{12} & & & & & & & & & & & = & 200, \\
& d_2: & & & & & x_{23} & + & & & & & x_{33} & & & = & 500, \\
& d_3: & & & & & & & x_{24} & + & & & & & x_{34} & = & 500, \\
& & & & & & & & & & & & & & x_{ij} & \geq & 0.
\end{array} \tag{8.1}$$

Solving problem (8.1) using **MOSEK** will result in an infeasibility status. The infeasibility certificate is contained in the dual variables an can be accessed from an API. The variables and constraints with nonzero solution values form an infeasible subproblem, which frequently is very small. See Sec. 12.1.2 or Sec. 12.2.2 for detailed specifications of infeasibility certificates.

A short infeasibility report can also be printed to the log stream. It can be turned on by setting the parameter *MSK_IPAR_INFEAS_REPORT_AUTO* to *MSK_ON*. 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

```
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             200              0                1
2            s2       none             1000             0                1
3            d0       1100             1100             1                0
4            d1       200              200              1                0


The following bound constraints are involved in the primal infeasibility.


Index        Name     Lower bound      Upper bound      Dual lower       Dual upper
5            x33      0                none             1                0
6            x34      0                none             1                0
```

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`, `d0`, `d1` 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 `d0` and `d1` 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 exists such that the objective tends towards infinity. For example, consider the problem

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

which is dual to (8.1) (and therefore is dual infeasible). The dual infeasibility report may look as follows:

```
Dual infeasibility report

Problem status: The problem is dual infeasible


The following constraints are involved in the dual infeasibility.


Index        Name     Activity         Objective        Lower bound      Upper bound
5            x33      -1                                none             2
6            x34      -1                                none             1


The following variables are involved in the dual infeasibility.


Index        Name     Activity         Objective        Lower bound      Upper bound
0            y1       -1               200              none             0
2            y3       -1               1000             none             0
3            y4       1                1100             none             none
4            y5       1                200              none             none
```

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

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

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

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

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

In the example problem we could

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

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.
- Remember that variables without explicitly initialized bounds are fixed at zero.
- 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 PTF 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.
- Remember that constraints without explicitly initialized bounds are free (no bound).

- 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 PTF 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 3. The **MOSEK** interface for Python must be installed following the installation instructions for Python API or Python Fusion API. The easiest option is

```
pip install Mosek
```

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
`paramdiff`	Show parameters with non-default values
`paramval name`	Show available values for a parameter
`info [name]`	Get an information item
`anapro`	Analyze problem data
`anapro+`	Analyze problem data with the internal analyzer
`hist`	Plot a histogram of problem data
`histsol`	Plot a histogram of the solutions
`spy`	Plot the sparsity pattern of the data matrices
`truncate epsilon`	Truncate small coefficients down to 0
`resobj [fac]`	Rescale objective by a factor
`rlb thr`	Remove large bounds
`anasol`	Analyze solutions
`removeitg`	Remove integrality constraints
`removecones`	Remove all cones and leave just the linear part
`delsol`	Remove solutions
`fixsol solname`	Fix all variables to a specific solution
`fixintsol`	Fix all integer variables to a specific solution
`infsub`	Replace current problem with its infeasible subproblem
`dualize`	Replace current problem with its dual
`writesol basename`	Write solution(s) to file(s) with given basename

continues on next page

Table 8.1 – continued from previous page

Command	Description
`writejsonsol name`	Write solutions to JSON file with given name
`ptf`	Print the PTF representation of the problem
`optserver [url]`	Use an OptServer to optimize
`ls`	List the current folder
`exit`	Leave

Chapter 9

Advanced Numerical Tutorials

9.1 Solving Linear Systems Involving the Basis Matrix

A linear optimization problem always has an optimal solution which is also a basic solution. In an optimal basic solution there are exactly m basic variables where m is the number of rows in the constraint matrix A. Define

$$B \in \mathbb{R}^{m \times m}$$

as a matrix consisting of the columns of A corresponding to the basic variables. The basis matrix B is always non-singular, i.e.

$$\det(B) \neq 0$$

or, equivalently, B^{-1} exists. This implies that the linear systems

$$B\bar{x} = w \tag{9.1}$$

and

$$B^T\bar{x} = w \tag{9.2}$$

each have a unique solution for all w.

MOSEK provides functions for solving the linear systems (9.1) and (9.2) for an arbitrary w. In the next sections we will show how to use **MOSEK** to

- *identify the solution basis*,
- *solve arbitrary linear systems*.

9.1.1 Basis identification

To use the solutions to (9.1) and (9.2) it is important to know how the basis matrix B is constructed.

Internally **MOSEK** employs the linear optimization problem

$$\begin{array}{lrcccl} \text{maximize} & & & c^T x & & \\ \text{subject to} & & & Ax - x^c & = & 0, \\ & l^x & \leq & x & \leq & u^x, \\ & l^c & \leq & x^c & \leq & u^c. \end{array} \tag{9.3}$$

where

$$x^c \in \mathbb{R}^m \text{ and } x \in \mathbb{R}^n.$$

The basis matrix is constructed of m columns taken from

$$\left[\begin{array}{cc} A & -I \end{array} \right].$$

If variable x_j is a basis variable, then the j-th column of A, denoted $a_{:,j}$, will appear in B. Similarly, if x_i^c is a basis variable, then the i-th column of $-I$ will appear in the basis. The ordering of the basis variables and therefore the ordering of the columns of B is arbitrary. The ordering of the basis variables may be retrieved by calling the function *initbasissolve*. This function initializes data structures for later use and returns the indexes of the basic variables in the array `basis`. The interpretation of the `basis` is as follows. If we have

$$\texttt{basis}[i] < \texttt{numcon}$$

then the i-th basis variable is

$$x^c_{\texttt{basis}[i]}.$$

Moreover, the i-th column in B will be the i-th column of $-I$. On the other hand if

$$\texttt{basis}[i] \geq \texttt{numcon},$$

then the i-th basis variable is the variable

$$x_{\texttt{basis}[i]-\texttt{numcon}}$$

and the i-th column of B is the column

$$A_{:,(\texttt{basis}[i]-\texttt{numcon})}.$$

For instance if $\texttt{basis}[0] = 4$ and $\texttt{numcon} = 5$, then since $\texttt{basis}[0] < \texttt{numcon}$, the first basis variable is x_4^c. Therefore, the first column of B is the fourth column of $-I$. Similarly, if $\texttt{basis}[1] = 7$, then the second variable in the basis is $x_{\texttt{basis}[1]-\texttt{numcon}} = x_2$. Hence, the second column of B is identical to $a_{:,2}$.

An example

Consider the linear optimization problem:

$$\begin{array}{lrcl} \text{minimize} & x_0 + x_1 & & \\ \text{subject to} & x_0 + 2x_1 & \leq & 2, \\ & x_0 + x_1 & \leq & 6, \\ & x_0, x_1 \geq 0. & & \end{array} \tag{9.4}$$

Suppose a call to *initbasissolve* returns an array `basis` so that

```
basis[0] = 1,
basis[1] = 2.
```

Then the basis variables are x_1^c and x_0 and the corresponding basis matrix B is

$$\begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix}.$$

Please note the ordering of the columns in B .

Listing 9.1: A program showing how to identify the basis.

```
using Mosek

maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    putobjname(task,"solvebasis")

    putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))
    numcon = 2
    numvar = 2
```

(continues on next page)

(continued from previous page)

```
c = Float64[1.0, 1.0]
ptrb = Int64[1,3]
ptre = Int64[3,4]
asub = Int32[1,2,
             1,2]
aval = Float64[1.0, 1.0,
               2.0, 1.0]
bkc = Boundkey[MSK_BK_UP
               MSK_BK_UP]

blc = Float64[-Inf
              -Inf]
buc = Float64[2.0
              6.0]

bkx = Boundkey[MSK_BK_LO
               MSK_BK_LO]
blx = Float64[0.0
              0.0]

bux = Float64[Inf
              Inf]
w1 = Float64[2.0, 6.0]
w2 = Float64[1.0, 0.0]

inputdata(task,
          Int32(numcon), Int32(numvar),
          c,
          0.0,
          ptrb,
          ptre,
          asub,
          aval,
          bkc,
          blc,
          buc,
          bkx,
          blx,
          bux)

putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)


r = optimize(task)
if r != MSK_RES_OK
    println("Mosek warning: $r")
end

basis = initbasissolve(task)

#List basis variables corresponding to columns of B
varsub = Int32[1, 2]

for i in 1:numcon
    if basis[varsub[i]] < numcon
```

(continues on next page)

(continued from previous page)

```
            println("Basis variable no $i is xc$(basis[i])")
        else
            println("Basis variable no $i is x$(basis[i]-numcon)")

            # solve Bx = w1
            # varsub contains index of non-zeros in b.
            #  On return b contains the solution x and
            # varsub the index of the non-zeros in x.
            nz = 2

            nz = solvewithbasis(task,false, nz, varsub, w1)
            println("nz = $nz")
            println("Solution to Bx = $w1")

            for i in 1:nz
                if basis[varsub[i]] < numcon
                    println("xc $(basis[varsub[i]]) = $(w1[varsub[i]])")
                else
                    println("x$(basis[varsub[i]] - numcon) = $(w1[varsub[i]])")
                end
            end

            # Solve B^Tx = w2
            nz = 1
            varsub[1] = 1

            nz = solvewithbasis(task,true,nz,varsub,w2)
            println(nz)

            println("Solution to B^Tx = $(w2)")

            for i in 1:nz
                if basis[varsub[i]] < numcon
                    println("xc$(basis[varsub[i]]) = $(w2[varsub[i]])")
                else
                    println("x$(basis[varsub[i]] - numcon) = $(w2[varsub[i]])")
                end
            end
        end
    end

end
```

In the example above the linear system is solved using the optimal basis for (9.4) and the original right-hand side of the problem. Thus the solution to the linear system is the optimal solution to the problem. When running the example program the following output is produced.

```
basis[0] = 1
Basis variable no 0 is xc1.
basis[1] = 2
Basis variable no 1 is x0.

Solution to Bx = b:

x0 = 2.000000e+00
xc1 = -4.000000e+00
```

(continues on next page)

(continued from previous page)

```
Solution to B^Tx = c:

x1 = -1.000000e+00
x0 = 1.000000e+00
```

Please note that the ordering of the basis variables is

$$\begin{bmatrix} x_1^c \\ x_0 \end{bmatrix}$$

and thus the basis is given by:

$$B = \begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix}$$

It can be verified that

$$\begin{bmatrix} x_1^c \\ x_0 \end{bmatrix} = \begin{bmatrix} -4 \\ 2 \end{bmatrix}$$

is a solution to

$$\begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} x_1^c \\ x_0 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \end{bmatrix}.$$

9.1.2 Solving arbitrary linear systems

MOSEK can be used to solve an arbitrary (rectangular) linear system

$$Ax = b$$

using the *solvewithbasis* function without optimizing the problem as in the previous example. This is done by setting up an A matrix in the task, setting all variables to basic and calling the *solvewithbasis* function with the b vector as input. The solution is returned by the function.

An example

Below we demonstrate how to solve the linear system

$$\begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{9.5}$$

with two inputs $b = (1, -2)$ and $b = (7, 0)$.

```
using Mosek

function setup(task   :: Mosek.Task,
               aval   :: Vector{Float64},
               asub   :: Vector{Int32},
               ptrb   :: Vector{Int64},
               ptre   :: Vector{Int64},
               numvar :: Int32)

    appendvars(task,numvar)
    appendcons(task,numvar)

    putacolslice(task,1,numvar+1,ptrb,ptre,asub,aval)

    putconboundsliceconst(task,1,numvar+1,MSK_BK_FX,0.0,0.0)
    putvarboundsliceconst(task,1,numvar+1,MSK_BK_FR,-Inf,Inf)
```

(continues on next page)

(continued from previous page)

```
    # Define a basic solution by specifying status keys for variables
    # & constraints.
    deletesolution(task,MSK_SOL_BAS)

    putskcslice(task,MSK_SOL_BAS, 1, numvar+1, fill(MSK_SK_FIX,numvar))
    putskxslice(task,MSK_SOL_BAS, 1, numvar+1, fill(MSK_SK_BAS,numvar))

    return initbasissolve(task)
end

let numcon = Int32(2),
    numvar = Int32(2),

    aval = [ -1.0 ,
             1.0, 1.0 ],
    asub = Int32[ 2,
                  1, 2 ],
    ptrb  = Int64[1, 2],
    ptre  = Int64[2, 4]

    # bsub  = new int[numvar];
    # b     = new double[numvar];
    # basis = new int[numvar];

    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        # Put A matrix and factor A. Call this function only once for a
        # given task.

        basis = setup(task,
                      aval,
                      asub,
                      ptrb,
                      ptre,
                      numvar)

        # now solve rhs
        let b    = Float64[1,-2],
            bsub = Int32[1,2],
            nz = solvewithbasis(task,false, 2, bsub, b)
            println("Solution to Bx = b:")

            # Print solution and show correspondents to original variables in the␣
↪problem
            for i in 1:nz
                if basis[bsub[i]] <= numcon
                    println("This should never happen")
                else
                    println("x $(basis[bsub[i]] - numcon) = $(b[bsub[i]])")
                end
            end
        end
```

(continues on next page)

(continued from previous page)

```
        let b    = Float64[7,0],
            bsub = Int32[1,0],
            nz = solvewithbasis(task,false,1, bsub, b)

            println("Solution to Bx = b:")
            # Print solution and show correspondents to original variables in the␣
↪problem
            for i in 1:nz
                if (basis[bsub[i]] <= numcon)
                    println("This should never happen")
                else
                    println("x $(basis[bsub[i]] - numcon) = $(b[bsub[i]])")
                end
            end
        end
    end
end
```

The most important step in the above example is the definition of the basic solution, where we define the status key for each variable. The actual values of the variables are not important and can be selected arbitrarily, so we set them to zero. All variables corresponding to columns in the linear system we want to solve are set to basic and the slack variables for the constraints, which are all non-basic, are set to their bound.

The program produces the output:

```
Solution to Bx = b:

x1 = 1
x0 = 3

Solution to Bx = b:

x1 = 7
x0 = 7
```

9.2 Computing a Sparse Cholesky Factorization

Given a positive semidefinite symmetric (PSD) matrix

$$A \in \mathbb{R}^{n \times n}$$

it is well known there exists a matrix L such that

$$A = LL^T.$$

If the matrix L is lower triangular then it is called a *Cholesky factorization*. Given A is positive definite (nonsingular) then L is also nonsingular. A Cholesky factorization is useful for many reasons:

- A system of linear equations $Ax = b$ can be solved by first solving the lower triangular system $Ly = b$ followed by the upper triangular system $L^T x = y$.
- A quadratic term $x^T Ax$ in a constraint or objective can be replaced with $y^T y$ for $y = L^T x$, potentially leading to a more robust formulation (see [And13]).

Therefore, **MOSEK** provides a function that can compute a Cholesky factorization of a PSD matrix. In addition a function for solving linear systems with a nonsingular lower or upper triangular matrix is available.

In practice A may be very large with n is in the range of millions. However, then A is typically sparse which means that most of the elements in A are zero, and sparsity can be exploited to reduce the cost of computing the Cholesky factorization. The computational savings depend on the positions of zeros in A. For example, below a matrix A is given together with a Cholesky factor up to 5 digits of accuracy:

$$A = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}, \quad L = \begin{bmatrix} 2.0000 & 0 & 0 & 0 \\ 0.5000 & 0.8660 & 0 & 0 \\ 0.5000 & -0.2887 & 0.8165 & 0 \\ 0.5000 & -0.2887 & -0.4082 & 0.7071 \end{bmatrix}. \tag{9.6}$$

However, if we symmetrically permute the rows and columns of A using a permutation matrix P

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad A' = PAP^T = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 4 \end{bmatrix},$$

then the Cholesky factorization of $A' = L'L'^T$ is

$$L' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

which is sparser than L.

Computing a permutation matrix that leads to the sparsest Cholesky factorization or the minimal amount of work is NP-hard. Good permutations can be chosen by using heuristics, such as the minimum degree heuristic and variants. The function *computesparsecholesky* provided by **MOSEK** for computing a Cholesky factorization has a build in permutation aka. reordering heuristic. The following code illustrates the use of *computesparsecholesky* and *sparsetriangularsolvedense*.

Listing 9.2: How to use the sparse Cholesky factorization routine available in **MOSEK**.

```
    (perm,
     diag,
     lnzc,
     lptrc,
     lensubnval,
     lsubc,
     lvalc) = computesparsecholesky(Int32(0),  #Mosek chooses number of threads
                                    Int32(1),  #Apply reordering heuristic
                                    1.0e-14,  #Singularity tolerance
                                    anzc, aptrc, asubc, avalc)

    printsparse(n, perm, diag, lnzc, lptrc, lensubnval, lsubc, lvalc)

    # Permuted b is stored as x.
    x = b[perm]

    # Compute  inv(L)*x.
    @show x
    sparsetriangularsolvedense(MSK_TRANSPOSE_NO,  lnzc, lptrc, lsubc, lvalc, x)
    # Compute  inv(L^T)*x.
    sparsetriangularsolvedense(MSK_TRANSPOSE_YES, lnzc, lptrc, lsubc, lvalc, x)

    println("\nSolution A x = b, x = $([ x[j] for i in 1:n for j in 1:n if perm[j] ==␣
↪i ])")
```

We can set up the data to recreate the matrix A from (9.6):

```
    # Observe that anzc, aptrc, asubc and avalc only specify the lower triangular␣
↪part.
    n     = Int32(4)
    anzc  = Int32[4, 1, 1, 1]
    asubc = Int32[1, 2, 3, 4,
                  2,
                     3,
                        4]
    aptrc = Int64[1, 5, 6, 7]
    avalc = Float64[4.0, 1.0, 1.0, 1.0,
                         1.0,
                              1.0,
                                   1.0]
    b     = Float64[13.0, 3.0, 4.0, 5.0]
```

and we obtain the following output:

```
Example with positive definite A.
P = [ 3 2 0 1 ]
diag(D) = [ 0.00 0.00 0.00 0.00 ]
L=
1.00 0.00 0.00 0.00
0.00 1.00 0.00 0.00
1.00 1.00 1.41 0.00
0.00 0.00 0.71 0.71

Solution A x = b, x = [ 1.00 2.00 3.00 4.00 ]
```

The output indicates that with the permutation matrix

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

there is a Cholesky factorization $PAP^T = LL^T$, where

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1.4142 & 0 \\ 0 & 0 & 0.7071 & 0.7071 \end{bmatrix}$$

The remaining part of the code solvers the linear system $Ax = b$ for $b = [13, 3, 4, 5]^T$. The solution is reported to be $x = [1, 2, 3, 4]^T$, which is correct.

The second example shows what happens when we compute a sparse Cholesky factorization of a singular matrix. In this example A is a rank 1 matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}^T \tag{9.7}$$

```
    n     = Int32(3)
    anzc  = Int32[3, 2, 1]
    asub  = Int32[0, 1, 2, 1, 2, 2]
    aptr  = Int64[0, 3, 5, ]
    avalc = Float64[1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

Now we get the output

```
P = [ 0 2 1 ]
diag(D) = [ 0.00e+00 1.00e-14 1.00e-14 ]
L=
1.00e+00 0.00e+00 0.00e+00
1.00e+00 1.00e-07 0.00e+00
1.00e+00 0.00e+00 1.00e-07
```

which indicates the decomposition

$$PAP^T = LL^T - D$$

where

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 10^{-7} & 0 \\ 1 & 0 & 10^{-7} \end{bmatrix}, \quad D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 10^{-14} & 0 \\ 0 & 0 & 10^{-14} \end{bmatrix}.$$

Since A is only positive semdefinite, but not of full rank, some of diagonal elements of A are boosted to make it truely positive definite. The amount of boosting is passed as an argument to *computesparsecholesky*, in this case 10^{-14}. Note that

$$PAP^T = LL^T - D$$

where D is a small matrix so the computed Cholesky factorization is exact of slightly perturbed A. In general this is the best we can hope for in finite precision and when A is singular or close to being singular.

We will end this section by a word of caution. Computing a Cholesky factorization of a matrix that is not of full rank and that is not suffciently well conditioned may lead to incorrect results i.e. a matrix that is indefinite may declared positive semidefinite and vice versa.

Chapter 10

Technical guidelines

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

10.1 Memory management and garbage collection

Users should make sure the **MOSEK** objects are fully cleaned up before they go out of scope so that no internally allocated memory leaks. Memory leaks can manifest themselves especially as:

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

should observe that the **MOSEK** task and environment objects are subject to standard garbage collection rules and will be cleaned up by the runtime when they go out of scope.

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 can be set and retrieved using various functions listed in the **Names** section of Sec. 15.2.

Note that file formats impose various restrictions on names, so not all names can be written verbatim to each type of file. If at least one name cannot be written to a given format then generic names and substitutions of offending characters will be used when saving to a file, resulting in a transformation of all names in the problem. See Sec. 16.

10.3 Multithreading

Thread safety

Sharing a task between threads is safe, as long as it is not accessed from more than one thread at a time. Multiple tasks can be created and used in parallel without any problems.

Parallelization

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

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. Note also that not all parts of the algorithm can be parallelized, so there are times when CPU utilization is only 1 even if more cores are available.

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*.

10.4 Timing

Unless otherwise mentioned all parameters, information items and log output entries in **MOSEK** which refer to time measurement are expressed in seconds of wall-clock time.

10.5 Efficiency

Although **MOSEK** is implemented to handle memory efficiently, the user may have valuable knowledge about a problem, which could be used to improve the performance of **MOSEK** This section discusses some tricks and general advice that hopefully make **MOSEK** process your problem faster.

Reduce the number of function calls and avoid input loops

For example, instead of setting the entries in the linear constraint matrix one by one (*putaij*) define them all at once (*putaijlist*) or in convenient large chunks (*putacollist* etc.)

Use one or no environment

Share the environment between all tasks in one process. For most applications you don't need an explicit environment at all.

Read part of the solution

When fetching the solution, data has to be copied from the optimizer to the user's data structures. Instead of fetching the whole solution, consider fetching only the interesting part (see for example *getxxslice* and similar).

Avoiding memory fragmentation

MOSEK stores the optimization problem in internal data structures in the memory. Initially **MOSEK** will allocate structures of a certain size, and as more items are added to the problem the structures are reallocated. For large problems the same structures may be reallocated many times causing memory fragmentation. One way to avoid this is to give **MOSEK** an estimated size of your problem using the functions:

- *putmaxnumvar*. Estimate for the number of variables.
- *putmaxnumcon*. Estimate for the number of constraints.
- *putmaxnumbarvar*. Estimate for the number of semidefinite matrix variables.
- *putmaxnumanz*. Estimate for the number of non-zeros in A.
- *putmaxnumqnz*. Estimate for the number of non-zeros in the quadratic terms.

None of these functions changes the problem, they only serve as hints. If the problem ends up growing larger, the estimates are automatically increased.

Do not mix `put-` and `get-` functions

MOSEK will queue `put-` requests internally until a `get-` function is called. If `put-` and `get-` calls are interleaved, the queue will have to be flushed more frequently, decreasing efficiency.

In general `get-` commands should not be called often (or at all) during problem setup.

Use the LIFO principle

When removing constraints and variables, try to use a LIFO (Last In First Out) approach. **MOSEK** can more efficiently remove constraints and variables with a high index than a small index.

An alternative to removing a constraint or a variable is to fix it at 0, and set all relevant coefficients to 0. Generally this will not have any impact on the optimization speed.

Add more constraints and variables than you need (now)

The cost of adding one constraint or one variable is about the same as adding many of them. Therefore, it may be worthwhile to add many variables instead of one. Initially fix the unused variable at zero, and then later unfix them as needed. Similarly, you can add multiple free constraints and then use them as needed.

Do not remove basic variables

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.

10.6 The license system

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

The following discussion is in practice only relevant for floating licenses (with a limited number of tokens). For uncounted license types (server, trial, personal academic and group) there is no need to monitor the license usage or check licenses back in since nothing restricts multiple processes from using the license.

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 *optimize* is first called and

- it is returned when the process is terminated or when the **MOSEK** environment is deleted (if using an explicit environment created by the user, see. Sec. 7.1)

Calling *optimize* from different threads using the same **MOSEK** environment only consumes one license token.

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 license wait flag with the parameter *MSK_IPAR_LICENSE_WAIT* will force **MOSEK** to wait until a license token becomes available instead of returning with an error. The wait time between checks can be set with *putlicensewait*.
- Additional license checkouts and checkins can be performed with the functions *checkinlicense* and *checkoutlicense*.
- The default path to the license file can be changed with *putlicensepath*. This must be done before the first optimization, further invocations will have no effect.
- Usually the license system is stopped automatically when the **MOSEK** library is unloaded. However, when the user explicitly unloads the library (using e.g. FreeLibrary), the license system must be stopped before the library is unloaded. This can be done by calling the function *licensecleanup* as the last function call to **MOSEK**.

10.7 Deployment

When redistributing a Julia application using the **MOSEK** Optimizer API for Julia 11.2.7, the following shared libraries from the **MOSEK** `bin` folder are required:

- Linux : `libmosek64`, `libtbb`,
- Windows : `mosek64`, `tbb`,
- OSX : `libmosek64`, `libtbb`.

Chapter 11

Case Studies

In this section we present some case studies in which the Optimizer API for Julia 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, factor model, transaction cost, market impact cost
  - **Type:** Conic Quadratic, Power Cone, Mixed-Integer Optimization
- *Logistic regression*
  - **Keywords:** machine learning, logistic regression, classifier, log-sum-exp, softplus, regularization
  - **Type:** Exponential Cone, Quadratic Cone
- *Concurrent Optimizer*
  - **Keywords:** Concurrent optimization
  - **Type:** Linear Optimization, Mixed-Integer Optimization

11.1 Portfolio Optimization

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

Familiarity with Sec. 6.2 is recommended to follow the syntax used to create affine conic constraints (ACCs) throughout all the models appearing in this case study.

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

11.1.1 The Basic Model

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

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

and covariance

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

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

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

and variance

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

The standard deviation

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

is usually associated with risk.

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

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

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 $G \in \mathbb{R}^{n \times k}$ such that

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

In general the choice of G is **not** unique and one possible choice of G is the Cholesky factorization of Σ. However, in many cases another choice is better for efficiency reasons as discussed in Sec. 11.1.3. For a given G we have that

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

Hence, we may write the risk constraint as

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

or equivalently

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

where $\mathcal{Q}^{k+1}$ is the $(k+1)$-dimensional quadratic cone. Note that specifically when G is derived using Cholesky factorization, $k = n$.

Therefore, problem (11.1) can be written as

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

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

$$\mu = \left[\begin{array}{cccccccc} 0.0720, & 0.1552, & 0.1754, & 0.0898, & 0.4290, & 0.3929, & 0.3217, & 0.1838 \end{array} \right]^T$$

and

$$\Sigma = \left[\begin{array}{cccccccc} 0.0946 & 0.0374 & 0.0349 & 0.0348 & 0.0542 & 0.0368 & 0.0321 & 0.0327 \\ 0.0374 & 0.0775 & 0.0387 & 0.0367 & 0.0382 & 0.0363 & 0.0356 & 0.0342 \\ 0.0349 & 0.0387 & 0.0624 & 0.0336 & 0.0395 & 0.0369 & 0.0338 & 0.0243 \\ 0.0348 & 0.0367 & 0.0336 & 0.0682 & 0.0402 & 0.0335 & 0.0436 & 0.0371 \\ 0.0542 & 0.0382 & 0.0395 & 0.0402 & 0.1724 & 0.0789 & 0.0700 & 0.0501 \\ 0.0368 & 0.0363 & 0.0369 & 0.0335 & 0.0789 & 0.0909 & 0.0536 & 0.0449 \\ 0.0321 & 0.0356 & 0.0338 & 0.0436 & 0.0700 & 0.0536 & 0.0965 & 0.0442 \\ 0.0327 & 0.0342 & 0.0243 & 0.0371 & 0.0501 & 0.0449 & 0.0442 & 0.0816 \end{array} \right].$$

Using Cholesky factorization, this implies

$$G^T = \left[\begin{array}{cccccccc} 0.3076 & 0.1215 & 0.1134 & 0.1133 & 0.1763 & 0.1197 & 0.1044 & 0.1064 \\ 0. & 0.2504 & 0.0995 & 0.0916 & 0.0669 & 0.0871 & 0.0917 & 0.0851 \\ 0. & 0. & 0.1991 & 0.0587 & 0.0645 & 0.0737 & 0.0647 & 0.0191 \\ 0. & 0. & 0. & 0.2088 & 0.0493 & 0.0365 & 0.0938 & 0.0774 \\ 0. & 0. & 0. & 0. & 0.3609 & 0.1257 & 0.1016 & 0.0571 \\ 0. & 0. & 0. & 0. & 0. & 0.2155 & 0.0566 & 0.0619 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0.2251 & 0.0333 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0.2202 \end{array} \right]$$

In Sec. 11.1.3, we present a different way of obtaining G based on a factor model, that leads to more efficient computation.

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 portfolio( mu :: Vector{Float64},
                    x0 :: Vector{Float64},
                    w  :: Float64,
                    gamma :: Float64,
                    GT :: Array{Float64,2})
    (k,n) = size(GT)
    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        # Directs the log task stream
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        totalBudget = sum(x0)+w

        #Offset of variables into the API variable.
        x_ofs = 0

        # Constraints offsets
        budget_ofs = 0

        # Holding variable x of length n
        # No other auxiliary variables are needed in this formulation
        appendvars(task,n)

        # Setting up variable x
        for j in 1:n
            # Optionally we can give the variables names
            putvarname(task, x_ofs+j, "x[$(j)]");
            # No short-selling - x^l = 0, x^u = inf
            putvarbound(task,x_ofs+j, MSK_BK_LO, 0.0, Inf);
        end

        # One linear constraint: total budget
        appendcons(task,1);
```

(continues on next page)

(continued from previous page)

```
        putconname(task,1,"budget");
        for j in 1:n
            # Coefficients in the first row of A
            putaij(task,budget_ofs+1, x_ofs+j, 1.0)
        end

        putconbound(task, budget_ofs+1, MSK_BK_FX, totalBudget, totalBudget)

        # Input (gamma, GTx) in the AFE (affine expression) storage
        # We need k+1 rows
        appendafes(task,k + 1)
        # The first affine expression = gamma
        putafeg(task,1, gamma)
        # The remaining k expressions comprise GT*x, we add them row by row
        # In more realisic scenarios it would be better to extract nonzeros and input␣
↪in sparse form

        subj = [1:n...]
        for i in 1:k
            putafefrow(task,i + 1, subj, GT[i,:])
        end

        # Input the affine conic constraint (gamma, GT*x) \in QCone
        # Add the quadratic domain of dimension k+1
        qdom = appendquadraticconedomain(task,k + 1)
        # Add the constraint
        appendaccseq(task,qdom,1,nothing)
        putaccname(task,1, "risk")

        # Objective: maximize expected return mu^T x
        putclist(task,[x_ofs+1:x_ofs+n...],mu)
        putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)

        optimize(task)

        # Display solution summary for quick inspection of results
        solutionsummary(task,MSK_STREAM_LOG)

        writedata(task,"portfolio_1_basic.ptf");

        # Check if the interior point solution is an optimal point
        if getsolsta(task, MSK_SOL_ITR) != MSK_SOL_STA_OPTIMAL
            # See https://docs.mosek.com/latest/juliaapi/accessing-solution.html␣
↪about handling solution statuses.
            error("Solution not optimal")
        end

        # Read the results
        xx = getxxslice(task,MSK_SOL_ITR, x_ofs+1,x_ofs+n+1)
        expret = mu' * xx

        (xx,expret)
    end
end # portfolio()
```

The code is organized as follows:

- We have n optimization variables, one per each asset in the portfolio. They correspond to the

variable x from (11.1) and their indices as variables in the task are from 0 to $n-1$ (inclusive).

- The linear part of the problem: budget constraint, no-short-selling bounds and the objective are added in the linear data of the task (A matrix, c vector and bounds) following the techniques introduced in the tutorial of Sec. 6.1.
- For the quadratic constraint we follow the path introduced in the tutorial of Sec. 6.2. We add the vector $(\gamma, G^T x)$ to the affine expression storage (AFE), create a quadratic domain of suitable length, and add the affine conic constraint (ACC) with the selected affine expressions. In the segment

```
# Input the affine conic constraint (gamma, GT*x) \in QCone
# Add the quadratic domain of dimension k+1
qdom = appendquadraticconedomain(task,k + 1)
# Add the constraint
appendaccseq(task,qdom,1,nothing)
```

we use *appendaccseq* to append a single ACC with the quadratic domain `qdom` and with a sequence of affine expressions starting at position 0 in the AFE storage and of length equal to the dimension of `qdom`. This is the simplest way to achieve what we need, since previously we also stored the required rows in AFE in the same order.

11.1.2 The Efficient Frontier

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

Given a nonnegative α the problem

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

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

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

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

Example code

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

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

```
function portfolio( mu :: Vector{Float64},
                    x0 :: Vector{Float64},
                    w  :: Float64,
                    alphas :: Vector{Float64},
                    GT :: Array{Float64,2})
```

(continues on next page)

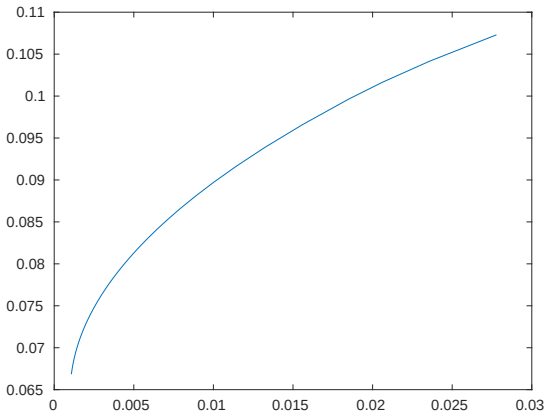


Fig. 11.1: The efficient frontier for the sample data.

(continued from previous page)

```
    (k,n) = size(GT)
    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        # Directs the log task stream
        #putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        totalBudget = sum(x0)+w

        #Offset of variables into the API variable.
        x_ofs = 0
        s_ofs = n

        # Constraints offsets
        budget_ofs = 0

        # Holding variable x of length n
        # No other auxiliary variables are needed in this formulation
        appendvars(task,n+1)

        # Setting up variable x
        for j in 1:n
            # Optionally we can give the variables names
            putvarname(task, x_ofs+j, "x[$(j)]")
            # No short-selling - x^l = 0, x^u = inf
            putvarbound(task,x_ofs+j, MSK_BK_LO, 0.0, Inf)
        end
        putvarname(task, s_ofs+1, "s")
        putvarbound(task,s_ofs+1, MSK_BK_FR, -Inf, Inf)

        # One linear constraint: total budget
        appendcons(task,1)
        putconname(task,1,"budget")
        for j in 1:n
            # Coefficients in the first row of A
            putaij(task,budget_ofs+1, x_ofs+j, 1.0)
        end

        putconbound(task, budget_ofs+1, MSK_BK_FX, totalBudget, totalBudget)

        # Input (gamma, GTx) in the AFE (affine expression) storage
        # We build the following F and g for variables [x, s]:
        #     [0, 1]      [0  ]
        # F = [0, 0], g = [0.5]
        #     [GT,0]      [0  ]
        # We need k+2 rows
        appendafes(task,k + 2)
        # The first affine expression = alpha
        putafefentry(task,1,s_ofs+1,1.0)
        putafeg(task,2,0.5)
        # The remaining k expressions comprise GT*x, we add them row by row
        # In more realisic scenarios it would be better to extract nonzeros and input␣
↪in sparse form
        subj = [1:n...]
        for i in 1:k
            putafefrow(task,i + 2, subj, GT[i,:])
        end
```

(continues on next page)

(continued from previous page)

```
        # Input the affine conic constraint (alpha, GT*x) \in QCone
        # Add the quadratic domain of dimension k+1
        qdom = appendrquadraticconedomain(task,k + 2)
        # Add the constraint
        appendaccseq(task,qdom,1,nothing)
        putaccname(task,1, "risk")

        # Objective: maximize expected return mu^T x
        putclist(task,[x_ofs+1:x_ofs+n...],mu)
        putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)

        expret = Float64[]
        stddev = Float64[]
        for alpha in alphas
            putcj(task,s_ofs+1,-alpha)
            optimize(task)
            writedata(task,"portfolio_2_frontier-$alpha.ptf")

            # Check if the interior point solution is an optimal point
            if getsolsta(task, MSK_SOL_ITR) != MSK_SOL_STA_OPTIMAL
                # See https://docs.mosek.com/latest/juliaapi/accessing-solution.html␣
↪about handling solution statuses.
                error("Solution not optimal")
            end

            # Display solution summary for quick inspection of results
            solutionsummary(task,MSK_STREAM_LOG)

            # Read the results
            r = mu' * getxxslice(task,MSK_SOL_ITR, x_ofs+1,x_ofs+n+1)
            slvl =    getxxslice(task,MSK_SOL_ITR, s_ofs+1,s_ofs+2)

            push!(expret,r)
            push!(stddev,sqrt(slvl[1]))
        end
        (expret,stddev)
    end
end # portfolio()
```

Note that we changed the coefficient α of the variable s in a loop. This way we were able to reuse the same model for all solves along the efficient frontier, simply changing the value of α between the solves.

11.1.3 Factor model and efficiency

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

The computational cost is of course to some extent dependent on the number of constraints and variables in the optimization problem. However, in practice a more important factor is the sparsity: the number of nonzeros used to represent the problem. Indeed it is often better to focus on the number of nonzeros in G see (11.2) and try to reduce that number by for instance changing the choice of G.

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

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

where D is a positive definite diagonal matrix. Moreover, V is a matrix with n rows and k columns. Such a model for the covariance matrix is called a factor model and usually k is much smaller than n.

In practice k 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

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

because then

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

This choice requires storage proportional to $n + kn$ which is much less than for the Cholesky choice of G. Indeed assuming k 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].

Factor model in finance

Factor model structure is typical in financial context. It is common to model security returns as the sum of two components using a factor model. The first component is the linear combination of a small number of factors common among a group of securities. The second component is a residual, specific to each security. It can be written as $R = \sum_j \beta_j F_j + \theta$, where R is a random variable representing the return of a security at a particular point in time, F_j is the random variable representing the common factor j, β_j is the exposure of the return to factor j, and θ is the specific component.

Such a model will result in the covariance structure

$$\Sigma = \Sigma_\theta + \beta \Sigma_F \beta^T,$$

where Σ_F is the covariance of the factors and Σ_θ is the residual covariance. This structure is of the form discussed earlier with $D = \Sigma_\theta$ and $V = \beta P$, assuming the decomposition $\Sigma_F = PP^T$. If the number of factors k is low and Σ_θ is diagonal, we get a very sparse G that provides the storage and solution time benefits.

Example code

Here we will work with the example data of a two-factor model ($k = 2$) built using the variables

$$\beta = \begin{bmatrix} 0.4256 & 0.1869 \\ 0.2413 & 0.3877 \\ 0.2235 & 0.3697 \\ 0.1503 & 0.4612 \\ 1.5325 & -0.2633 \\ 1.2741 & -0.2613 \\ 0.6939 & 0.2372 \\ 0.5425 & 0.2116 \end{bmatrix},$$

$$\theta = [0.0720, 0.0508, 0.0377, 0.0394, 0.0663, 0.0224, 0.0417, 0.0459],$$

and the factor covariance matrix is

$$\Sigma_F = \begin{bmatrix} 0.0620 & 0.0577 \\ 0.0577 & 0.0908 \end{bmatrix},$$

giving

$$P = \begin{bmatrix} 0.2491 & 0. \\ 0.2316 & 0.1928 \end{bmatrix}.$$

Then the matrix G would look like

$$G = \begin{bmatrix} \beta P & \Sigma_\theta^{1/2} \end{bmatrix} = \begin{bmatrix} 0.1493 & 0.0360 & 0.2683 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0.1499 & 0.0747 & 0. & 0.2254 & 0. & 0. & 0. & 0. & 0. & 0. \\ 0.1413 & 0.0713 & 0. & 0. & 0.1942 & 0. & 0. & 0. & 0. & 0. \\ 0.1442 & 0.0889 & 0. & 0. & 0. & 0.1985 & 0. & 0. & 0. & 0. \\ 0.3207 & -0.0508 & 0. & 0. & 0. & 0. & 0.2576 & 0. & 0. & 0. \\ 0.2568 & -0.0504 & 0. & 0. & 0. & 0. & 0. & 0.1497 & 0. & 0. \\ 0.2277 & 0.0457 & 0. & 0. & 0. & 0. & 0. & 0. & 0.2042 & 0. \\ 0.1841 & 0.0408 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0.2142 \end{bmatrix}.$$

This matrix is indeed very sparse.

In general, we get an $n \times (n+k)$ size matrix this way with k full columns and an $n \times n$ diagonal part. In order to maintain a sparse representation we do not construct the matrix G explicitly in the code but instead work with two pieces of data: the dense matrix $G_{\text{factor}} = \beta P$ of shape $n \times k$ and the diagonal vector θ of length n.

Example code

In the following we demonstrate how to write code to compute the matrix G_{factor} of the factor model. We start with the inputs

Listing 11.3: Inputs for the computation of the matrix G_{factor} from the factor model.

```
# Factor exposure matrix, n x 2
B = [ 0.4256  0.1869
      0.2413  0.3877
      0.2235  0.3697
      0.1503  0.4612
      1.5325 -0.2633
      1.2741 -0.2613
      0.6939  0.2372
      0.5425  0.2116 ],
# Factor covariance matrix, 2x2
S_F = [ 0.0620 0.0577
        0.0577 0.0908 ],
# Specific risk components
theta = [0.0720, 0.0508, 0.0377, 0.0394, 0.0663, 0.0224, 0.0417, 0.0459],
S_sqrt_theta = Matrix(Diagonal(sqrt.(theta))),
```

Then the matrix G_{factor} is obtained as:

```
P = Matrix(cholesky(S_F).L),
BP = B * P,
G = [ BP S_sqrt_theta ],
GT = Matrix(G') # 10 x 8
```

The code for computing an optimal portfolio in the factor model is very similar to the one from the basic model in Listing 11.1 with one notable exception: we construct the expression $G^T x$ appearing in the conic constraint by stacking together two separate vectors $G_{\text{factor}}^T x$ and $\Sigma_\theta^{1/2} x$:

```
        # Input (gamma, G_factor_T x, diag(sqrt(theta))*x) in the AFE (affine␣
↪expression) storage
        # We need k+n+1 rows and we fill them in in three parts
        appendafes(task,n+k+1)
        # 1. The first affine expression = gamma, will be specified later
        # 2. The next k expressions comprise G_factor_T*x, we add them row by row
        #    transposing the matrix G_factor on the fly
```

(continues on next page)

(continued from previous page)

```
        subj = [1:n...]
        for i in 1:k
            putafefrow(task,i + 1, subj, GT[i,:])
        end
        # 3. The remaining n rows contain sqrt(theta) on the diagonal

        for i in 1:n
            putafefentry(task,k + 1 + i, subj[i], sqrt(theta[i]))
        end
```

The full code is demonstrated below:

Listing 11.4: Implementation of portfolio optimization in the factor model.

```
function portfolio( mu :: Vector{Float64},
                    x0 :: Vector{Float64},
                    w  :: Float64,
                    gammas :: Vector{Float64},
                    theta :: Vector{Float64},
                    GT :: Array{Float64,2})
    (k,n) = size(GT)
    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        # Directs the log task stream
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        totalBudget = sum(x0)+w

        #Offset of variables into the API variable.
        x_ofs = 0

        # Constraints offsets
        budget_ofs = 0

        # Holding variable x of length n
        # No other auxiliary variables are needed in this formulation
        appendvars(task,n)

        # Setting up variable x
        for j in 1:n
            # Optionally we can give the variables names
            putvarname(task, x_ofs+j, "x[$(j)]");
            # No short-selling - x^l = 0, x^u = inf
            putvarbound(task,x_ofs+j, MSK_BK_LO, 0.0, Inf);
        end

        # One linear constraint: total budget
        appendcons(task,1);
        putconname(task,1,"budget");
        for j in 1:n
            # Coefficients in the first row of A
            putaij(task,budget_ofs+1, x_ofs+j, 1.0)
        end

        putconbound(task, budget_ofs+1, MSK_BK_FX, totalBudget, totalBudget)
        # Input (gamma, G_factor_T x, diag(sqrt(theta))*x) in the AFE (affine␣
```

(continues on next page)

(continued from previous page)

```
↪expression) storage
        # We need k+n+1 rows and we fill them in in three parts
        appendafes(task,n+k+1)
        # 1. The first affine expression = gamma, will be specified later
        # 2. The next k expressions comprise G_factor_T*x, we add them row by row
        #    transposing the matrix G_factor on the fly

        subj = [1:n...]
        for i in 1:k
            putafefrow(task,i + 1, subj, GT[i,:])
        end
        # 3. The remaining n rows contain sqrt(theta) on the diagonal

        for i in 1:n
            putafefentry(task,k + 1 + i, subj[i], sqrt(theta[i]))
        end

        # Input the affine conic constraint (gamma, GT*x) \in QCone
        # Add the quadratic domain of dimension k+1
        qdom = appendquadraticconedomain(task,n + k + 1)
        # Add the constraint
        appendaccseq(task,qdom,1,nothing)
        putaccname(task,1, "risk")

        # Objective: maximize expected return mu^T x
        putclist(task,[x_ofs+1:x_ofs+n...],mu)
        putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)

        res = Tuple{Float64,Float64}[]
        for gamma in gammas
            putafeg(task,1, gamma)

            optimize(task)

            if getsolsta(task, MSK_SOL_ITR) != MSK_SOL_STA_OPTIMAL
                # See https://docs.mosek.com/latest/juliaapi/accessing-solution.html␣
↪about handling solution statuses.
                error("Solution not optimal")
            end

            writedata(task,"portfolio_6_factor-$(gamma).ptf");

            # Read the results
            xx = getxxslice(task,MSK_SOL_ITR, x_ofs+1,x_ofs+n+1)
            expret = mu' * xx

            push!(res,(gamma,expret))
        end

        res
    end
end # portfolio()
```

11.1.4 Slippage Cost

The basic Markowitz model assumes that there are no costs associated with trading the assets and that the returns of the assets are independent of the amount traded. Neither of those assumptions is usually valid in practice. Therefore, a more realistic model is

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

Here Δx_j is the change in the holding of asset j i.e.

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

and $T_j(\Delta x_j)$ specifies the transaction costs when the holding of asset j is changed from its initial value. In the next two sections we show two different variants of this problem with two nonlinear cost functions T.

11.1.5 Market Impact Costs

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

$$T_j(\Delta x_j) = m_j |\Delta x_j|^{3/2}$$

where m_j is a constant that is estimated in some way by the trader. See [GK00] [p. 452] for details. From the Modeling Cookbook we know that $t \geq |z|^{3/2}$ can be modeled directly using the power cone $\mathcal{P}_3^{2/3,1/3}$:

$$\{(t,z) : t \geq |z|^{3/2}\} = \{(t,z) : (t,1,z) \in \mathcal{P}_3^{2/3,1/3}\}$$

Hence, it follows that $\sum_{j=1}^n T_j(\Delta x_j) = \sum_{j=1}^n m_j |x_j - x_j^0|^{3/2}$ can be modeled by $\sum_{j=1}^n m_j t_j$ under the constraints

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

Unfortunately this set of constraints is nonconvex due to the constraint

$$z_j = |x_j - x_j^0| \tag{11.7}$$

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

$$z_j \geq |x_j - x_j^0|, \tag{11.8}$$

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

$$z_j > |x_j - x_j^0| \tag{11.9}$$

cannot hold for an optimal solution.

If the optimal solution has the property (11.9) then the market impact cost within the model is larger than the true market impact cost and hence money are essentially considered garbage and removed by generating transaction costs. This may happen if a portfolio with very small risk is requested because the only way to obtain a small risk is to get rid of some of the assets by generating transaction costs. We generally assume that this is not the case and hence the models (11.7) and (11.8) are equivalent.

The above observations lead to

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

The revised budget constraint

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

specifies that the initial wealth covers the investment and the transaction costs. It should be mentioned that transaction costs of the form

$$t_j \geq |z_j|^p$$

where $p > 1$ is a real number can be modeled with the power cone as

$$(t_j, 1, z_j) \in \mathcal{P}_3^{1/p, 1-1/p}.$$

See the Modeling Cookbook for details.

Example code

Listing 11.5 demonstrates how to compute an optimal portfolio when market impact cost are included.

Listing 11.5: Implementation of model (11.10).

```
function portfolio( mu :: Vector{Float64},
                    x0 :: Vector{Float64},
                    w  :: Float64,
                    gamma :: Float64,
                    GT :: Array{Float64,2},
                    m  :: Vector{Float64})
    (k,n) = size(GT)
    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        # Directs the log task stream
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        totalBudget = sum(x0)+w

        #Offset of variables into the API variable.
        x_ofs = 0
        c_ofs = n
        z_ofs = 2*n
        # Constraints offsets
        budget_ofs = 0

        # Holding variable x of length n
        # No other auxiliary variables are needed in this formulation
        appendvars(task,3*n)

        # Setting up variable x
        for j in 1:n
            # Optionally we can give the variables names
            putvarname(task, x_ofs+j, "x[$(j)]");
            putvarname(task, c_ofs+j, "c[$(j)]");
```

(continues on next page)

(continued from previous page)

```
            putvarname(task, z_ofs+j, "z[$(j)]");
        end
        # No short-selling - x^l = 0, x^u = inf
        putvarboundsliceconst(task,x_ofs+1,x_ofs+n+1, MSK_BK_LO,  0.0, Inf);
        putvarboundsliceconst(task,c_ofs+1,c_ofs+n+1, MSK_BK_FR, -Inf, Inf);
        putvarboundsliceconst(task,z_ofs+1,z_ofs+n+1, MSK_BK_FR, -Inf, Inf);

        # Linear constraint: total budget
        let budget_ofs = getnumcon(task)
            appendcons(task,1);
            putconname(task,1,"budget");
            for j in 1:n
                # Coefficients in the first row of A
                putaij(task,budget_ofs+1, x_ofs+j, 1.0)
                putaij(task,budget_ofs+1, c_ofs+j, m[j])
            end
        end

        putconbound(task, budget_ofs+1, MSK_BK_FX, totalBudget, totalBudget)

        # - Absolute value
        let zabs1_ofs = getnumcon(task),
            zabs2_ofs = zabs1_ofs+n

            appendcons(task,2*n)
            for i in 1:n
                putconname(task,zabs1_ofs + i, "zabs1[$i]")
                putaij(task,zabs1_ofs + i, x_ofs + i, -1.0)
                putaij(task,zabs1_ofs + i, z_ofs + i, 1.0)
                putconbound(task,zabs1_ofs + i, MSK_BK_LO, -x0[i], Inf)
                putconname(task,zabs2_ofs + i, "zabs2[$i]")
                putaij(task,zabs2_ofs + i, x_ofs + i, 1.0)
                putaij(task,zabs2_ofs + i, z_ofs + i, 1.0)
                putconbound(task,zabs2_ofs + i, MSK_BK_LO, x0[i], Inf)
            end
        end

        let qdom = appendquadraticconedomain(task,k + 1)
            afei = getnumafe(task)
            acci = getnumacc(task)
            # Input (gamma, GTx) in the AFE (affine expression) storage
            # We need k+1 rows
            appendafes(task,k + 1)
            # The first affine expression = gamma
            putafeg(task,afei+1, gamma)
            # The remaining k expressions comprise GT*x, we add them row by row
            # In more realisic scenarios it would be better to extract nonzeros and␣
↪input in sparse form

            subj = [1:n...]
            for i in 1:k
                putafefrow(task,afei+i+1, subj, GT[i,:])
            end

            # Input the affine conic constraint (gamma, GT*x) \in QCone
            # Add the quadratic domain of dimension k+1
```

(continues on next page)

(continued from previous page)

```
            # Add the constraint
            appendaccseq(task,qdom,1,nothing)
            putaccname(task,acci+1, "risk")
        end

        let dom = appendprimalpowerconedomain(task,3,[2.0,1.0])
            afei = getnumafe(task)
            acci = getnumacc(task)
            afe0 = afei

            appendafes(task,2*n+1)
            putafeg(task,afe0+1,1.0)

            afei += 1

            for i in 1:n
                putafefentry(task,afei + i,     c_ofs + i, 1.0);
                putafefentry(task,afei + n + i, z_ofs + i, 1.0);
            end

            accafes = Int64[ k for i in 1:n for k in [ afei + i, afe0+1, afei + n + i␣
↪] ]
            accb    = zeros(n*3)
            accdom  = Int64[ dom for i in 1:n ]

            appendaccs(task,accdom,accafes,accb)

            for i in 1:n
                putaccname(task,acci+i,"market_impact[$i]")
            end
        end

        # Objective: maximize expected return mu^T x
        putclist(task,[x_ofs+1:x_ofs+n...],mu)
        putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)

        optimize(task)

        # Check if the interior point solution is an optimal point
        if getsolsta(task, MSK_SOL_ITR) != MSK_SOL_STA_OPTIMAL
            # See https://docs.mosek.com/latest/juliaapi/accessing-solution.html␣
↪about handling solution statuses.
            error("Solution not optimal")
        end

        # Display solution summary for quick inspection of results
        solutionsummary(task,MSK_STREAM_LOG)

        writedata(task,"portfolio_3_impact.ptf");

        # Read the results
        xx = getxxslice(task,MSK_SOL_ITR, x_ofs+1,x_ofs+n+1)
        expret = mu' * xx

        (xx,expret)
```

(continues on next page)

(continued from previous page)

```
    end
end # portfolio()
```

Note that in the following part of the code:

```
        let dom = appendprimalpowerconedomain(task,3,[2.0,1.0])
            afei = getnumafe(task)
            acci = getnumacc(task)
            afe0 = afei

            appendafes(task,2*n+1)
            putafeg(task,afe0+1,1.0)

            afei += 1

            for i in 1:n
                putafefentry(task,afei + i,     c_ofs + i, 1.0);
                putafefentry(task,afei + n + i, z_ofs + i, 1.0);
            end

            accafes = Int64[ k for i in 1:n for k in [ afei + i, afe0+1, afei + n + i
↪] ]
            accb    = zeros(n*3)
            accdom  = Int64[ dom for i in 1:n ]

            appendaccs(task,accdom,accafes,accb)

            for i in 1:n
                putaccname(task,acci+i,"market_impact[$i]")
            end
        end
```

we create a sequence of power cones of the form $(t_k, 1, x_k - x_k^0) \in \mathcal{P}_3^{2/3,1/3}$. The power cones are determined by the sequence of exponents $(2, 1)$; we create a single domain to account for that.

Moreover, note that the second coordinate of all these affine conic constraints is the same affine expression equal to 1, and we use the feature that allows us to define this affine expression only once (as AFE number `aoff_pow + 2 * n`) and reuse it in all the ACCs.

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

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

First observe that

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

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

$$f_j y_j + g_j z_j.$$

Example code

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

Listing 11.6: Code solving problem (11.11).

```
function portfolio( mu :: Vector{Float64},
                    x0 :: Vector{Float64},
                    w  :: Float64,
                    gamma :: Float64,
                    GT :: Array{Float64,2},
                    f  :: Vector{Float64},
                    g  :: Vector{Float64})
    (k,n) = size(GT)
    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        # Directs the log task stream
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        totalBudget = sum(x0)+w

        #Offset of variables into the API variable.
        x_ofs = 0
        y_ofs = n
        z_ofs = 2*n

        # Constraints offsets

        # Holding variable x of length n
        # No other auxiliary variables are needed in this formulation
        appendvars(task,3*n)

        # Setting up variable x
        for j in 1:n
            # Optionally we can give the variables names
            putvarname(task, x_ofs+j, "x[$(j)]");
            putvarname(task, z_ofs+j, "z[$(j)]");
            putvarname(task, y_ofs+j, "y[$(j)]");
            # No short-selling - x^l = 0, x^u = inf
            putvartype(task, y_ofs+j, MSK_VAR_TYPE_INT);
        end
        putvarboundsliceconst(task,x_ofs+1,x_ofs+n+1, MSK_BK_LO, 0.0, Inf);
        putvarboundsliceconst(task,y_ofs+1,y_ofs+n+1, MSK_BK_RA, 0.0, 1.0);
        putvarboundsliceconst(task,z_ofs+1,z_ofs+n+1, MSK_BK_FR, -Inf, Inf);

        # One linear constraint: total budget
        let coni = getnumcon(task)
            appendcons(task,1);
```

(continues on next page)

(continued from previous page)

```
            putconname(task,coni+1,"budget")

            consub = Int32[ coni+1 for i in 1:n ]
            putaijlist(task,consub,[x_ofs+1:x_ofs+n...], ones(n))
            putaijlist(task,consub,[z_ofs+1:z_ofs+n...],g)
            putaijlist(task,consub,[y_ofs+1:y_ofs+n...],f)

            putconbound(task, coni+1, MSK_BK_FX, totalBudget, totalBudget)
        end

        let coni = getnumcon(task)
            appendcons(task,2*n)
            for i in 1:n
                putconname(task,coni+i,"zabs1[$i]")
                putconname(task,coni+n+i,"zabs2[$i]")

                putaij(task,coni + i, x_ofs + i, -1.0)
                putaij(task,coni + i, z_ofs + i, 1.0)
                putconbound(task,coni + i, MSK_BK_LO, -x0[i], Inf)

                putaij(task,coni + n + i, x_ofs + i, 1.0)
                putaij(task,coni + n + i, z_ofs + i, 1.0)
                putconbound(task,coni + n + i, MSK_BK_LO, x0[i], Inf)
            end
        end

        # - Switch
        let coni = getnumcon(task)
            appendcons(task,n)
            for i in 1:n
                putconname(task,coni+i, "switch[$i]")
                putaij(task,coni + i, z_ofs + i, 1.0)
                putaij(task,coni + i, y_ofs + i, -totalBudget)
                putconbound(task,coni + i, MSK_BK_UP, -Inf, 0.0)
            end
        end


        let afei = getnumafe(task),
            acci = getnumacc(task)
            # Input (gamma, GTx) in the AFE (affine expression) storage
            # We need k+1 rows
            appendafes(task,k + 1)
            # The first affine expression = gamma
            putafeg(task,afei+1, gamma)
            # The remaining k expressions comprise GT*x, we add them row by row
            # In more realisic scenarios it would be better to extract nonzeros and␣
↪input in sparse form


            subj = [1:n...]
            for i in 1:k
                putafefrow(task,afei+i+1, subj, GT[i,:])
            end

            # Input the affine conic constraint (gamma, GT*x) \in QCone
```

(continues on next page)

(continued from previous page)

```
            # Add the quadratic domain of dimension k+1
            qdom = appendquadraticconedomain(task,k + 1)
            # Add the constraint
            appendaccseq(task,qdom,1,nothing)
            putaccname(task,acci+1, "risk")
        end


        # Objective: maximize expected return mu^T x
        putclist(task,[x_ofs+1:x_ofs+n...],mu)
        putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)

        optimize(task)

        # Check if the integer solution is an optimal point
        if getsolsta(task, MSK_SOL_ITG) != MSK_SOL_STA_INTEGER_OPTIMAL
            # See https://docs.mosek.com/latest/juliaapi/accessing-solution.html␣
↪about handling solution statuses.
            error("Solution not optimal")
        end

        writedata(task,"portfolio_4_transcost.ptf")

        # Display solution summary for quick inspection of results
        solutionsummary(task,MSK_STREAM_LOG)

        writedata(task,"portfolio_4_transcost.ptf");

        # Read the results
        xx = getxxslice(task,MSK_SOL_ITG, x_ofs+1,x_ofs+n+1)
        expret = mu' * xx

        (xx,expret)
    end
end # portfolio()
```

11.1.7 Cardinality constraints

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

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

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

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

Example code

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

Listing 11.7: Code solving problem (11.12).

```
function portfolio( mu :: Vector{Float64},
                    x0 :: Vector{Float64},
                    w  :: Float64,
                    gamma :: Float64,
                    GT :: Array{Float64,2},
                    K  :: Int)
    (k,n) = size(GT)
    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        # Directs the log task stream
        # putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

        totalBudget = sum(x0)+w

        #Offset of variables into the API variable.
        x_ofs = 0
        y_ofs = n
        z_ofs = 2*n

        # Constraints offsets

        # Holding variable x of length n
        # No other auxiliary variables are needed in this formulation
        appendvars(task,3*n)

        # Setting up variable x
        for j in 1:n
            # Optionally we can give the variables names
            putvarname(task, x_ofs+j, "x[$(j)]");
            putvarname(task, y_ofs+j, "y[$(j)]");
            putvarname(task, z_ofs+j, "z[$(j)]");
            putvartype(task, y_ofs+j, MSK_VAR_TYPE_INT);
        end
        # No short-selling - x^l = 0, x^u = inf
        putvarboundsliceconst(task,x_ofs+1,x_ofs+n+1, MSK_BK_LO, 0.0, Inf);
        putvarboundsliceconst(task,y_ofs+1,y_ofs+n+1, MSK_BK_RA, 0.0, 1.0);
        putvarboundsliceconst(task,z_ofs+1,z_ofs+n+1, MSK_BK_FR, -Inf, Inf);

        # One linear constraint: total budget
        let coni = getnumcon(task)
            appendcons(task,1);
            putconname(task,coni+1,"budget")

            consub = Int32[ coni+1 for i in 1:n ]
            putaijlist(task,consub,[x_ofs+1:x_ofs+n...], ones(n))
            putconbound(task, coni+1, MSK_BK_FX, totalBudget, totalBudget)
        end

        let coni = getnumcon(task)
            appendcons(task,1)
            putarow(task,coni+1,[y_ofs+1:y_ofs+1+n...],ones(n));
            putconbound(task,coni+1,MSK_BK_UP, 0.0, K)
```

(continues on next page)

(continued from previous page)

```
            putconname(task,coni+1,"cardinality")
        end

        let coni = getnumcon(task)
            appendcons(task,2*n)
            for i in 1:n
                putconname(task,coni+i,"zabs1[$i]")
                putconname(task,coni+n+i,"zabs2[$i]")

                putaij(task,coni + i, x_ofs + i, -1.0)
                putaij(task,coni + i, z_ofs + i, 1.0)
                putconbound(task,coni + i, MSK_BK_LO, -x0[i], Inf)

                putaij(task,coni + n + i, x_ofs + i, 1.0)
                putaij(task,coni + n + i, z_ofs + i, 1.0)
                putconbound(task,coni + n + i, MSK_BK_LO, x0[i], Inf)
            end
        end

        # - Switch
        let coni = getnumcon(task)
            appendcons(task,n)
            for i in 1:n
                putconname(task,coni+i, "switch[$i]")
                putaij(task,coni + i, z_ofs + i, 1.0)
                putaij(task,coni + i, y_ofs + i, -totalBudget)
                putconbound(task,coni + i, MSK_BK_UP, -Inf, 0.0)
            end
        end

        let afei = getnumafe(task),
            acci = getnumacc(task)
            # Input (gamma, GTx) in the AFE (affine expression) storage
            # We need k+1 rows
            appendafes(task,k + 1)
            # The first affine expression = gamma
            putafeg(task,afei+1, gamma)
            # The remaining k expressions comprise GT*x, we add them row by row
            # In more realisic scenarios it would be better to extract nonzeros and␣
↪input in sparse form


            subj = [1:n...]
            for i in 1:k
                putafefrow(task,afei+i+1, subj, GT[i,:])
            end

            # Input the affine conic constraint (gamma, GT*x) \in QCone
            # Add the quadratic domain of dimension k+1
            qdom = appendquadraticconedomain(task,k + 1)
            # Add the constraint
            appendaccseq(task,qdom,1,nothing)
            putaccname(task,acci+1, "risk")
        end
```

(continues on next page)

(continued from previous page)

```
        # Objective: maximize expected return mu^T x
        putclist(task,[x_ofs+1:x_ofs+n...],mu)
        putobjsense(task,MSK_OBJECTIVE_SENSE_MAXIMIZE)

        optimize(task)

        if getsolsta(task, MSK_SOL_ITG) != MSK_SOL_STA_INTEGER_OPTIMAL
            # See https://docs.mosek.com/latest/juliaapi/accessing-solution.html␣
↪about handling solution statuses.
            error("Solution not optimal")
        end

        # Display solution summary for quick inspection of results
        solutionsummary(task,MSK_STREAM_LOG)

        writedata(task,"portfolio_5_card-$K.ptf");

        # Read the results
        xx = getxxslice(task,MSK_SOL_ITG, x_ofs+1,x_ofs+n+1)
        expret = mu' * xx

        (xx,expret)
    end
end # portfolio()
```

If we solve our running example with $K = 1, \ldots, n$ then we get the following solutions, with increasing expected returns:

```
Bound 1   Solution: 0.0000e+00   0.0000e+00   1.0000e+00   0.0000e+00   0.0000e+00  ␣
↪0.0000e+00   0.0000e+00   0.0000e+00
Bound 2   Solution: 0.0000e+00   0.0000e+00   3.5691e-01   0.0000e+00   0.0000e+00  ␣
↪6.4309e-01   -0.0000e+00  0.0000e+00
Bound 3   Solution: 0.0000e+00   0.0000e+00   1.9258e-01   0.0000e+00   0.0000e+00  ␣
↪5.4592e-01   2.6150e-01   0.0000e+00
Bound 4   Solution: 0.0000e+00   0.0000e+00   2.0391e-01   0.0000e+00   6.7098e-02  ␣
↪4.9181e-01   2.3718e-01   0.0000e+00
Bound 5   Solution: 0.0000e+00   3.1970e-02   1.7028e-01   0.0000e+00   7.0741e-02  ␣
↪4.9551e-01   2.3150e-01   0.0000e+00
Bound 6   Solution: 0.0000e+00   3.1970e-02   1.7028e-01   0.0000e+00   7.0740e-02  ␣
↪4.9551e-01   2.3150e-01   0.0000e+00
Bound 7   Solution: 0.0000e+00   3.1970e-02   1.7028e-01   0.0000e+00   7.0740e-02  ␣
↪4.9551e-01   2.3150e-01   0.0000e+00
Bound 8   Solution: 1.9557e-10   2.6992e-02   1.6706e-01   2.9676e-10   7.1245e-02  ␣
↪4.9559e-01   2.2943e-01   9.6905e-03
```

11.2 Logistic regression

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

Formulation as an optimization problem

Define the sigmoid function

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

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

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

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

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

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

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

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

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

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

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

This can equivalently be phrased as

$$\begin{array}{lrcll} \text{minimize} & \sum_i t_i + \lambda r & & & \\ \text{subject to} & t_i & \geq -\log(h_\theta(x)) & = \log(1+\exp(-\theta^T x_i)) & \text{if } y_i = 1, \\ & t_i & \geq -\log(1-h_\theta(x)) & = \log(1+\exp(\theta^T x_i)) & \text{if } y_i = 0, \\ & r & \geq \|\theta\|_2. & & \end{array} \tag{11.13}$$

Implementation

As can be seen from (11.13) the key point is to implement the softplus bound $t \geq \log(1+e^u)$, which is the simplest example of a log-sum-exp constraint for two terms. Here t is a scalar variable and u will be the affine expression of the form $\pm\theta^T x_i$. This is equivalent to

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

and further to

$$\begin{array}{rcll} (z_1, 1, u-t) & \in & K_{\exp} & (z_1 \geq \exp(u-t)), \\ (z_2, 1, -t) & \in & K_{\exp} & (z_2 \geq \exp(-t)), \\ z_1 + z_2 & \leq & 1. & \end{array} \tag{11.14}$$

This formulation can be entered using affine conic constraints (see Sec. 6.2).

Listing 11.8: Implementation of $t \geq \log(1 + e^u)$ as in (11.14).

```
"""
Adds ACCs for t_i >= log ( 1 + exp((1-2*y[i]) * theta' * X[i]) )
Adds auxiliary variables, AFE rows and constraints
"""
function softplus(task :: Mosek.Task, d::Int, n::Int, theta::Int, t::Int, X::Matrix
↪{Float64}, y::Vector{Bool})
    nvar = getnumvar(task)
    ncon = getnumcon(task)
    nafe = getnumafe(task)
    z1       = nvar
    z2       = z1+n
    zcon     = ncon
    thetaafe = nafe
    tafe     = thetaafe+n
    z1afe    = tafe+n
    z2afe    = z1afe+n

    appendvars(task,2*n)    # z1, z2
    appendcons(task,n)      # z1 + z2 = 1
    appendafes(task,4*n)    #theta * X[i] - t[i], -t[i], z1[i], z2[i]

    # Linear constraints
    let subi = Vector{Int32}(undef,2*n),
        subj = Vector{Int32}(undef,2*n),
        aval = Vector{Float64}(undef,2*n),
        k = 1

        for i in 1:n
            # z1 + z2 = 1
            subi[k] = zcon+i;  subj[k] = z1+i;  aval[k] = 1;  k += 1
            subi[k] = zcon+i;  subj[k] = z2+i;  aval[k] = 1;  k += 1
        end

        putaijlist(task,subi, subj, aval)
    end
    putconboundsliceconst(task,zcon+1, zcon+n+1, MSK_BK_FX, 1, 1)
    putvarboundsliceconst(task,nvar+1, nvar+2*n+1, MSK_BK_FR, -Inf, Inf)

    # Affine conic expressions
    let afeidx = Vector{Int64}(undef,d*n+4*n),
        varidx = Vector{Int32}(undef,d*n+4*n),
        fval   = Vector{Float64}(undef,d*n+4*n),
        k = 1
        # Thetas
        for i in 1:n
            for j in 1:d
                afeidx[k] = thetaafe + i; varidx[k] = theta + j
                fval[k]   = if y[i] X[i,j]*(-1.0) else X[i,j] end
                k += 1
            end
        end

        # -t[i]
        for i in 1:n
            afeidx[k] = thetaafe + i; varidx[k] = t + i; fval[k] = -1; k += 1
```

(continues on next page)

(continued from previous page)

```
            afeidx[k] = tafe + i;     varidx[k] = t + i; fval[k] = -1; k += 1
        end

        # z1, z2
        for i in 1:n
            afeidx[k] = z1afe + i; varidx[k] = z1 + i; fval[k] = 1; k += 1
            afeidx[k] = z2afe + i; varidx[k] = z2 + i; fval[k] = 1; k += 1
        end

        # Add the expressions
        putafefentrylist(task,afeidx, varidx, fval)
    end

    # Add a single row with the constant expression "1.0"

    let oneafe = getnumafe(task)+1,
        # Add an exponential cone domain
        expdomain = appendprimalexpconedomain(task)

        appendafes(task,1)
        putafeg(task,oneafe,1.0)

        # Conic constraints
        for i in 1:n
            appendacc(task,expdomain, [z1afe+i, oneafe, thetaafe+i], nothing)
            appendacc(task,expdomain, [z2afe+i, oneafe, tafe+i],     nothing)
        end
    end
end # softplus
```

Once we have this subroutine, it is easy to implement a function that builds the regularized loss function model (11.13).

Listing 11.9: Implementation of (11.13).

```
"""
Model logistic regression (regularized with full 2-norm of theta)
X - n x d matrix of data points
y - length n vector classifying training points
lamb - regularization parameter
"""
function logisticRegression(X    :: Matrix{Float64},
                            y    :: Vector{Bool},
                            lamb :: Float64)
    (n,d) = size(X)

    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        # Variables [r; theta; t]
        nvar = 1+d+n
        appendvars(task,nvar)
        putvarboundsliceconst(task,1, nvar+1, MSK_BK_FR, -Inf, Inf)

        r     = 0
        theta = r+1
        t     = theta+d
```

(continues on next page)

(continued from previous page)

```
        # Objective lambda*r + sum(t)
        putcj(task,r+1,lamb)
        for i in 1:n
            putcj(task,t+i, 1.0)
        end

        # Softplus function constraints
        softplus(task, d, n, theta, t, X, y)

        # Regularization
        # Append a sequence of linear expressions (r, theta) to F
        numafe = getnumafe(task)
        appendafes(task,1+d)
        putafefentry(task,numafe+1, r+1, 1.0)

        for i in 1:d
            putafefentry(task,numafe+i+1, theta+i, 1.0)
        end

        # Add the constraint
        appendaccseq(task,appendquadraticconedomain(task,1+d), numafe+1, zeros(d+1))

        # Solution
        optimize(task)

        getxxslice(task,MSK_SOL_ITR, theta+1, theta+d+1)
    end
end # logisticRegression
```

Example: 2D dataset fitting

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

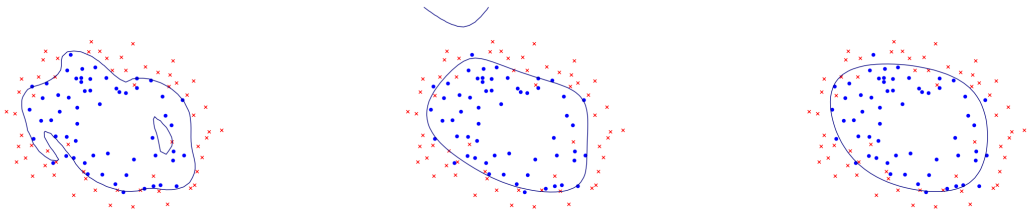


Fig. 11.2: Logistic regression example with none, medium and strong regularization (small, medium, large λ). Without regularization we get obvious overfitting.

11.3 Concurrent optimizer

The idea of the concurrent optimizer is to run multiple optimizations of **the same problem** simultaneously, and pick the one that provides the fastest or best answer. This approach is especially useful for problems which require a very long time and it is hard to say in advance which optimizer or algorithm will perform best.

The major applications of concurrent optimization we describe in this section are:

- Using the interior-point and simplex optimizers simultaneously on a linear problem. Note that any solution present in the task will also be used for hot-starting the simplex algorithms. One possible scenario would therefore be running a hot-start simplex in parallel with interior point, taking advantage of both the stability of the interior-point method and the ability of the simplex method to use an initial solution.
- Using multiple instances of the mixed-integer optimizer to solve many copies of one mixed-integer problem. This is not in contradiction with the run-to-run determinism of **MOSEK** if a different value of the MIO seed parameter *MSK_IPAR_MIO_SEED* is set in each instance. As a result each setting leads to a different optimizer run (each of them being deterministic in its own right).

The downloadable file contains usage examples of both kinds.

11.3.1 Common setup

We first define a method that runs a number of optimization tasks in parallel, using the standard multithreading setup available in the language. All tasks register for a callback function which will signal them to interrupt as soon as the first task completes successfully (with response code *MSK_RES_OK*).

Listing 11.10: Simple callback function which signals the optimizer to stop.

```
# Defines a Mosek callback function whose only function
# is to indicate if the optimizer should be stopped.
stop = false
firstStop = 0
function callback(caller  :: Callbackcode,
                  douinf  :: Vector{Float64},
                  intinf  :: Vector{Int32},
                  lintinf :: Vector{Int64})
    if stop
        1
    else
        0
    end
end
```

When all remaining tasks respond to the stop signal, response codes and statuses are returned to the caller, together with the index of the task which won the race.

Listing 11.11: A routine for parallel task race.

```
function runTask(num, task)
    global stop
    global firstStop

    ttrm =
        try
            optimize(task)
        catch e
            if isa(e,MosekError)
                return r.rcode,MSK_RES_ERR_UNKNOWN
```

(continues on next page)

(continued from previous page)

```
            else
                rethrow()
            end
        end

    # If this finished with success, inform other tasks to interrupt
    # Note that data races around stop/firstStop are irrelevant
    if ! stop
        stop = true
        firstStop = num
    end

    return MSK_RES_OK,ttrm
end

function optimizeconcurrent(tasks::Vector{Mosek.Task})
    res = [ MSK_RES_ERR_UNKNOWN for t in tasks ]
    trm = [ MSK_RES_ERR_UNKNOWN for t in tasks ]

    # Set a callback function
    for t in tasks
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        putcallbackfunc(t, callback)
    end

    # Start parallel optimizations, one per task


    Threads.@threads for i in 1:length(tasks)
        (tres,ttrm) = runTask(i,tasks[i])
        res[i] = tres
        trm[i] = ttrm
    end

    # For debugging, print res and trm codes for all optimizers
    for (i,(tres,ttrm)) in enumerate(zip(res,trm))
        println("Optimizer  $i   res $tres   trm $ttrm")
    end

  return firstStop, res, trm
end
```

11.3.2 Linear optimization

We use the multithreaded setup to run the interior-point and simplex optimizers simultaneously on a linear problem. The next methods simply clones the given task and sets a different optimizer for each. The result is the clone which finished first.

Listing 11.12: Concurrent optimization with different optimizers.

```
function optimizeconcurrent(task, optimizers)
    # Choose various optimizers for cloned tasks
    tasks = Mosek.Task[ let t = maketask(task)
                            # Use remote server: putoptserverhost(task,"http://solve.
↪mosek.com:30080")
                            putintparam(t,MSK_IPAR_OPTIMIZER, opt)
```

(continues on next page)

(continued from previous page)

```
                            t
                        end for opt in optimizers ]


    # Solve tasks in parallel
    firstOK, res, trm = optimizeconcurrent(tasks)

    if firstOK > 0
        return firstOK, tasks[firstOK], trm[firstOK], res[firstOK]
    else
        return 0, Nothing, Nothing, Nothing
    end
end
```

It remains to call the method with a choice of optimizers, for example:

Listing 11.13: Calling concurrent linear optimization.

```
optimizers = [ MSK_OPTIMIZER_CONIC,
               MSK_OPTIMIZER_DUAL_SIMPLEX,
               MSK_OPTIMIZER_PRIMAL_SIMPLEX ]
optimizeconcurrent(task, optimizers)
```

11.3.3 Mixed-integer optimization

We use the multithreaded setup to run many, differently seeded copies of the mixed-integer optimizer. This approach is most useful for hard problems where we don't expect an optimal solution in reasonable time. The input task would typically contain a time limit. It is possible that all the cloned tasks reach the time limit, in which case it doesn't really mater which one terminated first. Instead we examine all the task clones for the best objective value.

Listing 11.14: Concurrent optimization of a mixed-integer problem.

```
function optimizeconcurrentMIO(task, seeds)
    # Choose various seeds for cloned tasks
    tasks = Mosek.Task[ let t = maketask(task)
                            putintparam(MSK_IPAR_MIO_SEED, seed)
                            t
                        end for seed in seeds ]

    # Solve tasks in parallel
    (firstOK, res, trm) = optimizeconcurrent(tasks)

    sense = getobjsense(task)
    bestObj = if sense == MSK_OBJECTIVE_SENSE_MINIMIZE 1.0e+10 else -1.0e+10 end
    bestPos = -1

    if firstOK >= 0
        # Pick the task that ended with res = ok
        # and contains an integer solution with best objective value

        for (i,t) in enumerate(tasks)
            pobj = getprimalobj(t,MSK_SOL_ITG)
            print("$i   $pobj")
        end

        for (i,(tres,ttrm,t)) in enumerate(zip(res,trm,tasks))
```

(continues on next page)

(continued from previous page)

```
            solsta = getsolsta(t,MSK_SOL_ITG)
            if tres == MSK_RES_OK &&
                ( solsta == MSK_SOL_STA_PRIM_FEAS ||
                  solsta == MSK_SOL_STA_INTEGER_OPTIMAL)
                pobj = getprimalobj(t,MSK_SOL_ITG)
                if ( ( sense == MSK_OBJECTIVE_SENSE_MINIMIZE &&
                       getprimalobj(t,MSK_SOL_ITG) < bestObj ) ||
                     ( sense == MSK_OBJECTIVE_SENSE_MAXIMIZE &&
                       getprimalobj(t,MSK_SOL_ITG) > bestObj ) )
                    bestObj = pobj
                    bestPos = i
                end
            end
        end
    end

    if bestPos > 0
        return bestPos, tasks[bestPos], trm[bestPos], res[bestPos]
    else
        return 0, Nothing, Nothing, Nothing
    end
end
```

It remains to call the method with a choice of seeds, for example:

Listing 11.15: Calling concurrent integer optimization.

```
                seeds = [ 42, 13, 71749373 ]

                optimizeconcurrentMIO(task, seeds)
```

Chapter 12

Problem Formulation and Solutions

In this chapter we will discuss the following topics:

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

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

12.1 Linear Optimization

MOSEK accepts linear optimization problems of the form

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

where

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

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

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

12.1.1 Duality for Linear Optimization

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

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

where

- s_l^c are the dual variables for lower bounds of constraints,
- s_u^c are the dual variables for upper bounds of constraints,
- s_l^x are the dual variables for lower bounds of variables,
- s_u^x are the dual variables for upper bounds of variables.

If a bound in the primal problem is plus or minus infinity, the corresponding dual variable is fixed at 0, and we use the convention that the product of the bound value and the corresponding dual variable is 0. This is equivalent to removing the corresponding dual variable from the dual problem. For example:

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

A solution

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

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

A solution

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

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

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

containing the activities of linear constraints.

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

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

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

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

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

are satisfied.

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

12.1.2 Infeasibility for Linear Optimization

Primal Infeasible Problems

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

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

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

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

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

to (12.4) so that

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

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

Dual Infeasible Problems

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

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

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

where

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

and

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

such that

$$c^T x < 0.$$

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

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

12.1.3 Minimalization vs. Maximalization

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

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

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

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

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

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

such that the objective value is strictly negative

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

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

12.2 Conic Optimization

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

$$\begin{array}{lccccc}\text{minimize} & & & c^T x + c^f & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \\ & & & Fx + g & \in & \mathcal{D}, \end{array} \tag{12.7}$$

where

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

is the same as in Sec. 12.1 and moreover:

- $F \in \mathbb{R}^{k \times n}$ is the affine conic constraint matrix.,
- $g \in \mathbb{R}^k$ is the affine conic constraint constant term vector.,
- $\mathcal{D}$ is a Cartesian product of conic domains, namely $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_p$, where p is the number of individual affine conic constraints (ACCs), and each domain is one from Sec. 15.8.

The total dimension of the domain $\mathcal{D}$ must be equal to k, the number of rows in F and g. Lower and upper bounds can be infinite, or in other words the corresponding bound may be omitted.

MOSEK supports also the cone of positive semidefinite matrices. In order not to obscure this section with additional notation, that extension is discussed in Sec. 12.3.

12.2.1 Duality for Conic Optimization

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

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x - g^T \dot{y} + c^f \\ \text{subject to} & \\ & \begin{array}{c} A^T y + s_l^x - s_u^x + F^T \dot{y} = c, \\ -y + s_l^c - s_u^c = 0, \\ s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \\ \dot{y} \in \mathcal{D}^*, \end{array} \end{array} \tag{12.8}$$

where

- s_l^c are the dual variables for lower bounds of constraints,
- s_u^c are the dual variables for upper bounds of constraints,
- s_l^x are the dual variables for lower bounds of variables,
- s_u^x are the dual variables for upper bounds of variables,
- $\dot{y}$ are the dual variables for affine conic constraints,
- the dual domain $\mathcal{D}^* = \mathcal{D}_1^* \times \cdots \times \mathcal{D}_p^*$ is a Cartesian product of cones dual to $\mathcal{D}_i$.

One can check that the dual problem of the dual problem is identical to the original primal problem.

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

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

A solution

$$(y, s_l^c, s_u^c, s_l^x, s_u^x, \dot{y})$$

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

A solution

$$(x^*, y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*, (\dot{y})^*)$$

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

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

containing the activities of linear constraints.

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

$$\begin{array}{l} c^T x^* + c^f - \left\{ (l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* - g^T (\dot{y})^* + c^f \right\} \\ = \sum_{i=0}^{m-1} \left[(s_l^c)_i^* ((x_i^c)^* - l_i^c) + (s_u^c)_i^* (u_i^c - (x_i^c)^*) \right] \\ + \sum_{j=0}^{n-1} \left[(s_l^x)_j^* (x_j - l_j^x) + (s_u^x)_j^* (u_j^x - x_j^*) \right] \\ + ((\dot{y})^*)^T (Fx^* + g) \geq 0 \end{array} \tag{12.9}$$

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

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

$$\begin{array}{rcll}
(s_l^c)_i^*((x_i^c)^* - l_i^c) & = & 0, & i = 0, \ldots, m-1, \\
(s_u^c)_i^*(u_i^c - (x_i^c)^*) & = & 0, & i = 0, \ldots, m-1, \\
(s_l^x)_j^*(x_j^* - l_j^x) & = & 0, & j = 0, \ldots, n-1, \\
(s_u^x)_j^*(u_j^x - x_j^*) & = & 0, & j = 0, \ldots, n-1, \\
((\dot{y})^*)^T(Fx^* + g) & = & 0, &
\end{array} \tag{12.10}$$

are satisfied.

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

12.2.2 Infeasibility for Conic Optimization

Primal Infeasible Problems

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

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

$$\begin{array}{ll}
\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x - g^T \dot{y} \\
\text{subject to} & \\
& A^T y + s_l^x - s_u^x + F^T \dot{y} = 0, \\
& -y + s_l^c - s_u^c = 0, \\
& s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \\
& \dot{y} \in \mathcal{D}^*,
\end{array} \tag{12.11}$$

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

$$(y^*, (s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*, (\dot{y})^*)$$

to (12.11) so that

$$(l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* - g^T \dot{y} > 0.$$

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

Dual Infeasible Problems

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

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

$$\begin{array}{lccccc}
\text{minimize} & & & c^T x & & \\
\text{subject to} & \hat{l}^c & \leq & Ax & \leq & \hat{u}^c, \\
& \hat{l}^x & \leq & x & \leq & \hat{u}^x, \\
& & & Fx & \in \mathcal{D} &
\end{array} \tag{12.12}$$

where

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

and

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

such that

$$c^T x < 0.$$

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

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

12.2.3 Minimalization vs. Maximalization

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

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

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

$$\begin{array}{ll} \text{minimize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x - g^T \dot{y} + c^f \\ \text{subject to} & A^T y + s_l^x - s_u^x + F^T \dot{y} = c, \\ & -y + s_l^c - s_u^c = 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x \leq 0, \\ & -\dot{y} \in \mathcal{D}^* \end{array}$$

This means that the duality gap, defined in (12.9) as the primal minus the dual objective value, becomes nonpositive. It follows that the dual objective will always be greater than or equal to the primal objective. The primal infeasibility certificate will be reported by **MOSEK** as a solution to the system

$$\begin{array}{c} A^T y + s_l^x - s_u^x + F^T \dot{y} = 0, \\ -y + s_l^c - s_u^c = 0, \\ s_l^c, s_u^c, s_l^x, s_u^x \leq 0, \\ -\dot{y} \in \mathcal{D}^* \end{array} \tag{12.15}$$

such that the objective value is strictly negative

$$(l^c)^T (s_l^c)^* - (u^c)^T (s_u^c)^* + (l^x)^T (s_l^x)^* - (u^x)^T (s_u^x)^* - g^T \dot{y} < 0.$$

Similarly, the certificate of dual infeasibility is an x satisfying the requirements of (12.12) such that $c^T x > 0$.

12.3 Semidefinite Optimization

Semidefinite optimization is an extension of conic optimization (see Sec. 12.2) allowing positive semidefinite matrix variables to be used in addition to the usual scalar variables. All the other parts of the input are defined exactly as in Sec. 12.2, and the discussion from that section applies verbatim to all properties of problems with semidefinite variables. We only briefly indicate how the corresponding formulae should be modified with semidefinite terms.

A semidefinite optimization problem can be written as

$$\begin{array}{lccccc} \text{minimize} & & & c^T x + \langle \overline{C}, \overline{X} \rangle + c^f & & \\ \text{subject to} & l^c & \leq & Ax + \langle \overline{A}, \overline{X} \rangle & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \\ & & & Fx + \langle \overline{F}, \overline{X} \rangle + g & \in & \mathcal{D}, \\ & & & \overline{X}_j & \in & \mathcal{S}_+^{r_j}, j = 1, \ldots, s \end{array}$$

where

- m is the number of constraints.
- n is the number of decision variables.
- $x \in \mathbb{R}^n$ is a vector of decision variables.
- $c \in \mathbb{R}^n$ is the linear part of the objective function.
- $c^f \in \mathbb{R}$ is a constant term in the objective
- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $l^c \in \mathbb{R}^m$ is the lower limit on the activity for the constraints.
- $u^c \in \mathbb{R}^m$ is the upper limit on the activity for the constraints.
- $l^x \in \mathbb{R}^n$ is the lower limit on the activity for the variables.
- $u^x \in \mathbb{R}^n$ is the upper limit on the activity for the variables.
- $F \in \mathbb{R}^{k \times n}$ is the affine conic constraint matrix.,
- $g \in \mathbb{R}^k$ is the affine conic constraint constant term vector.,
- $\mathcal{D}$ is a Cartesian product of conic domains, namely $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_p$, where p is the number of individual affine conic constraints (ACCs), and each domain is one from Sec. 15.8.

is the same as in Sec. 12.2 and moreover:

- there are s symmetric positive semidefinite variables, the j-th of which is $\overline{X}_j \in \mathcal{S}_+^{r_j}$ of dimension r_j,
- $\overline{C} = (\overline{C}_j)_{j=1,\ldots,s}$ is a collection of symmetric coefficient matrices in the objective, with $\overline{C}_j \in \mathcal{S}^{r_j}$, and we interpret the notation $\langle \overline{C}, \overline{X} \rangle$ as a shorthand for

$$\langle \overline{C}, \overline{X} \rangle := \sum_{j=1}^{s} \langle \overline{C}_j, \overline{X}_j \rangle.$$

- $\overline{A} = (\overline{A}_{ij})_{i=1,\ldots,m, j=1,\ldots,s}$ is a collection of symmetric coefficient matrices in the constraints, with $\overline{A}_{ij} \in \mathcal{S}^{r_j}$, and we interpret the notation $\langle \overline{A}, \overline{X} \rangle$ as a shorthand for the vector

$$\langle \overline{A}, \overline{X} \rangle := \left(\sum_{j=1}^{s} \langle \overline{A}_{ij}, \overline{X}_j \rangle \right)_{i=1,\ldots,m}.$$

- $\overline{F} = (\overline{F}_{ij})_{i=1,\ldots,k, j=1,\ldots,s}$ is a collection of symmetric coefficient matrices in the affine conic constraints, with $\overline{F}_{ij} \in \mathcal{S}^{r_j}$, and we interpret the notation $\langle \overline{F}, \overline{X} \rangle$ as a shorthand for the vector

$$\langle \overline{F}, \overline{X} \rangle := \left(\sum_{j=1}^{s} \langle \overline{F}_{ij}, \overline{X}_j \rangle \right)_{i=1,\ldots,k}.$$

In each case the matrix inner product between symmetric matrices of the same dimension r is defined as

$$\langle U, V \rangle := \sum_{i=1}^{r} \sum_{j=1}^{r} U_{ij} V_{ij}.$$

To summarize, above the formulation extends that from Sec. 12.2 by the possibility of including semidefinite terms in the objective, constraints and affine conic constraints.

Duality

The definition of the dual problem (12.8) becomes:

$$\begin{array}{ll}\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x - g^T \dot{y} + c^f \\ \text{subject to} & \\ & \begin{array}{ll} A^T y + s_l^x - s_u^x + F^T \dot{y} = c, & \\ -y + s_l^c - s_u^c = 0, & \\ \overline{C}_j - \sum_{i=1}^m y_i \overline{A}_{ij} - \sum_{i=1}^k \dot{y}_i \overline{F}_{ij} = S_j, & j = 1, \ldots, s, \\ s_l^c, s_u^c, s_l^x, s_u^x \geq 0, & \\ \dot{y} \in \mathcal{D}^*, & \\ \overline{S}_j \in \mathcal{S}_+^{r_j}, & j = 1, \ldots, s. \end{array} \end{array} \tag{12.16}$$

Complementarity conditions (12.10) include the additional relation:

$$\langle \overline{X}_j, \overline{S}_j \rangle = 0 \quad j = 1, \ldots, s. \tag{12.17}$$

Infeasibility

A certificate of primal infeasibility (12.11) is now a feasible solution to:

$$\begin{array}{ll}\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x - g^T \dot{y} \\ \text{subject to} & \\ & \begin{array}{ll} A^T y + s_l^x - s_u^x + F^T \dot{y} = 0, & \\ -y + s_l^c - s_u^c = 0, & \\ -\sum_{i=1}^m y_i \overline{A}_{ij} - \sum_{i=1}^k \dot{y}_i \overline{F}_{ij} = S_j, & j = 1, \ldots, s, \\ s_l^c, s_u^c, s_l^x, s_u^x \geq 0, & \\ \dot{y} \in \mathcal{D}^*, & \\ \overline{S}_j \in \mathcal{S}_+^{r_j}, & j = 1, \ldots, s. \end{array} \end{array} \tag{12.18}$$

such that the objective value is strictly positive.

Similarly, a dual infeasibility certificate (12.12) is a feasible solution to

$$\begin{array}{lccccl}\text{minimize} & & & c^T x + \langle \overline{C}, \overline{X} \rangle & & \\ \text{subject to} & \hat{l}^c & \leq & Ax + \langle \overline{A}, \overline{X} \rangle & \leq & \hat{u}^c, \\ & \hat{l}^x & \leq & x & \leq & \hat{u}^x, \\ & & & Fx + \langle \overline{F}, \overline{X} \rangle & \in & \mathcal{D}, \\ & & & \overline{X}_j & \in & \mathcal{S}_+^{r_j}, j = 1, \ldots, s \end{array} \tag{12.19}$$

where the modified bounds are as in (12.13) and (12.14) and the objective value is strictly negative.

12.4 Quadratic and Quadratically Constrained Optimization

A convex quadratic and quadratically constrained optimization problem has the form

$$\begin{array}{lccccll}\text{minimize} & & & \frac{1}{2} x^T Q^o x + c^T x + c^f & & & \\ \text{subject to} & l_k^c & \leq & \frac{1}{2} x^T Q^k x + \sum_{j=0}^{n-1} a_{kj} x_j & \leq & u_k^c, & k = 0, \ldots, m-1, \\ & l_j^x & \leq & x_j & \leq & u_j^x, & j = 0, \ldots, n-1, \end{array} \tag{12.20}$$

where all variables and bounds have the same meaning as for linear problems (see Sec. 12.1) and Q^o and all Q^k are symmetric matrices. Moreover, for convexity, Q^o must be a positive semidefinite matrix and Q^k must satisfy

$$\begin{array}{rcl} -\infty < l_k^c & \Rightarrow & Q^k \text{ is negative semidefinite,} \\ u_k^c < \infty & \Rightarrow & Q^k \text{ is positive semidefinite,} \\ -\infty < l_k^c \leq u_k^c < \infty & \Rightarrow & Q^k = 0. \end{array}$$

The convexity requirement is very important and **MOSEK** checks whether it is fulfilled.

12.4.1 A Recommendation

Any convex quadratic optimization problem can be reformulated as a conic quadratic optimization problem, see Modeling Cookbook and [And13]. In fact **MOSEK** does such conversion internally as a part of the solution process for the following reasons:

- the conic optimizer is numerically more robust than the one for quadratic problems.
- the conic optimizer is usually faster because quadratic cones are simpler than quadratic functions, even though the conic reformulation usually has more constraints and variables than the original quadratic formulation.
- it is easy to dualize the conic formulation if deemed worthwhile potentially leading to (huge) computational savings.

However, instead of relying on the automatic reformulation we recommend to formulate the problem as a conic problem from scratch because:

- it saves the computational overhead of the reformulation including the convexity check. A conic problem is convex by construction and hence no convexity check is needed for conic problems.
- usually the modeler can do a better reformulation than the automatic method because the modeler can exploit the knowledge of the problem at hand.

To summarize we recommend to formulate quadratic problems and in particular quadratically constrained problems directly in conic form.

12.4.2 Duality for Quadratic and Quadratically Constrained Optimization

The dual problem corresponding to the quadratic and quadratically constrained optimization problem (12.20) is given by

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + \frac{1}{2} x^T \left\{ \sum_{k=0}^{m-1} y_k Q^k - Q^o \right\} x + c^f \\ \text{subject to} & A^T y + s_l^x - s_u^x + \left\{ \sum_{k=0}^{m-1} y_k Q^k - Q^o \right\} x = c, \\ & -y + s_l^c - s_u^c = 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x \geq 0. \end{array} \tag{12.21}$$

The dual problem is related to the dual problem for linear optimization (see Sec. 12.1.1), but depends on the variable x which in general can not be eliminated. In the solutions reported by **MOSEK**, the value of x is the same for the primal problem (12.20) and the dual problem (12.21).

12.4.3 Infeasibility for Quadratic Optimization

In case **MOSEK** finds a problem to be infeasible it reports a certificate of infeasibility. We write them out explicitly for quadratic problems, that is when $Q^k = 0$ for all k and quadratic terms appear only in the objective Q^o. In this case the constraints both in the primal and dual problem are linear, and **MOSEK** produces for them the same infeasibility certificate as for linear problems.

The certificate of primal infeasibility is a solution to the problem (12.4) such that the objective value is strictly positive.

The certificate of dual infeasibility is a solution to the problem (12.5) together with an additional constraint

$$Q^o x = 0$$

such that the objective value is strictly negative.

Below is an outline of the different problem types for quick reference.

Continuous problem formulations

- **Linear optimization (LO)**

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

- **Conic optimization (CO)**

  Conic optimization extends linear optimization with *affine conic constraints* (ACC):

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

  where $\mathcal{D}$ is a product of domains from Sec. 15.8.

- **Semidefinite optimization (SDO)**

  A conic optimization problem can be further extended with *semidefinite variables*:

$$\begin{array}{lccccc} \text{minimize} & & & c^T x + \langle \overline{C}, \overline{X} \rangle + c^f & & \\ \text{subject to} & l^c & \leq & Ax + \langle \overline{A}, \overline{X} \rangle & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \\ & & & Fx + \langle \overline{F}, \overline{X} \rangle + g & \in & \mathcal{D}, \\ & & & \overline{X} & \in & \mathcal{S}_+, \end{array}$$

  where $\mathcal{D}$ is a product of domains from Sec. 15.8 and $\mathcal{S}_+$ is a product of PSD cones meaning that $\overline{X}$ is a sequence of PSD matrix variables.

- **Quadratic and quadratically constrained optimization (QO, QCQO)**

  A quadratic problem or quadratically constrained problem has the form

$$\begin{array}{lccccc} \text{minimize} & & & \frac{1}{2} x^T Q^o x + c^T x + c^f & & \\ \text{subject to} & l^c & \leq & \frac{1}{2} x^T Q^c x + Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x. \end{array}$$

Mixed-integer extensions

Coninuous problems can be extended with constraints requiring the mixed-integer optimizer. We outline them briefly here. The continuous part of a mixed-integer problem is formulated according to one of the continuous types above, however only the primal information and solution fields are relevant, there are no dual values and no infeasibility certificates.

- **Integer variables.** Specifies that a subset of variables take integer values, that is

$$x_I \in \mathbb{Z}$$

  for some index set I.

- **Disjunctive constraints.** Appends disjunctions of the form

$$\bigvee_{i=1}^{t} \bigwedge_{j=1}^{s_i} (D_{ij} x + d_{ij} \in \mathcal{D}_{ij})$$

  ie. a disjunction of conjunctions of linear constraints, where each $D_{ij} x + d_{ij}$ is an affine expression of the optimization variables and each $\mathcal{D}_{ij}$ is an affine domain. Linear and conic problems can be extended with disjunctive constraints.

Chapter 13

Optimizers

The most essential part of **MOSEK** are the optimizers:

- *primal simplex* (linear problems),
- *dual simplex* (linear problems),
- *interior-point* (linear, quadratic and conic problems),
- *mixed-integer* (problems with integer variables).

The structure of a successful optimization process is roughly:

- **Presolve**
  1. *Elimination*: Reduce the size of the problem.
  2. *Dualizer*: Choose whether to solve the primal or the dual form of the problem.
  3. *Scaling*: Scale the problem for better numerical stability.
- **Optimization**
  1. *Optimize*: Solve the problem using selected method.
  2. *Terminate*: Stop the optimization when specific termination criteria have been met.
  3. *Report*: Return the solution or an infeasibility certificate.

The preprocessing stage is transparent to the user, but useful to know about for tuning purposes. The purpose of the preprocessing steps is to make the actual optimization more efficient and robust. We discuss the details of the above steps in the following sections.

13.1 Presolve

Before an optimizer actually performs the optimization the problem is preprocessed using the so-called presolve. The purpose of the presolve is to

1. remove redundant constraints,
2. eliminate fixed variables,
3. remove linear dependencies,
4. substitute out (implied) free variables, and
5. reduce the size of the optimization problem in general.

After the presolved problem has been optimized the solution is automatically postsolved so that the returned solution is valid for the original problem. Hence, the presolve is completely transparent. For further details about the presolve phase, please see [AA95] and [AGMeszarosX96].

It is possible to fine-tune the behavior of the presolve or to turn it off entirely. If presolve consumes too much time or memory compared to the reduction in problem size gained it may be disabled. This is done by setting the parameter *MSK_IPAR_PRESOLVE_USE* to *MSK_PRESOLVE_MODE_OFF*.

In the following we describe in more detail the presolve applied to continuous, i.e., linear and conic optimization problems, see Sec. 13.2 and Sec. 13.3. The mixed-integer optimizer, Sec. 13.4, applies similar techniques. The two most time-consuming steps of the presolve for continuous optimization problems are

- the eliminator, and
- the linear dependency check.

Therefore, in some cases it is worthwhile to disable one or both of these.

Numerical issues in the presolve

During the presolve the problem is reformulated so that it hopefully solves faster. However, in rare cases the presolved problem may be harder to solve then the original problem. The presolve may also be infeasible although the original problem is not. If it is suspected that presolved problem is much harder to solve than the original, we suggest to first turn the eliminator off by setting the parameter *MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_NUM_TRIES* to 0. If that does not help, then trying to turn entire presolve off may help.

Since all computations are done in finite precision, the presolve employs some tolerances when concluding a variable is fixed or a constraint is redundant. If it happens that **MOSEK** incorrectly concludes a problem is primal or dual infeasible, then it is worthwhile to try to reduce the parameters *MSK_DPAR_PRESOLVE_TOL_X* and *MSK_DPAR_PRESOLVE_TOL_S*. However, if reducing the parameters actually helps then this should be taken as an indication that the problem is badly formulated.

Eliminator

The purpose of the eliminator is to eliminate free and implied free variables from the problem using substitution. For instance, given the constraints

$$\begin{array}{rcl} y & = & \sum_j x_j, \\ y, x & \geq & 0, \end{array}$$

y is an implied free variable that can be substituted out of the problem, if deemed worthwhile. If the eliminator consumes too much time or memory compared to the reduction in problem size gained it may be disabled. This can be done by setting the parameter *MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_NUM_TRIES* to 0. In rare cases the eliminator may cause that the problem becomes much hard to solve.

Linear dependency checker

The purpose of the linear dependency check is to remove linear dependencies among the linear equalities. For instance, the three linear equalities

$$\begin{array}{rcl} x_1 + x_2 + x_3 & = & 1, \\ x_1 + 0.5x_2 & = & 0.5, \\ 0.5x_2 + x_3 & = & 0.5. \end{array}$$

contain exactly one linear dependency. This implies that one of the constraints can be dropped without changing the set of feasible solutions. Removing linear dependencies is in general a good idea since it reduces the size of the problem. Moreover, the linear dependencies are likely to introduce numerical problems in the optimization phase. It is best practice to build models without linear dependencies, but that is not always easy for the user to control. If the linear dependencies are removed at the modeling stage, the linear dependency check can safely be disabled by setting the parameter *MSK_IPAR_PRESOLVE_LINDEP_USE* to *MSK_OFF*.

Dualizer

All linear, conic, and convex optimization problems have an equivalent dual problem associated with them. **MOSEK** has built-in heuristics to determine if it is more efficient to solve the primal or dual problem. The form (primal or dual) is displayed in the **MOSEK** log and available as an information item from the solver. Should the internal heuristics not choose the most efficient form of the problem it may be worthwhile to set the dualizer manually by setting the parameters:

- *MSK_IPAR_INTPNT_SOLVE_FORM*: In case of the interior-point optimizer.
- *MSK_IPAR_SIM_SOLVE_FORM*: In case of the simplex optimizer.

Note that currently only linear and conic (but not semidefinite) problems may be automatically dualized.

Scaling

Problems containing data with large and/or small coefficients, say $1.0e+9$ or $1.0e-7$, are often hard to solve. Significant digits may be truncated in calculations with finite precision, which can result in the optimizer relying on inaccurate data. Since computers work in finite precision, extreme coefficients should be avoided. In general, data around the same *order of magnitude* is preferred, and we will refer to a problem, satisfying this loose property, as being *well-scaled*. If the problem is not well scaled, **MOSEK** will try to scale (multiply) constraints and variables by suitable constants. **MOSEK** solves the scaled problem to improve the numerical properties.

The scaling process is transparent, i.e. the solution to the original problem is reported. It is important to be aware that the optimizer terminates when the termination criterion is met on the scaled problem, therefore significant primal or dual infeasibilities may occur after unscaling for badly scaled problems. The best solution of this issue is to reformulate the problem, making it better scaled.

By default **MOSEK** heuristically chooses a suitable scaling. The scaling for interior-point and simplex optimizers can be controlled with the parameters *MSK_IPAR_INTPNT_SCALING* and *MSK_IPAR_SIM_SCALING* respectively.

13.2 Linear Optimization

13.2.1 Optimizer Selection

Two different types of optimizers are available for linear problems: The default is an interior-point method, and the alternative is the simplex method (primal or dual). The optimizer can be selected using the parameter *MSK_IPAR_OPTIMIZER*.

The Interior-point or the Simplex Optimizer?

Given a linear optimization problem, which optimizer is the best: the simplex or the interior-point optimizer? It is impossible to provide a general answer to this question. However, the interior-point optimizer behaves more predictably: it tends to use between 20 and 100 iterations, almost independently of problem size, but cannot perform warm-start. On the other hand the simplex method can take advantage of an initial solution, but is less predictable from cold-start. The interior-point optimizer is used by default.

The Primal or the Dual Simplex Variant?

MOSEK provides both a primal and a dual simplex optimizer. Predicting which simplex optimizer is faster is impossible, however, in recent years the dual optimizer has seen several algorithmic and computational improvements, which, in our experience, make it faster on average than the primal version. Still, it depends much on the problem structure and size. Setting the *MSK_IPAR_OPTIMIZER* parameter to *MSK_OPTIMIZER_FREE_SIMPLEX* instructs **MOSEK** to choose one of the simplex variants automatically.

To summarize, if you want to know which optimizer is faster for a given problem type, it is best to try all the options.

13.2.2 The Interior-point Optimizer

The purpose of this section is to provide information about the algorithm employed in the **MOSEK** interior-point optimizer for linear problems and about its termination criteria.

The homogeneous primal-dual problem

In order to keep the discussion simple it is assumed that **MOSEK** solves linear optimization problems of standard form

$$\begin{array}{lrcl} \text{minimize} & c^T x & & \\ \text{subject to} & Ax & = & b, \\ & x \geq 0. & & \end{array} \tag{13.1}$$

This is in fact what happens inside **MOSEK**; for efficiency reasons **MOSEK** converts the problem to standard form before solving, then converts it back to the input form when reporting the solution.

Since it is not known beforehand whether problem (13.1) has an optimal solution, is primal infeasible or is dual infeasible, the optimization algorithm must deal with all three situations. This is the reason why **MOSEK** solves the so-called homogeneous model

$$\begin{array}{rcl} Ax - b\tau & = & 0, \\ A^T y + s - c\tau & = & 0, \\ -c^T x + b^T y - \kappa & = & 0, \\ x, s, \tau, \kappa & \geq & 0, \end{array} \tag{13.2}$$

where y and s correspond to the dual variables in (13.1), and τ and κ are two additional scalar variables. Note that the homogeneous model (13.2) always has solution since

$$(x, y, s, \tau, \kappa) = (0, 0, 0, 0, 0)$$

is a solution, although not a very interesting one. Any solution

$$(x^*, y^*, s^*, \tau^*, \kappa^*)$$

to the homogeneous model (13.2) satisfies

$$x_j^* s_j^* = 0 \text{ and } \tau^* \kappa^* = 0.$$

Moreover, there is always a solution that has the property $\tau^* + \kappa^* > 0$.

First, assume that $\tau^* > 0$. It follows that

$$\begin{array}{rcl} A\frac{x^*}{\tau^*} & = & b, \\ A^T \frac{y^*}{\tau^*} + \frac{s^*}{\tau^*} & = & c, \\ -c^T \frac{x^*}{\tau^*} + b^T \frac{y^*}{\tau^*} & = & 0, \\ x^*, s^*, \tau^*, \kappa^* & \geq & 0. \end{array}$$

This shows that $\frac{x^*}{\tau^*}$ is a primal optimal solution and $(\frac{y^*}{\tau^*}, \frac{s^*}{\tau_*})$ is a dual optimal solution; this is reported as the optimal interior-point solution since

$$(x, y, s) = \left\{ \frac{x^*}{\tau^*}, \frac{y^*}{\tau^*}, \frac{s^*}{\tau^*} \right\}$$

is a primal-dual optimal solution (see Sec. 12.1 for the mathematical background on duality and optimality).

On other hand, if $\kappa^* > 0$ then

$$\begin{array}{rcl} Ax^* & = & 0, \\ A^T y^* + s^* & = & 0, \\ -c^T x^* + b^T y^* & = & \kappa^*, \\ x^*, s^*, \tau^*, \kappa^* & \geq & 0. \end{array}$$

This implies that at least one of

$$c^T x^* < 0 \tag{13.3}$$

or

$$b^T y^* > 0 \tag{13.4}$$

is satisfied. If (13.3) is satisfied then x^* is a certificate of dual infeasibility, whereas if (13.4) is satisfied then y^* is a certificate of primal infeasibility.

In summary, by computing an appropriate solution to the homogeneous model, all information required for a solution to the original problem is obtained. A solution to the homogeneous model can be computed using a primal-dual interior-point algorithm [And09].

Interior-point Termination Criterion

For efficiency reasons it is not practical to solve the homogeneous model exactly. Hence, an exact optimal solution or an exact infeasibility certificate cannot be computed and a reasonable termination criterion has to be employed.

In the k-th iteration of the interior-point algorithm a trial solution

$$(x^k, y^k, s^k, \tau^k, \kappa^k)$$

to homogeneous model is generated, where

$$x^k, s^k, \tau^k, \kappa^k > 0.$$

Optimal case

Whenever the trial solution satisfies the criterion

$$\begin{array}{rcl} \left\| A\frac{x^k}{\tau^k} - b \right\|_\infty & \leq & \epsilon_p(1 + \|b\|_\infty), \\ \left\| A^T\frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c \right\|_\infty & \leq & \epsilon_d(1 + \|c\|_\infty), \text{ and} \\ \min\left(\frac{(x^k)^T s^k}{(\tau^k)^2}, |\frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k}|\right) & \leq & \epsilon_g \max\left(1, \frac{\min(|c^T x^k|, |b^T y^k|)}{\tau^k}\right), \end{array} \tag{13.5}$$

the interior-point optimizer is terminated and

$$\frac{(x^k, y^k, s^k)}{\tau^k}$$

is reported as the primal-dual optimal solution. The interpretation of (13.5) is that the optimizer is terminated if

- $\frac{x^k}{\tau^k}$ is approximately primal feasible,
- $\left\{\frac{y^k}{\tau^k}, \frac{s^k}{\tau^k}\right\}$ is approximately dual feasible, and
- the duality gap is almost zero.

Dual infeasibility certificate

On the other hand, if the trial solution satisfies

$$-\epsilon_i c^T x^k > \frac{\|c\|_\infty}{\max(1, \|b\|_\infty)} \left\| Ax^k \right\|_\infty$$

then the problem is declared dual infeasible and x^k is reported as a certificate of dual infeasibility. The motivation for this stopping criterion is as follows: First assume that $\left\| Ax^k \right\|_\infty = 0$; then x^k is an exact certificate of dual infeasibility. Next assume that this is not the case, i.e.

$$\left\| Ax^k \right\|_\infty > 0,$$

and define

$$\bar{x} := \epsilon_i \frac{\max(1, \|b\|_\infty)}{\|Ax^k\|_\infty \|c\|_\infty} x^k.$$

It is easy to verify that

$$\|A\bar{x}\|_\infty = \epsilon_i \frac{\max(1, \|b\|_\infty)}{\|c\|_\infty} \text{ and } -c^T\bar{x} > 1,$$

which shows $\bar{x}$ is an approximate certificate of dual infeasibility, where ε_i controls the quality of the approximation. A smaller value means a better approximation.

Primal infeasibility certificate

Finally, if

$$\epsilon_i b^T y^k > \frac{\|b\|_\infty}{\max(1, \|c\|_\infty)} \left\|A^T y^k + s^k\right\|_\infty$$

then y^k is reported as a certificate of primal infeasibility.

Adjusting optimality criteria

It is possible to adjust the tolerances ε_p, ε_d, ε_g and ε_i using parameters; see table for details.

Table 13.1: Parameters employed in termination criterion

ToleranceParameter	name
ε_p	*MSK_DPAR_INTPNT_TOL_PFEAS*
ε_d	*MSK_DPAR_INTPNT_TOL_DFEAS*
ε_g	*MSK_DPAR_INTPNT_TOL_REL_GAP*
ε_i	*MSK_DPAR_INTPNT_TOL_INFEAS*

The default values of the termination tolerances are chosen such that for a majority of problems appearing in practice it is not possible to achieve much better accuracy. Therefore, tightening the tolerances usually is not worthwhile. However, an inspection of (13.5) reveals that the quality of the solution depends on $\|b\|_\infty$ and $\|c\|_\infty$; the smaller the norms are, the better the solution accuracy.

The interior-point method as implemented by **MOSEK** will converge toward optimality and primal and dual feasibility at the same rate [And09]. This means that if the optimizer is stopped prematurely then it is very unlikely that either the primal or dual solution is feasible. Another consequence is that in most cases all the tolerances, ε_p, ε_d, ε_g and ε_i, have to be relaxed together to achieve an effect.

The basis identification discussed in Sec. 13.2.2 requires an optimal solution to work well; hence basis identification should be turned off if the termination criterion is relaxed.

To conclude the discussion in this section, relaxing the termination criterion is usually not worthwhile.

Basis Identification

An interior-point optimizer does not return an optimal basic solution unless the problem has a unique primal and dual optimal solution. Therefore, the interior-point optimizer has an optional post-processing step that computes an optimal basic solution starting from the optimal interior-point solution. More information about the basis identification procedure may be found in [AY96]. In the following we provide an overall idea of the procedure.

There are some cases in which a basic solution could be more valuable:

- a basic solution is often more accurate than an interior-point solution,
- a basic solution can be used to warm-start the simplex algorithm in case of reoptimization,
- a basic solution is in general more sparse, i.e. more variables are fixed to zero. This is particularly appealing when solving continuous relaxations of mixed integer problems, as well as in all applications in which sparser solutions are preferred.

To illustrate how the basis identification routine works, we use the following trivial example:

$$\begin{array}{lll} \text{minimize} & x+y & \\ \text{subject to} & x+y & = \quad 1, \\ & x,y \geq 0. & \end{array}$$

It is easy to see that all feasible solutions are also optimal. In particular, there are two basic solutions, namely

$$\begin{array}{rcl} (x_1^*, y_1^*) & = & (1,0), \\ (x_2^*, y_2^*) & = & (0,1). \end{array}$$

The interior point algorithm will actually converge to the center of the optimal set, i.e. to $(x^*, y^*) = (1/2, 1/2)$ (to see this in **MOSEK** deactivate *Presolve*).

In practice, when the algorithm gets close to the optimal solution, it is possible to construct in polynomial time an initial basis for the simplex algorithm from the current interior point solution. This basis is used to warm-start the simplex algorithm that will provide the optimal basic solution. In most cases the constructed basis is optimal, or very few iterations are required by the simplex algorithm to make it optimal and hence the final *clean-up* phase be short. However, for some cases of ill-conditioned problems the additional simplex clean up phase may take of lot a time.

By default **MOSEK** performs a basis identification. However, if a basic solution is not needed, the basis identification procedure can be turned off. The parameters

- *MSK_IPAR_INTPNT_BASIS*,
- *MSK_IPAR_BI_IGNORE_MAX_ITER*, and
- *MSK_IPAR_BI_IGNORE_NUM_ERROR*

control when basis identification is performed.

The type of simplex algorithm to be used (primal/dual) can be tuned with the parameter *MSK_IPAR_BI_CLEAN_OPTIMIZER*, and the maximum number of iterations can be set with *MSK_IPAR_BI_MAX_ITERATIONS*.

Finally, it should be mentioned that there is no guarantee on which basic solution will be returned.

The Interior-point Log

Below is a typical log output from the interior-point optimizer:

```
Optimizer  - threads                : 1
Optimizer  - solved problem         : the dual
Optimizer  - Constraints            : 2
Optimizer  - Cones                  : 0
Optimizer  - Scalar variables       : 6                 conic                  : 0
Optimizer  - Semi-definite variables: 0                 scalarized             : 0
Factor     - setup time             : 0.00              dense det. time        : 0.00
Factor     - ML order time          : 0.00              GP order time          : 0.00
Factor     - nonzeros before factor : 3                 after factor           : 3
Factor     - dense dim.             : 0                 flops                  : 7.
↪00e+001
ITE PFEAS    DFEAS    GFEAS    PRSTATUS   POBJ              DOBJ              MU
↪ TIME
0   1.0e+000 8.6e+000 6.1e+000 1.00e+000  0.000000000e+000  -2.208000000e+003 1.
↪0e+000 0.00
1   1.1e+000 2.5e+000 1.6e-001 0.00e+000  -7.901380925e+003 -7.394611417e+003 2.
↪5e+000 0.00
2   1.4e-001 3.4e-001 2.1e-002 8.36e-001  -8.113031650e+003 -8.055866001e+003 3.3e-
↪001 0.00
3   2.4e-002 5.8e-002 3.6e-003 1.27e+000  -7.777530698e+003 -7.766471080e+003 5.7e-
↪002 0.01
```

(continues on next page)

(continued from previous page)

```
4   1.3e-004 3.2e-004 2.0e-005 1.08e+000  -7.668323435e+003 -7.668207177e+003 3.2e-
↪004 0.01
5   1.3e-008 3.2e-008 2.0e-009 1.00e+000  -7.668000027e+003 -7.668000015e+003 3.2e-
↪008 0.01
6   1.3e-012 3.2e-012 2.0e-013 1.00e+000  -7.667999994e+003 -7.667999994e+003 3.2e-
↪012 0.01
```

The first line displays the number of threads used by the optimizer and the second line indicates if the optimizer chose to solve the primal or dual problem (see *MSK_IPAR_INTPNT_SOLVE_FORM*). The next lines display the problem dimensions as seen by the optimizer, and the `Factor...` lines show various statistics. This is followed by the iteration log.

Using the same notation as in Sec. 13.2.2 the columns of the iteration log have the following meaning:

- `ITE`: Iteration index k.
- `PFEAS`: $\left\|Ax^k - b\tau^k\right\|_\infty$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- `DFEAS`: $\left\|A^T y^k + s^k - c\tau^k\right\|_\infty$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- `GFEAS`: $|-c^T x^k + b^T y^k - \kappa^k|$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- `PRSTATUS`: This number converges to 1 if the problem has an optimal solution whereas it converges to -1 if that is not the case.
- `POBJ`: $c^T x^k/\tau^k$. An estimate for the primal objective value.
- `DOBJ`: $b^T y^k/\tau^k$. An estimate for the dual objective value.
- `MU`: $\frac{(x^k)^T s^k + \tau^k \kappa^k}{n+1}$. The numbers in this column should always converge to zero.
- `TIME`: Time spent since the optimization started.

13.2.3 The Simplex Optimizer

An alternative to the interior-point optimizer is the simplex optimizer. The simplex optimizer uses a different method that allows exploiting an initial guess for the optimal solution to reduce the solution time. Depending on the problem it may be faster or slower to use an initial guess; see Sec. 13.2.1 for a discussion. **MOSEK** provides both a primal and a dual variant of the simplex optimizer.

Simplex Termination Criterion

The simplex optimizer terminates when it finds an optimal basic solution or an infeasibility certificate. A basic solution is optimal when it is primal and dual feasible; see Sec. 12.1 for a definition of the primal and dual problem. Due to the fact that computations are performed in finite precision **MOSEK** allows violations of primal and dual feasibility within certain tolerances. The user can control the allowed primal and dual tolerances with the parameters *MSK_DPAR_BASIS_TOL_X* and *MSK_DPAR_BASIS_TOL_S*.

Setting the parameter *MSK_IPAR_OPTIMIZER* to *MSK_OPTIMIZER_FREE_SIMPLEX* instructs **MOSEK** to select automatically between the primal and the dual simplex optimizers. Hence, **MOSEK** tries to choose the best optimizer for the given problem and the available solution. The same parameter can also be used to force one of the variants.

Starting From an Existing Solution

When using the simplex optimizer it may be possible to reuse an existing solution and thereby reduce the solution time significantly. When a simplex optimizer starts from an existing solution it is said to perform a *warm-start*. If the user is solving a sequence of optimization problems by solving the problem, making modifications, and solving again, **MOSEK** will warm-start automatically.

By default **MOSEK** uses presolve when performing a warm-start. If the optimizer only needs very few iterations to find the optimal solution it may be better to turn off the presolve.

Numerical Difficulties in the Simplex Optimizers

Though **MOSEK** is designed to minimize numerical instability, completely avoiding it is impossible when working in finite precision. **MOSEK** treats a "numerically unexpected behavior" event inside the optimizer as a *set-back*. The user can define how many set-backs the optimizer accepts; if that number is exceeded, the optimization will be aborted. Set-backs are a way to escape long sequences where the optimizer tries to recover from an unstable situation.

Examples of set-backs are: repeated singularities when factorizing the basis matrix, repeated loss of feasibility, degeneracy problems (no progress in objective) and other events indicating numerical difficulties. If the simplex optimizer encounters a lot of set-backs the problem is usually badly scaled; in such a situation try to reformulate it into a better scaled problem. Then, if a lot of set-backs still occur, trying one or more of the following suggestions may be worthwhile:

- Raise tolerances for allowed primal or dual feasibility: increase the value of
  - *MSK_DPAR_BASIS_TOL_X*, and
  - *MSK_DPAR_BASIS_TOL_S*.
- Raise or lower pivot tolerance: Change the *MSK_DPAR_SIMPLEX_ABS_TOL_PIV* parameter.
- Switch optimizer: Try another optimizer.
- Switch off crash: Set both *MSK_IPAR_SIM_PRIMAL_CRASH* and *MSK_IPAR_SIM_DUAL_CRASH* to 0.
- Experiment with other pricing strategies: Try different values for the parameters
  - *MSK_IPAR_SIM_PRIMAL_SELECTION* and
  - *MSK_IPAR_SIM_DUAL_SELECTION*.
- If you are using warm-starts, in rare cases switching off this feature may improve stability. This is controlled by the *MSK_IPAR_SIM_HOTSTART* parameter.
- Increase maximum number of set-backs allowed controlled by *MSK_IPAR_SIM_MAX_NUM_SETBACKS*.
- If the problem repeatedly becomes infeasible try switching off the special degeneracy handling. See the parameter *MSK_IPAR_SIM_DEGEN* for details.

The Simplex Log

Below is a typical log output from the simplex optimizer:

```
Optimizer  - solved problem         : the primal
Optimizer  - Constraints            : 667
Optimizer  - Scalar variables       : 1424              conic                  : 0
Optimizer  - hotstart               : no
ITER      DEGITER(%)  PFEAS       DFEAS       POBJ                   DOBJ           ␣
↪     TIME      TOTTIME
0         0.00        1.43e+05    NA          6.5584140832e+03       NA             ␣
↪     0.00      0.02
1000      1.10        0.00e+00    NA          1.4588289726e+04       NA             ␣
↪     0.13      0.14
2000      0.75        0.00e+00    NA          7.3705564855e+03       NA             ␣
```

(continues on next page)

(continued from previous page)

```
↪    0.21      0.22
3000     0.67        0.00e+00    NA          6.0509727712e+03      NA          ␣
↪    0.29      0.31
4000     0.52        0.00e+00    NA          5.5771203906e+03      NA          ␣
↪    0.38      0.39
4533     0.49        0.00e+00    NA          5.5018458883e+03      NA          ␣
↪    0.42      0.44
```

The first lines summarize the problem the optimizer is solving. This is followed by the iteration log, with the following meaning:

- `ITER`: Number of iterations.
- `DEGITER(%)`: Ratio of degenerate iterations.
- `PFEAS`: Primal feasibility measure reported by the simplex optimizer. The numbers should be 0 if the problem is primal feasible (when the primal variant is used).
- `DFEAS`: Dual feasibility measure reported by the simplex optimizer. The number should be 0 if the problem is dual feasible (when the dual variant is used).
- `POBJ`: An estimate for the primal objective value (when the primal variant is used).
- `DOBJ`: An estimate for the dual objective value (when the dual variant is used).
- `TIME`: Time spent since this instance of the simplex optimizer was invoked (in seconds).
- `TOTTIME`: Time spent since optimization started (in seconds).

13.3 Conic Optimization - Interior-point optimizer

For conic optimization problems only an interior-point type optimizer is available. The same optimizer is used for quadratic optimization problems which are internally reformulated to conic form.

13.3.1 The homogeneous primal-dual problem

The interior-point optimizer is an implementation of the so-called homogeneous and self-dual algorithm. For a detailed description of the algorithm, please see [ART03]. In order to keep our discussion simple we will assume that **MOSEK** solves a conic optimization problem of the form:

$$\begin{array}{lll} \text{minimize} & c^T x & \\ \text{subject to} & Ax & = \quad b, \\ & x \in \mathcal{K} & \end{array} \tag{13.6}$$

where K is a convex cone. The corresponding dual problem is

$$\begin{array}{lll} \text{maximize} & b^T y & \\ \text{subject to} & A^T y + s & = \quad c, \\ & s \in \mathcal{K}^* & \end{array} \tag{13.7}$$

where $\mathcal{K}^*$ is the dual cone of $\mathcal{K}$. See Sec. 12.2 for definitions.

Since it is not known beforehand whether problem (13.6) has an optimal solution, is primal infeasible or is dual infeasible, the optimization algorithm must deal with all three situations. This is the reason that **MOSEK** solves the so-called homogeneous model

$$\begin{array}{rcl} Ax - b\tau & = & 0, \\ A^T y + s - c\tau & = & 0, \\ -c^T x + b^T y - \kappa & = & 0, \\ x & \in & \mathcal{K}, \\ s & \in & \mathcal{K}^*, \\ \tau, \kappa & \geq & 0, \end{array} \tag{13.8}$$

where y and s correspond to the dual variables in (13.6), and τ and κ are two additional scalar variables. Note that the homogeneous model (13.8) always has a solution since

$$(x, y, s, \tau, \kappa) = (0, 0, 0, 0, 0)$$

is a solution, although not a very interesting one. Any solution

$$(x^*, y^*, s^*, \tau^*, \kappa^*)$$

to the homogeneous model (13.8) satisfies

$$(x^*)^T s^* + \tau^* \kappa^* = 0$$

i.e. complementarity. Observe that $x^* \in \mathcal{K}$ and $s^* \in \mathcal{K}^*$ implies

$$(x^*)^T s^* \geq 0$$

and therefore

$$\tau^* \kappa^* = 0.$$

since $\tau^*, \kappa^* \geq 0$. Hence, at least one of τ^* and κ^* is zero.

First, assume that $\tau^* > 0$ and hence $\kappa^* = 0$. It follows that

$$\begin{aligned} A\frac{x^*}{\tau^*} &= b, \\ A^T\frac{y^*}{\tau^*} + \frac{s^*}{\tau^*} &= c, \\ -c^T\frac{x^*}{\tau^*} + b^T\frac{y^*}{\tau^*} &= 0, \\ x^*/\tau^* &\in \mathcal{K}, \\ s^*/\tau^* &\in \mathcal{K}^*. \end{aligned}$$

This shows that $\frac{x^*}{\tau^*}$ is a primal optimal solution and $(\frac{y^*}{\tau^*}, \frac{s^*}{\tau^*})$ is a dual optimal solution; this is reported as the optimal interior-point solution since

$$(x, y, s) = \left(\frac{x^*}{\tau^*}, \frac{y^*}{\tau^*}, \frac{s^*}{\tau^*}\right)$$

is a primal-dual optimal solution.

On other hand, if $\kappa^* > 0$ then

$$\begin{aligned} Ax^* &= 0, \\ A^T y^* + s^* &= 0, \\ -c^T x^* + b^T y^* &= \kappa^*, \\ x^* &\in \mathcal{K}, \\ s^* &\in \mathcal{K}^*. \end{aligned}$$

This implies that at least one of

$$c^T x^* < 0 \tag{13.9}$$

or

$$b^T y^* > 0 \tag{13.10}$$

holds. If (13.9) is satisfied, then x^* is a certificate of dual infeasibility, whereas if (13.10) holds then y^* is a certificate of primal infeasibility.

In summary, by computing an appropriate solution to the homogeneous model, all information required for a solution to the original problem is obtained. A solution to the homogeneous model can be computed using a primal-dual interior-point algorithm [And09].

13.3.2 Interior-point Termination Criterion

Since computations are performed in finite precision, and for efficiency reasons, it is not possible to solve the homogeneous model exactly in general. Hence, an exact optimal solution or an exact infeasibility certificate cannot be computed and a reasonable termination criterion has to be employed.

In every iteration k of the interior-point algorithm a trial solution

$$(x^k, y^k, s^k, \tau^k, \kappa^k)$$

to the homogeneous model is generated, where

$$x^k \in \mathcal{K}, s^k \in \mathcal{K}^*, \tau^k, \kappa^k > 0.$$

Therefore, it is possible to compute the values:

$$\begin{array}{rcl}
\rho_p^k & = & \arg\min_\rho \left\{\rho \mid \left\| A\frac{x^k}{\tau^k} - b \right\|_\infty \leq \rho\varepsilon_p(1 + \|b\|_\infty)\right\}, \\
\rho_d^k & = & \arg\min_\rho \left\{\rho \mid \left\| A^T\frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c \right\|_\infty \leq \rho\varepsilon_d(1 + \|c\|_\infty)\right\}, \\
\rho_g^k & = & \arg\min_\rho \left\{\rho \mid \left(\frac{(x^k)^T s^k}{(\tau^k)^2}, |\frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k}|\right) \leq \rho\varepsilon_g \max\left(1, \frac{\min(|c^T x^k|, |b^T y^k|)}{\tau^k}\right)\right\}, \\
\rho_{pi}^k & = & \arg\min_\rho \left\{\rho \mid \left\| A^T y^k + s^k \right\|_\infty \leq \rho\varepsilon_i b^T y^k,\ b^T y^k > 0\right\} \text{ and} \\
\rho_{di}^k & = & \arg\min_\rho \left\{\rho \mid \left\| Ax^k \right\|_\infty \leq -\rho\varepsilon_i c^T x^k,\ c^T x^k < 0\right\}.
\end{array}$$

Note $\varepsilon_p, \varepsilon_d, \varepsilon_g$ and ε_i are nonnegative user specified tolerances.

Optimal Case

Observe ρ_p^k measures how far x^k/τ^k is from being a good approximate primal feasible solution. Indeed if $\rho_p^k \leq 1$, then

$$\left\| A\frac{x^k}{\tau^k} - b \right\|_\infty \leq \varepsilon_p(1 + \|b\|_\infty). \tag{13.11}$$

This shows the violations in the primal equality constraints for the solution x^k/τ^k is small compared to the size of b given ε_p is small.

Similarly, if $\rho_d^k \leq 1$, then $(y^k, s^k)/\tau^k$ is an approximate dual feasible solution. If in addition $\rho_g \leq 1$, then the solution $(x^k, y^k, s^k)/\tau^k$ is approximate optimal because the associated primal and dual objective values are almost identical.

In other words if $\max(\rho_p^k, \rho_d^k, \rho_g^k) \leq 1$, then

$$\frac{(x^k, y^k, s^k)}{\tau^k}$$

is an approximate optimal solution.

Dual Infeasibility Certificate

Next assume that $\rho_{di}^k \leq 1$ and hence

$$\left\| Ax^k \right\|_\infty \leq -\varepsilon_i c^T x^k \text{ and } -c^T x^k > 0$$

holds. Now in this case the problem is declared dual infeasible and x^k is reported as a certificate of dual infeasibility. The motivation for this stopping criterion is as follows. Let

$$\bar{x} := \frac{x^k}{-c^T x^k}$$

and it is easy to verify that

$$\|A\bar{x}\|_\infty \leq \varepsilon_i \text{ and } c^T\bar{x} = -1$$

which shows $\bar{x}$ is an approximate certificate of dual infeasibility, where ε_i controls the quality of the approximation.

Primal Infeasiblity Certificate

Next assume that $\rho_{pi}^k \leq 1$ and hence

$$\left\|A^T y^k + s^k\right\|_\infty \leq \varepsilon_i b^T y^k \text{ and } b^T y^k > 0$$

holds. Now in this case the problem is declared primal infeasible and (y^k, s^k) is reported as a certificate of primal infeasibility. The motivation for this stopping criterion is as follows. Let

$$\bar{y} := \frac{y^k}{b^T y^k} \text{ and } \bar{s} := \frac{s^k}{b^T y^k}$$

and it is easy to verify that

$$\left\|A^T \bar{y} + \bar{s}\right\|_\infty \leq \varepsilon_i \text{ and } b^T \bar{y} = 1$$

which shows (y^k, s^k) is an approximate certificate of dual infeasibility, where ε_i controls the quality of the approximation.

13.3.3 Adjusting optimality criteria

It is possible to adjust the tolerances ε_p, ε_d, ε_g and ε_i using parameters; see the next table for details. Note that although this section discusses the conic optimizer, if the problem was originally input as a quadratic or quadratically constrained optimization problem then the parameter names that apply are those from the third column (with infix `QO` instead of `CO`).

Table 13.2: Parameters employed in termination criterion

ToleranceParameter	Name (for conic problems)	Name (for quadratic problems)
ε_p	*MSK_DPAR_INTPNT_CO_TOL_PFEAS*	*MSK_DPAR_INTPNT_QO_TOL_PFEAS*
ε_d	*MSK_DPAR_INTPNT_CO_TOL_DFEAS*	*MSK_DPAR_INTPNT_QO_TOL_DFEAS*
ε_g	*MSK_DPAR_INTPNT_CO_TOL_REL_GAP*	*MSK_DPAR_INTPNT_QO_TOL_REL_GAP*
ε_i	*MSK_DPAR_INTPNT_CO_TOL_INFEAS*	*MSK_DPAR_INTPNT_QO_TOL_INFEAS*

The default values of the termination tolerances are chosen such that for a majority of problems appearing in practice it is not possible to achieve much better accuracy. Therefore, tightening the tolerances usually is not worthwhile. However, an inspection of (13.11) reveals that the quality of the solution depends on $\|b\|_\infty$ and $\|c\|_\infty$; the smaller the norms are, the better the solution accuracy.

The interior-point method as implemented by **MOSEK** will converge toward optimality and primal and dual feasibility at the same rate [And09]. This means that if the optimizer is stopped prematurely then it is very unlikely that either the primal or dual solution is feasible. Another consequence is that in most cases all the tolerances, ε_p, ε_d, ε_g and ε_i, have to be relaxed together to achieve an effect.

If the optimizer terminates without locating a solution that satisfies the termination criteria, for example because of a stall or other numerical issues, then it will check if the solution found up to that point satisfies the same criteria with all tolerances multiplied by the value of *MSK_DPAR_INTPNT_CO_TOL_NEAR_REL*. If this is the case, the solution is still declared as optimal.

To conclude the discussion in this section, relaxing the termination criterion is usually not worthwhile.

13.3.4 The Interior-point Log

Below is a typical log output from the interior-point optimizer:

```
Optimizer  - threads                : 20
Optimizer  - solved problem         : the primal
Optimizer  - Constraints            : 1
Optimizer  - Cones                  : 2
Optimizer  - Scalar variables       : 6                 conic                  : 6
Optimizer  - Semi-definite variables: 0                 scalarized             : 0
Factor     - setup time             : 0.00              dense det. time        : 0.00
```

(continues on next page)

(continued from previous page)

```
Factor     - ML order time          : 0.00              GP order time          : 0.00
Factor     - nonzeros before factor : 1                 after factor           : 1
Factor     - dense dim.             : 0                 flops                  : 1.
↪70e+01
ITE PFEAS    DFEAS    GFEAS    PRSTATUS   POBJ              DOBJ              MU
↪ TIME
0   1.0e+00  2.9e-01  3.4e+00  0.00e+00   2.414213562e+00   0.000000000e+00   1.0e+00
↪ 0.01
1   2.7e-01  7.9e-02  2.2e+00  8.83e-01   6.969257574e-01   -9.685901771e-03  2.7e-01
↪ 0.01
2   6.5e-02  1.9e-02  1.2e+00  1.16e+00   7.606090061e-01   6.046141322e-01   6.5e-02
↪ 0.01
3   1.7e-03  5.0e-04  2.2e-01  1.12e+00   7.084385672e-01   7.045122560e-01   1.7e-03
↪ 0.01
4   1.4e-08  4.2e-09  4.9e-08  1.00e+00   7.071067941e-01   7.071067599e-01   1.4e-08
↪ 0.01
```

The first line displays the number of threads used by the optimizer and the second line indicates if the optimizer chose to solve the primal or dual problem (see *MSK_IPAR_INTPNT_SOLVE_FORM*). The next lines display the problem dimensions as seen by the optimizer, and the `Factor...` lines show various statistics. This is followed by the iteration log.

Using the same notation as in Sec. 13.3.1 the columns of the iteration log have the following meaning:

- `ITE`: Iteration index k.
- `PFEAS`: $\left\|Ax^k - b\tau^k\right\|_\infty$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- `DFEAS`: $\left\|A^T y^k + s^k - c\tau^k\right\|_\infty$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- `GFEAS`: $|-c^T x^k + b^T y^k - \kappa^k|$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- `PRSTATUS`: This number converges to 1 if the problem has an optimal solution whereas it converges to -1 if that is not the case.
- `POBJ`: $c^T x^k/\tau^k$. An estimate for the primal objective value.
- `DOBJ`: $b^T y^k/\tau^k$. An estimate for the dual objective value.
- `MU`: $\frac{(x^k)^T s^k + \tau^k \kappa^k}{n+1}$. The numbers in this column should always converge to zero.
- `TIME`: Time spent since the optimization started (in seconds).

13.4 The Optimizer for Mixed-Integer Problems

Solving optimization problems where one or more of the variables are constrained to be integer valued is called Mixed-Integer Optimization (MIO). For an introduction to model building with integer variables, the reader is recommended to consult the **MOSEK** Modeling Cookbook, and for further reading we highlight textbooks such as [Wol98] or [CCornuejolsZ14].

MOSEK can perform mixed-integer

- linear (MILO),
- quadratic (MIQO) and quadratically constrained (MIQCQO), and
- conic (MICO)

optimization, except for mixed-integer semidefinite problems.

By default the mixed-integer optimizer is run-to-run deterministic. This means that if a problem is solved twice on the same computer with identical parameter settings and no time limit, then the obtained solutions will be identical. The mixed-integer optimizer is parallelized, i.e., it can exploit multiple cores during the optimization.

In practice, it often happens that the integer variables in MIO problems are actuall binary variables, taking values in $\{0, 1\}$, leading to Mixed- or pure binary problems. In the general setting however, an integer variable may have arbitrary lower and upper bounds.

13.4.1 Branch-and-Bound

In order to succeed in solving mixed-integer problems, it can be useful to have a basic understanding of the underlying solution algorithms. The most important concept in this regard is arguably the so-called Branch-and-Bound algorithm, employed also by **MOSEK**. The more experienced reader may skip this section and advance directly to Sec. 13.4.2.

In order to comprehend Branch-and-Bound, the concept of a *relaxation* is important. Consider for example a mixed-integer linear optimization problem of minimization type

$$\begin{array}{rcllll} z^* & = & \text{minimize} & c^T x & & \\ & & \text{subject to} & Ax & = & b \\ & & & x \geq 0 & & \\ & & & x_j \in \mathbb{Z}, & \forall j \in \mathcal{J}. & \end{array} \tag{13.12}$$

It has the continuous relaxation

$$\begin{array}{rcllll} \underline{z} & = & \text{minimize} & c^T x & & \\ & & \text{subject to} & Ax & = & b \\ & & & x \geq 0, & & \end{array} \tag{13.13}$$

simply obtained by ignoring the integrality restrictions. The first step in Branch-and-Bound is to solve this so-called *root* relaxation, which is a continuous optimization problem. Since (13.13) is less constrained than (13.12), one certainly gets

$$\underline{z} \leq z^*,$$

and $\underline{z}$ is therefore called the *objective bound*: it bounds the optimal objective value from below.

After the solution of the root relaxation, in the most likely outcome there will be one or more integer constrained variables with fractional values, i.e., violating the integrality constraints. Branch-and-Bound now takes such a variable, $x_j = f_j \in \mathbb{R}\backslash\mathbb{Z}$ with $j \in \mathcal{J}$, say, and creates two branches leading to relaxations with the additional constraint $x_j \leq \lfloor f_j \rfloor$ or $x_j \geq \lceil f_j \rceil$, respectively. The intuitive idea here is to exclude the undesired fractional value from the outcomes in the two created branches. If the integer variable was actually a binary variable, branching would lead to fixing its value to 0 in one branch, and to 1 in the other.

The Branch-and-Bound process continues in this way and successively solves relaxations and creates branches to refined relaxations. Whenever the solution $\hat{x}$ to some relaxation does not violate any integrality constraints, it is feasible to (13.12) and is called an *integer feasible solution*. There is no guarantee though that it is also optimal, its solution value $\bar{z} := c^T \hat{x}$ is only an upper bound on the optimal objective value,

$$z^* \leq \bar{z}.$$

By the successive addition of constraints in the created branches, the objective bound $\underline{z}$ (now defined as the minimum over all solution values of so far solved relaxations) can only increase during the algorithm. At the same time, the upper bound $\bar{z}$ (the solution value of the best integer feasible solution encountered so far, also called *incumbent solution*) can only decrease during the algorithm. Since at any time we also have

$$\underline{z} \leq z^* \leq \bar{z},$$

objective bound and incumbent solution value are encapsulating the optimal objective value, eventually converging to it.

The Branch-and-Bound scheme can be depicted by means of a tree, where branches and relaxations correspond to edges and nodes. Figure Fig. 13.1 shows an example of such a tree. The strength of Branch-and-Bound is its ability to prune nodes in this tree, meaning that no new child nodes will be created. Pruning can occur in several cases:

- A relaxation leads to an integer feasible solution $\hat{x}$. In this case we may update the incumbent and its solution value $\bar{z}$, but no new branches need to be created.
- A relaxation is infeasible. The subtree rooted at this node cannot contain any feasible relaxation, so it can be discarded.
- A relaxation has a solution value that exceeds $\bar{z}$. The subtree rooted at this node cannot contain any integer feasible solution with a solution value better than the incumbent we already have, so it can be discarded.

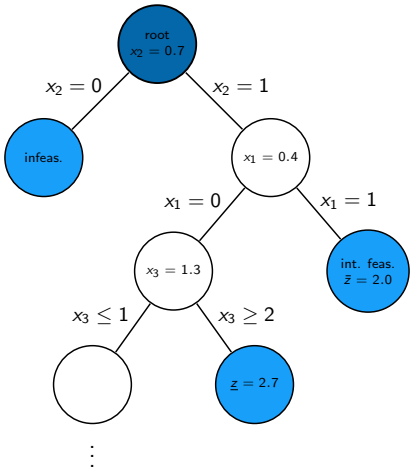


Fig. 13.1: An examplary Branch-and-Bound tree. Pruned nodes are shown in light blue.

Having objective bound and incumbent solution value is a quite fundamental property of Branch-and-Bound, and helps to asses solution quality and control termination of the algorithm, as we detail in the next section. Note that the above explanation is coined for minimization problems, but the Branch-and-bound scheme has a straightforward extension to maximization problems.

13.4.2 Solution quality and termination criteria

The issue of terminating the mixed-integer optimizer is rather delicate. Mixed-integer optimization is generally much harder than continuous optimization; in fact, solving continuous sub-problems is just one component of a mixed-integer optimizer. Despite the ability to prune nodes in the tree, the computational effort required to solve mixed-integer problems grows exponentially with the size of the problem in a worst-case scenario (solving mixed-integer problems is NP-hard). For instance, a problem with n binary variables, may require the solution of 2^n relaxations. The value of 2^n is huge even for moderate values of n. In practice it is often advisable to accept near-optimal or appproximate solutions in order to counteract this complexity burden. The user has numerous possibilities of influencing optimizer termination with various parameters, in particular related to solution quality, and the most important ones are highlighted here.

Solution quality in terms of optimality

In order to assess the quality of any incumbent solution in terms of its objective value, one may check the *optimality gap*, defined as

$$\epsilon = |(\text{incumbent solution value}) - (\text{objective bound})| = |\bar{z} - \underline{z}|.$$

It measures how much the objectives of the incumbent and the optimal solution can deviate in the worst case. Often it is more meaningful to look at the *relative optimality gap*

$$\epsilon_{\text{rel}} = \frac{|\bar{z} - \underline{z}|}{\max(\delta_1, |\bar{z}|)}.$$

This is essentially the above *absolute* optimality gap normalized against the magnitude of the incumbent solution value; the purpose of the (small) constant δ_1 is to avoid overweighing incumbent solution values that are very close to zero. The relative optimality gap can thus be interpreted as answering the question: *"Within what fraction of the optimal solution is the incumbent solution in the worst case?"*

Absolute and relative optimality gaps provide useful means to define termination criteria for the mixed-integer optimizer in **MOSEK**. The idea is to terminate the optimization process as soon as the quality of the incumbent solution, measured in absolute or relative gap, is good enough. In fact, whenever an incumbent solution is located, the criterion

$$\epsilon \leq \delta_2 \text{ or } \epsilon_{\text{rel}} \leq \delta_3$$

is checked. If satisfied, i.e., if either absolute or relative optimality gap are below the thresholds δ_2 or δ_3 (see Table 13.3), the optimizer terminates and reports the incumbent as an optimal solution. The optimality gaps at termination can always be retrieved through the information items *MSK_DINF_MIO_OBJ_ABS_GAP* and *MSK_DINF_MIO_OBJ_REL_GAP*.

The tolerances discussed above can be adjusted using suitable parameters, see Table 13.3. By default, the optimality parameters δ_2 and δ_3 are quite small, i.e., restrictive. These default values for the absolute and relative gap amount to solving any instance to (almost) optimality: the incumbent is required to be within at most a tiny percentage of the optimal solution. As anticipated, this is not tractable in many practical situations, and one should resort to finding near-optimal solutions quickly rather than insisting on finding the optimal one. It may happen, for example, that an optimal or close-to-optimal solution is found very early by the optimizer, but it spends a huge amount of further computational time for branching, trying to increase $\underline{z}$ that last missing bit: a typical situation that practioneers would want to avoid. The concept of optimality gaps is fundamental for controlling solution quality when resorting to near-optimal solutions.

MIO performance tweaks: termination criteria

One of the first things to do in order to cut down excessive solution time is to increase the relative gap tolerance *MSK_DPAR_MIO_TOL_REL_GAP* to some non-default value, so as to not insist on finding optimal solutions. Typical values could be $0.01, 0.05$ or 0.1, guaranteeing that the delivered solutions lie within $1\%, 5\%$ or 10% of the optimum. Increasing the tolerance will lead to less computational time spent by the optimizer.

Solution quality in terms of feasibility

For an optimizer relying on floating-point arithmetic like the mixed-integer optimizer in **MOSEK**, it may be hard to achieve exact integrality of the solution values of integer variables in most cases, and it makes sense to numerically relax this constraint. Any candidate solution $\hat{x}$ is accepted as integer feasible if the criterion

$$\min(\hat{x_j} - \lfloor \hat{x_j} \rfloor, \lceil \hat{x_j} \rceil - \hat{x_j}) \leq \delta_4 \quad \forall j \in \mathcal{J}$$

is satisfied, meaning that $\hat{x_j}$ is at most δ_4 away from the nearest integer. As above, δ_4 can be adjusted using a parameter, see Table 13.3, and impacts the quality of the acieved solution in terms of integer feasibility. By influencing what solution may be accepted as imcumbent, it can also have an impact on the termination of the optimizer.

MIO performance tweaks: feasibility criteria

Whether increasing the integer feasibility tolerance *MSK_DPAR_MIO_TOL_ABS_RELAX_INT* leads to less solution time is highly problem dependent. Intuitively, the optimizer is more flexible in finding new incumbent soutions so as to improve $\bar{z}$. But this effect has do be examined with care on indivuidual instances: it may worsen solution quality with no effect at all on the solution time. It may in some cases even lead to contrary effects on the solution time.

Table 13.3: Tolerances for the mixed-integer optimizer.

Tolerance	Parameter name	Default value
δ_1	*MSK_DPAR_MIO_REL_GAP_CONST*	1.0e-10
δ_2	*MSK_DPAR_MIO_TOL_ABS_GAP*	0.0
δ_3	*MSK_DPAR_MIO_TOL_REL_GAP*	1.0e-4
δ_4	*MSK_DPAR_MIO_TOL_ABS_RELAX_INT*	1.0e-5

Further controlling optimizer termination

There are more ways to limit the computational effort employed by the mixed-integer optimizer by simply limiting the number of explored branches, solved relaxations or updates of the incumbent solution. When any of the imposed limits is hit, the optimizer terminates and the incumbent solution may be retrieved. See Table 13.4 for a list of corresponding parameters. In contrast to the parameters discussed in Sec. 13.4.2, interfering with these does not maintain any guarantees in terms of solution quality.

Table 13.4: Other parameters affecting the integer optimizer termination criterion.

Parameter name	Explanation
MSK_IPAR_MIO_MAX_NUM_BRANCHES	Maximum number of branches allowed.
MSK_IPAR_MIO_MAX_NUM_RELAXS	Maximum number of relaxations allowed.
MSK_IPAR_MIO_MAX_NUM_SOLUTIONS	Maximum number of feasible integer solutions allowed.

13.4.3 The Mixed-Integer Log

The Branch-and-Bound scheme from Sec. 13.4.1 is only the basic skeleton of the mixed-integer optimizer in **MOSEK**, and several components are built on top of that in order to enhance its functionality and increase its speed. A mixed-integer optimizer is sometimes referred to as a *"giant bag of tricks"*, and it would be impossible to describe all of these tricks here. Yet, some of the additional components are worth mentioning. They can be influenced by various user parameters, and although the default values of these parameters are optimized to work well on average mixed-integer problems, it may pay off to adjust them for an individual problem, or a specific problem class. The mixed-integer log can give insights on which parameters might be worth an adjustment. Below is a typical log output:

```
Presolve started.
Presolve terminated. Time = 0.23, probing time =  0.09
Presolved problem: 1176 variables, 1344 constraints, 4968 non-zeros
Presolved problem: 328 general integer, 392 binary, 456 continuous
Clique table size: 55
Symmetry factor  : 0.79 (detection time = 0.01)
Removed blocks   : 2
BRANCHES RELAXS   ACT_NDS  DEPTH    BEST_INT_OBJ         BEST_RELAX_OBJ       REL_GAP(
↪%)  TIME
```

(continues on next page)

(continued from previous page)

```
0        0        1        0        8.3888091139e+07     NA                   NA       ␣
↪    0.2
0        1        1        0        8.3888091139e+07     2.5492512136e+07     69.61    ␣
↪    0.3
0        1        1        0        3.1273162420e+07     2.5492512136e+07     18.48    ␣
↪    0.4
0        1        1        0        2.6047699632e+07     2.5492512136e+07     2.13     ␣
↪    0.4
Rooot cut generation started.
0        1        1        0        2.6047699632e+07     2.5492512136e+07     2.13     ␣
↪    0.4
0        2        1        0        2.6047699632e+07     2.5589986247e+07     1.76     ␣
↪    0.4
Rooot cut generation terminated. Time = 0.05
0        4        1        0        2.5990071367e+07     2.5662741991e+07     1.26     ␣
↪    0.5
0        8        1        0        2.5971002767e+07     2.5662741991e+07     1.19     ␣
↪    0.6
0        11       1        0        2.5925040617e+07     2.5662741991e+07     1.01     ␣
↪    0.6
0        12       1        0        2.5915504014e+07     2.5662741991e+07     0.98     ␣
↪    0.6
2        23       1        0        2.5915504014e+07     2.5662741991e+07     0.98     ␣
↪    0.7
14       35       1        0        2.5915504014e+07     2.5662741991e+07     0.98     ␣
↪    0.7

[ ... ]

Objective of best integer solution : 2.578282162804e+07
Best objective bound               : 2.569877601306e+07
Construct solution objective       : Not employed
User objective cut value           : Not employed
Number of cuts generated           : 192
  Number of Gomory cuts            : 52
  Number of CMIR cuts              : 137
  Number of clique cuts            : 3
Number of branches                 : 29252
Number of relaxations solved       : 31280
Number of interior point iterations: 16
Number of simplex iterations       : 105440
Time spend presolving the root     : 0.23
Time spend optimizing the root     : 0.07
Mixed integer optimizer terminated. Time: 6.96
```

The main part here is the iteration log, a progressing series of similar rows reflecting the progress made during the Branch-and-bound process. The columns have the following meanings:

- `BRANCHES`: Number of branches / nodes generated.
- `RELAXS`: Number of relaxations solved.
- `ACT_NDS`: Number of active / non-processed nodes.
- `DEPTH`: Depth of the last solved node.
- `BEST_INT_OBJ`: The incumbent solution / best integer objective value, $\bar{z}$.
- `BEST_RELAX_OBJ`: The objective bound, $\underline{z}$.

- `REL_GAP(%)`: Relative optimality gap, $100\% \cdot \epsilon_{\text{rel}}$
- `TIME`: Time (in seconds) from the start of optimization.

Also a short solution summary with several statistics is printed. When the solution time for a mixed-integer problem has to be cut down, the log can help to understand where time is spent and what might be improved. We go into some more detail about some further items in the mixed-integer log giving hints about individual components of the optimizer. Alternatively, most of these items can also be retrieved as information items, see Sec. 7.6.

Presolve

Similar to the case of continuous problems, see Sec. 13.1, the mixed-integer optimizer applies various presolve reductions before the actual Branch-and-bound is initiated. The first lines of the mixed-integer log contain a summary of the presolve process, including the time spent therein (`Presolve terminated. Time = 0.23...`). Just as in the continuous case, the use of presolve can be controlled with the parameter *MSK_IPAR_PRESOLVE_USE*. If presolve time seems excessive, instead of switching it off completely one may also try to reduce the time spent in one or more of its individual components. On some models it can also make sense to increase the use of a certain presolve technique. Table Table 13.5 lists some of these with their respective parameters.

Table 13.5: Parameters affecting presolve

Parameter name	Explanation	Possible reference in log
MSK_IPAR_MIO_PROBING_LEVE	Probing aggressivity level.	`... probing time = 0.09`
MSK_IPAR_MIO_SYMMETRY_LEV	Symmetry detection aggressivity level.	`Symmetry factor : 0.79 (detection time = 0.01)`
*MSK_IPAR_MIO_INDEPENDENT_*	Block structure detection level, see Sec. 13.4.3.	`Removed blocks : 2`
*MSK_DPAR_MIO_CLIQUE_TABLE_*	Maximum size of the clqiue table.	`Clique table size: 55`
MSK_IPAR_MIO_PRESOLVE_AGG	Should variable agggregation be enabled?	–

Primal Heuristics

It might happen that the value in the colum `BEST_INT_OBJ` stalls over a long period of log lines, an indication that the optimizer has a hard time improving the incumbent solution, i.e., $\bar{z}$. Solving relaxations in the tree to an integer feasible solution $\hat{x}$ is not the only way to find new incumbent solutions. There is a variety of procedures that, given a mixed-integer problem in a generic form like (13.12), attempt to produce integer feasible solutions in an ad-hoc way. These procedures are called Primal Heuristics, and several of them are implemented in **MOSEK**. For example, whenever a relaxation leads to a fractional solution, one may round the solution values of the integer variables, in various ways, and hope that the outcome is still feasible to the remaining constraints. Primal heuristics are mostly employed while processing the root node, but play a role throughout the whole solution process. The goal of a primal heuristic is to improve the incumbent solution and thus the bound $\bar{z}$, and this can of course affect the quality of the solution that is returned after termination of the optimizer. The user parameters affecting primal heuristics are listed in Table 13.6.

MIO performance tweaks: primal heuristics

- If the mixed-integer optimizer struggles to improve the incumbent solution `BEST_INT_OBJ`, it can be helpful to intensify the use of primal heuristics.
  - Set parameters related to primal heuristics to more aggressive values than the default ones, so that more effort is spent in this component. A List of the respective parameters can be found in Table 13.6. In particular, if the optimizer has difficulties finding any integer feasible solution at all, indicated by `NA` in the column `BEST_INT_OBJ` in the mixed-integer log, one may try to activate a construction heuristic like the Feasibility Pump with *MSK_IPAR_MIO_FEASPUMP_LEVEL*.

- Specify a good initial solution: In many cases a good feasible solution is either known or easily computed using problem-specific knowledge that the optimizer does not have. If so, it is usually worthwhile to use this as a starting point for the mixed-integer optimizer. See also the parameter *MSK_IPAR_MIO_CONSTRUCT_SOL*, and Section Sec. 6.8.2.
- For feasibility problems, i.e., problems having a constant objective, the goal is to find a single integer feasible solution, and this can be hard by itself on some instances. Try setting the objective to something meaningful anyway, even if the underlying application does not require this. After all, the feasible set is not changed, but the optimizer might benefit from being able to pursue a concrete goal.

- In rare cases it may also happen that the optimizer spends an excessive amount of time on primal heuristics without drawing any benefit from it, and one may try to limit their use with the respective parameters.

Table 13.6: Parameters affecting primal heuristics

Parameter name	Explanation
MSK_IPAR_MIO_HEURISTIC_LEVEL	Primal heuristics aggressivity level.
MSK_IPAR_MIO_RINS_MAX_NODES	Maximum number of nodes allowed in the RINS heuristic.
MSK_IPAR_MIO_RENS_MAX_NODES	Maximum number of nodes allowed in the RENS heuristic.
MSK_IPAR_MIO_CROSSOVER_MAX_NODES	Maximum number of nodes allowed in the Crossover heuristic.
MSK_IPAR_MIO_OPT_FACE_MAX_NODES	Maximum number of nodes allowed in the optimal face heuristic.
MSK_IPAR_MIO_FEASPUMP_LEVEL	Way of using the Feasibility Pump heuristic.

Cutting Planes

It might as well happen that the value in the colum `BEST_RELAX_OBJ` stalls over a long period of log lines, an indication that the optimizer has a struggles to improve the objective bound $\underline{z}$. A component of the optimizer designed to act on the objective bound is given by Cutting planes, also called cuts or valid inequalities. Cuts do not remove any integer feasible solutions from the feasible set of the mixed-integer problem (13.12). They may, however, remove solutions from the feasible set of the relaxation (13.13), ideally making it a *stronger* relaxation with better objective bound.

As an example, take the constraints

$$2x_1 + 3x_2 + x_3 \leq 4, \quad x_1, x_2 \in \{0, 1\}, \quad x_3 \geq 0. \tag{13.14}$$

One may realize that there cannot be a feasible solution in which both binary variables take on a value of 1. So certainly

$$x_1 + x_2 \leq 1 \tag{13.15}$$

is a valid inequality (there is no integer solution satisfying (13.14), but violating (13.15)). The latter does cut off a portion of the feasible region of the continuous relaxation of (13.14) though, obtained by replacing $x_1, x_2 \in \{0, 1\}$ with $x_1, x_2 \in [0, 1]$. For example, the fractional point $(x_1, x_2, x_3) = (0.5, 1, 0)$ is feasible to the relaxation, but violates the cut (13.15).

There are many classes of general-purpose cuttting planes that may be generated for a mixed-integer problem in a generic form like (13.12), and **MOSEK**'s mixed-integer optimizer supports several of them. For instance, the above is an example of a so-called clique cut. The most effort on generating cutting planes is spent after the solution of the root relaxation; the beginning and the end of root cut generation is highlighted in the log, and the number of log lines in between reflects to the computational effort spent here. Also the solution summary at the end of the log highlights for each cut class the number of generated cuts. Cuts can also be generated later on in the tree, which is why we also use the term Branch-and-cut, an extension of the basic Branch-and-bound scheme. Cuts aim at improving the objective bound $\underline{z}$ and can thus have significant impact on the solution time. The user parameters affecting cut generation can be seen in Table 13.7.

MIO performance tweaks: cutting planes

- If the mixed-integer optimizer struggles to improve the objective bound `BEST_RELAX_OBJ`, it can be helpful to intensify the use of cutting planes.
  - Some types of cutting planes are not activated by default, but doing so may help to improve the objective bound.
  - The parameters *MSK_DPAR_MIO_TOL_REL_DUAL_BOUND_IMPROVEMENT* and *MSK_IPAR_MIO_CUT_SELECTION_LEVEL* determine how aggressively cuts will be generated and selected.
  - If some valid inequalities can be deduced from problem-specific knowledge that the optimizer does not have, it may be helpful to add these to the problem formulation as constraints. This has to be done with care, since there is a tradeoff between the benefit obtained from an improved objective boud, and the amount of additional constraints that make the relaxations larger.
- In rare cases it may also be observed that the optimizer spends an excessive effort on cutting planes, and one may limit their use with *MSK_IPAR_MIO_MAX_NUM_ROOT_CUT_ROUNDS*, or by disabling a certain type of cutting planes.

Table 13.7: Parameters affecting cutting planes

Parameter name	Explanation
MSK_IPAR_MIO_CUT_CLIQUE	Should clique cuts be enabled?
MSK_IPAR_MIO_CUT_CMIR	Should mixed-integer rounding cuts be enabled?
MSK_IPAR_MIO_CUT_GMI	Should GMI cuts be enabled?
MSK_IPAR_MIO_CUT_IMPLIED_BOUND	Should implied bound cuts be enabled?
MSK_IPAR_MIO_CUT_KNAPSACK_COVER	Should knapsack cover cuts be enabled?
MSK_IPAR_MIO_CUT_LIPRO	Should lift-and-project cuts be enabled?
MSK_IPAR_MIO_CUT_SELECTION_LEVEL	Cut selection aggressivity level.
MSK_IPAR_MIO_MAX_NUM_ROOT_CUT_ROUN	Maximum number of root cut rounds.
MSK_DPAR_MIO_TOL_REL_DUAL_BOUND_IM	Minimum required objective bound improvement during root cut generation.

Restarts

The mixed-integer optimizer employs so-called restarts, i.e., if the progress made while exploring the tree is deemed unsufficient, it might decide to restart the solution process from scratch, possibly making use of the information collected so far. When a restart happens, this is displayed in the log:

```
[ ... ]

1948     4664     699      36       NA                   1.1800000000e+02     NA      ␣
↪    7.2
1970     4693     705      50       NA                   1.1800000000e+02     NA      ␣
↪    7.2

Performed MIP restart 1.
Presolve started.
Presolve terminated. Time = 0.01, probing time =  0.00
Presolved problem: 523 variables, 765 constraints, 3390 non-zeros
Presolved problem: 0 general integer, 404 binary, 119 continuous
Clique table size: 143
BRANCHES RELAXS   ACT_NDS  DEPTH    BEST_INT_OBJ         BEST_RELAX_OBJ       REL_GAP(
↪%)  TIME
1988     4729     1        0        NA                   1.1800000000e+02     NA      ␣
↪    7.3
1988     4730     1        0        4.0000000000e+01     1.1800000000e+02     195.00  ␣
```

(continues on next page)

(continued from previous page)

```
↪    7.3

[ ... ]
```

Restarts tend to be useful especially for hard models. However, in individual cases the optimizer may decide to perform a restart while it would have been better to continue exploring the tree. Their use can be controlled with the parameter *MSK_IPAR_MIO_MAX_NUM_RESTARTS*.

Block decomposition

Sometimes the optimizer faces a model that actually represents two or more completely independent subproblems. For a linear problem such as (13.13), this means that the constraint matrix A is a block-diagonal. Block-diagonal structure can occur after **MOSEK** applies some presolve reductions, e.g., a variable is fixed that was the only variable connecting two otherwise independent subproblems. Or, more rarely, the original model provided by the user is already block-diagonal.

In principle, solving such blocks independently is easier than letting the optimizer work on the single, large model, and **MOSEK** thus tries to exploit this structure. Some blocks may be completely solved and removed from the model during presolve, which can be seen by a line at the end of the presolve summary, see also Sec. 13.4.3. If after presolve there are still independent blocks, **MOSEK** can apply a dedicated algorithm to solve them independently while periodically combining their individual solution statusses (such as incumbent solutions and objective bounds) to the solution status of the original model. Just like the removal of blocks during presolve, the application of this latter strategy is indicated in the log:

```
[ ... ]

15       38       1        0        4.1759800000e+05     3.8354200000e+05     8.16    ␣
↪    0.9
Root cut generation started.
15       38       1        0        4.1759800000e+05     3.8354200000e+05     8.16    ␣
↪    1.1
Root cut generation terminated. Time = 0.11
15       40       1        0        4.1645600000e+05     3.8934425000e+05     6.51    ␣
↪    2.0
15       41       1        0        4.1622400000e+05     3.8934425000e+05     6.46    ␣
↪    2.0
23       52       1        0        4.1622400000e+05     3.8934425000e+05     6.46    ␣
↪    2.0
Decomposition solver started with 5 independent blocks.
532      425      5        118      4.1592600000e+05     3.8935275000e+05     6.39    ␣
↪    4.5
1858     11911    815      286      4.1007800000e+05     3.8946400000e+05     5.03    ␣
↪    11.8

[ ... ]
```

How block-diagonal structure is detected and handled by the optimizer can be controlled with the parameter *MSK_IPAR_MIO_INDEPENDENT_BLOCK_LEVEL*.

13.4.4 Mixed-Integer Nonlinear Optimization

Due to the involved non-linearities, MI(QC)QO or MICO problems are on average harder than MILO problems of comparable size. Yet, the Branch-and-Bound scheme can be applied to these probelm classes in a straightforward manner. The relaxations have to be solved as conic problems with the interior point algorithm in that case, see Sec. 13.3, opposed to MILO where it is often beneficial to solve relaxations with the dual simplex method, see Sec. 13.2.3. There is another solution approach for these types of problems implemented in **MOSEK**, namely the Outer-Approximation algorithm, making use of dynamically refined linear approximations of the non-linearities.

MICO performance tweaks: choice of algorithm

Whether conic Branch-and-Bound or Outer-Approximation is applied to a mixed-integer conic problem can be set with *MSK_IPAR_MIO_CONIC_OUTER_APPROXIMATION*. The best value for this option is highly problem dependent.

MI(QC)QO

MOSEK is specialized in solving linear and conic optimization problems, both with or without mixed-integer variables. Just like for continuous problems, mixed-integer quadratic problems are converted internally to conic form, see Sec. 12.4.1

Contrary to the continuous case, **MOSEK** can solve certain mixed-integer quadratic problems where one or more of the involved matrices are not positive semidefinite, so-called non-convex MI(QC)QO problems. These are automatically reformulated to an equivalent convex MI(QC)QO problem, provided that such a reformulation is possible on the given instance (otherwiese **MOSEK** will reject the problem and issue an error message). The concept of reformulations can also affect the solution times of MI(QC)QO problems.

MI(QC)QO performance tweaks: applying a reformulation method

There are several reformulation methods for MI(QC)QO problems, available through the parameter *MSK_IPAR_MIO_QCQO_REFORMULATION_METHOD*. The chosen method can have significant impact on the mixed-integer optimizer's speed on such problems, both convex and non-convex. The best value for this option is highly problem dependent.

13.4.5 Disjunctive constraints

Problems with disjunctive constraints (DJC) see Sec. 6.9 are typically reformulated to mixed-integer problems, and even if this is not the case they are solved with an algorithm that is based on the mixed-integer optimizer. In **MOSEK**, these problems thus fall into the realm of MIO. In particular, **MOSEK** automatically attempts to replace any DJC by so called big-M constraints, potentially after transforming it to several, less complicated DJCs. As an example, take the DJC

$$[z = 0] \vee [z = 1, x_1 + x_2 \geq 1000],$$

where $z \in \{0, 1\}$ and $x_1, x_2 \in [0, 750]$. This is an example of a DJC formulation of a so-called indicator constraint. A big-M reformulation is given by

$$x_1 + x_2 \geq 1000 - M \cdot (1 - z),$$

where $M > 0$ is a large constant. The practical difficulty of these constructs is that M should always be sufficiently large, but ideally not larger. Too large values for M can be harmful for the mixed-integer optimizer. During presolve, and taking into account the bounds of the involved variables, **MOSEK** automatically reformulates DJCs to big-M constraints if the required M values do not exceed the parameter *MSK_DPAR_MIO_DJC_MAX_BIGM*. From a performance point-of-view, all DJCs would ideally be linearized to big-Ms after presolve without changing this parameter's default value of 1.0e6. Whether or not this is the case can be seen by retrieving the information item *MSK_IINF_MIO_PRESOLVED_NUMDJC*, or by a line in the mixed-integer optimizer's log as in the example below. Both state the number of remaining disjunctions after presolve.

```
Presolved problem: 305 variables, 204 constraints, 708 non-zeros
Presolved problem: 0 general integer, 100 binary, 205 continuous
Presolved problem: 100 disjunctions
Clique table size: 0
BRANCHES RELAXS   ACT_NDS  DEPTH    BEST_INT_OBJ         BEST_RELAX_OBJ       REL_GAP(
↪%)  TIME
0        1        1        0        NA                   0.0000000000e+00     NA      ␣
↪    0.0
0        1        1        0        5.0574653969e+05     0.0000000000e+00     100.00  ␣
↪    0.0

[ ... ]
```

DJC performance tweaks: managing variable bounds

- Always specify the tightest known bounds on the variables of any problem with DJCs, even if they seem trivial from the user-perspective. The mixed-integer optimizer can only benefit from these when reformulating DJCs and thus gain performance; even if bounds don't help with reformulations, it is very unlikely that they hurt the optimizer.
- Increasing *MSK_DPAR_MIO_DJC_MAX_BIGM* can lead to more DJC reformulations and thus increase optimizer speed, but it may in turn hurt numerical solution quality and has to be examined with care. The other way round, on numerically challenging instances with DJCs, decreasing *MSK_DPAR_MIO_DJC_MAX_BIGM* may lead to numerically more robust solutions.

13.4.6 Randomization

A mixed-integer optimizer is usually prone to performance variability, meaning that a small change in either

- problem data, or
- computer hardware, or
- algorithmic parameters

can lead to significant changes in solution time, due to different solution paths in the Branch-and-cut tree. In extreme cases the exact same problem can vary from being solvable in less than a second to seemingly unsolvable in any reasonable amount of time on a different computer.

One practical implication of this is that one should ideally verify whether a seemingly beneficial set of parameters, established experimentally on a single problem, is still beneficial (on average) on a larger set of problems from the same problem class. This protects against making parameter changes that had positive effects only due to random effects on that single problem.

In the absence of a large set of test problems, one may also change the random seed of the optimizer to a series of different values in order to hedge against drawing such wrong conclusions regarding parameters. The random seed, accessible through *MSK_IPAR_MIO_SEED*, impacts for example random tie-breaking in many of the mixed-integer optimizer's components. Changing the random seed can be combined with a permutation of the problem data to further incite randomness, accessible through the parameter *MSK_IPAR_MIO_DATA_PERMUTATION_METHOD*.

13.4.7 Further performance tweaks

In addition to what was mentioned previously, there may be other ways to speed up the solution of a given mixed-integer problem. For example, there are further user parameters affecting some algorithmic settings in the mixed-integer optimizer. As mentioned above, default parameter values are optimized to work well on average, but on individual problems they may be adjusted.

MIO performance tweaks: miscellaneous

- While exploring the tree, the optimizer applies certain strategies to decide which fractional variable to branch on, see Sec. 13.4.1. The chosen strategy can have a big impact on performance, and may be controlled with *MSK_IPAR_MIO_VAR_SELECTION*.
- Similarly, the strategy to chose the next node to explore in the tree is controlled with *MSK_IPAR_MIO_NODE_SELECTION*.
- The optimizer employs specialized techniques to learn from infeasible nodes and use that knowledge to avoid creating similar nodes in other parts of the tree. The effort spent here can be influenced with *MSK_IPAR_MIO_DUAL_RAY_ANALYSIS_LEVEL* and *MSK_IPAR_MIO_CONFLICT_ANALYSIS_LEVEL*.
- When relaxations in the tree are linear optimization problems (e.g., in MILO or when solving MICO probelms with the Outer-Approximation method), it is usually best to employ the dual simplex method for their solution. In rare cases the primal simplex method may actually be the better choice, and this can be set with the parameter *MSK_IPAR_MIO_NODE_OPTIMIZER*.
- Some problems are numerically more challenging than others, for example if the ratio between the smallest and the largest involved coefficients is large, say $\geq 1e9$. An indication of numerical issues are, for example, large violations in the final solution, observable in the solution summery of the log output, see Sec. 8.1.3. Similarly, a problem that is known to be feasible by the user may be declared infeasible by the optimizer. In such cases it is usually best to try to rescale the model. Otherwise, the mixed-integer optimizer can be instructed to be more cautios regarding numerics with the parameter *MSK_IPAR_MIO_NUMERICAL_EMPHASIS_LEVEL*. This may in turn be at the cost of solution speed though.
- Improve the formulation: A MIO problem may be impossible to solve in one form and quite easy in another form. However, it is beyond the scope of this manual to discuss good formulations for mixed-integer problems. For discussions on this topic see for example [Wol98].

Chapter 14

Additional features

In this section we describe additional features and tools which enable more detailed analysis of optimization problems with **MOSEK**.

14.1 Problem Analyzer

The problem analyzer prints a survey of the structure of the problem, with information about linear constraints and objective, quadratic constraints, conic constraints and variables.

In the initial stages of model formulation the problem analyzer may be used as a quick way of verifying that the model has been built or imported correctly. In later stages it can help revealing special structures within the model that may be used to tune the optimizer's performance or to identify the causes of numerical difficulties.

The problem analyzer is run using *analyzeproblem*. It prints its output to a log stream. The output is similar to the one below (this is the problem survey of the `aflow30a` problem from the MIPLIB 2003 collection).

```
                    Analyzing the problem

*** Structural report
               Dimensions
               Constraints   Variables     Matrix var.   Cones
               479           842           0             0

               Constraint and bound types
               Free          Lower         Upper         Ranged        Fixed
 Constraints:  0             0             421           0             58
   Variables:  0             0             0             842           0

               Integer constraint types
               Binary        General
               421           0

*** Data report
               Nonzeros      Min           Max
        |cj|:  421           1.1e+01       5.0e+02
       |Aij|:  2091          1.0e+00       1.0e+02

               # finite      Min           Max
      |blci|:  58            1.0e+00       1.0e+01
      |buci|:  479           0.0e+00       1.0e+01
      |blxj|:  842           0.0e+00       0.0e+00
      |buxj|:  842           1.0e+00       1.0e+02
```

(continues on next page)

(continued from previous page)

```
*** Done analyzing the problem
```

The survey is divided into a structural and numerical report. The content should be self-explanatory.

14.2 Automatic Repair of Infeasible Problems

MOSEK provides an automatic repair tool for infeasible linear problems which we cover in this section. Note that most infeasible models are so due to bugs which can (and should) be more reliably fixed manually, using the knowledge of the model structure. We discuss this approach in Sec. 8.3.

14.2.1 Automatic repair

The main idea can be described as follows. Consider the linear optimization problem with m constraints and n variables

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

which is assumed to be infeasible.

One way of making the problem feasible is to reduce the lower bounds and increase the upper bounds. If the change is sufficiently large the problem becomes feasible. Now an obvious idea is to compute the optimal relaxation by solving an optimization problem. The problem

$$\begin{array}{lccccc} \text{minimize} & & & p(v_l^c, v_u^c, v_l^x, v_u^x) & & \\ \text{subject to} & l^c - v_l^c & \leq & Ax & \leq & u^c + v_u^c, \\ & l^x - v_l^x & \leq & x & \leq & u^x + v_u^x, \\ & & & v_l^c, v_u^c, v_l^x, v_u^x \geq 0 & & \end{array} \tag{14.1}$$

does exactly that. The additional variables $(v_l^c)_i$, $(v_u^c)_i$, $(v_l^x)_j$ and $(v_u^c)_j$ are *elasticity* variables because they allow a constraint to be violated and hence add some elasticity to the problem. For instance, the elasticity variable $(v_l^c)_i$ controls how much the lower bound $(l^c)_i$ should be relaxed to make the problem feasible. Finally, the so-called penalty function

$$p(v_l^c, v_u^c, v_l^x, v_u^x)$$

is chosen so it penalizes changes to bounds. Given the weights

- $w_l^c \in \mathbb{R}^m$ (associated with l^c),
- $w_u^c \in \mathbb{R}^m$ (associated with u^c),
- $w_l^x \in \mathbb{R}^n$ (associated with l^x),
- $w_u^x \in \mathbb{R}^n$ (associated with u^x),

a natural choice is

$$p(v_l^c, v_u^c, v_l^x, v_u^x) = (w_l^c)^T v_l^c + (w_u^c)^T v_u^c + (w_l^x)^T v_l^x + (w_u^x)^T v_u^x.$$

Hence, the penalty function $p()$ is a weighted sum of the elasticity variables and therefore the problem (14.1) keeps the amount of relaxation at a minimum. Please observe that

- the problem (14.1) is always feasible.
- a negative weight implies problem (14.1) is unbounded. For this reason if the value of a weight is negative **MOSEK** fixes the associated elasticity variable to zero. Clearly, if one or more of the weights are negative, it may imply that it is not possible to repair the problem.

A simple choice of weights is to set them all to 1, but of course that does not take into account that constraints may have different importance.

Caveats

- Observe if the infeasible problem

$$\begin{array}{lrcl} \text{minimize} & x+z & & \\ \text{subject to} & x & = & -1, \\ & x & \geq & 0 \end{array}$$

  is repaired then it will become unbounded. Hence, a repaired problem may not have an optimal solution.

- Only a minimal repair is performed i.e. the repair that only just makes the problem feasible. Hence, the repaired problem is barely feasible and that sometimes makes the repaired problem hard to solve.

- The infeasibility repair is *unable* to fix crossing bounds, that is ranged constraints of the form $l \leq Ax \leq u$ where $l > u$. In this case it will always fail with an error. Crossing bounds have to be fixed by the user prior to calling automatic repair.

Using the automatic repair tool

In this subsection we consider an infeasible linear optimization example:

$$\begin{array}{lrcrcl} \text{minimize} & -10x_1 & & -9x_2, & & \\ \text{subject to} & 7/10x_1 & + & 1x_2 & \leq & 630, \\ & 1/2x_1 & + & 5/6x_2 & \leq & 600, \\ & 1x_1 & + & 2/3x_2 & \leq & 708, \\ & 1/10x_1 & + & 1/4x_2 & \leq & 135, \\ & x_1, & & x_2 & \geq & 0, \\ & & & x_2 & \geq & 650. \end{array} \tag{14.2}$$

The function *primalrepair* can be used to repair an infeasible problem. This can be used for linear and conic optimization problems, possibly with integer variables.

Listing 14.1: An example of feasibility repair applied to problem (14.2).

```
feasrepair_lp = """
minimize
 obj: - 10 x1 - 9 x2
st
 c1: + 7e-01 x1 + x2 <= 630
 c2: + 5e-01 x1 + 8.333333333e-01 x2 <= 600
 c3: + x1 + 6.6666667e-01 x2 <= 708
 c4: + 1e-01 x1 + 2.5e-01 x2 <= 135
bounds
x2 >= 650
end
"""

using Mosek

if length(ARGS) < 1
    println("Syntax: feasrepairex1 [ FILENAME | - ]")
else
    filename = ARGS[1]
    maketask() do task
        # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
        putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))
```

(continues on next page)

(continued from previous page)

```
        if filename != "-"
            readdata(task,filename)
        else
            readlpstring(task,feasrepair_lp)
        end

        putintparam(task,MSK_IPAR_LOG_FEAS_REPAIR, 3)

        numvar = getnumvar(task)
        numcon = getnumcon(task)
        println(numvar,",",numcon)
        primalrepair(task,zeros(numcon),zeros(numcon),zeros(numvar),zeros(numvar))

        sum_viol = getdouinf(task,MSK_DINF_PRIMAL_REPAIR_PENALTY_OBJ);

        println("Minimized sum of violations = $(sum_viol)");

        optimize(task)
        solutionsummary(task,MSK_STREAM_LOG)
    end
end
```

The above code will produce the following log report:

```
MOSEK Version 9.0.0.25(ALPHA) (Build date: 2017-11-7 16:11:50)
Copyright (c) MOSEK ApS, Denmark. WWW: mosek.com
Platform: Linux/64-X86

Open file 'feasrepair.lp'
Reading started.
Reading terminated. Time: 0.00

Read summary
  Type             : LO (linear optimization problem)
  Objective sense  : min
  Scalar variables : 2
  Matrix variables : 0
  Constraints      : 4
  Cones            : 0
  Time             : 0.0

Problem
  Name                   :
  Objective sense        : min
  Type                   : LO (linear optimization problem)
  Constraints            : 4
  Cones                  : 0
  Scalar variables       : 2
  Matrix variables       : 0
  Integer variables      : 0

Primal feasibility repair started.
Optimizer started.
Presolve started.
Linear dependency checker started.
Linear dependency checker terminated.
Eliminator started.
```

(continues on next page)

(continued from previous page)

```
Freed constraints in eliminator : 2
Eliminator terminated.
Eliminator - tries                  : 1                 time                   : 0.00
Lin. dep.  - tries                  : 1                 time                   : 0.00
Lin. dep.  - number                 : 0
Presolve terminated. Time: 0.00
Problem
  Name                   :
  Objective sense        : min
  Type                   : LO (linear optimization problem)
  Constraints            : 8
  Cones                  : 0
  Scalar variables       : 14
  Matrix variables       : 0
  Integer variables      : 0

Optimizer  - threads                : 20
Optimizer  - solved problem         : the primal
Optimizer  - Constraints            : 2
Optimizer  - Cones                  : 0
Optimizer  - Scalar variables       : 5                 conic                  : 0
Optimizer  - Semi-definite variables: 0                 scalarized             : 0
Factor     - setup time             : 0.00              dense det. time        : 0.00
Factor     - ML order time          : 0.00              GP order time          : 0.00
Factor     - nonzeros before factor : 3                 after factor           : 3
Factor     - dense dim.             : 0                 flops                  : 5.
↪00e+01
ITE PFEAS    DFEAS    GFEAS    PRSTATUS   POBJ              DOBJ              MU
↪ TIME
0   2.7e+01  1.0e+00  4.0e+00  1.00e+00   3.000000000e+00   0.000000000e+00   1.0e+00␣
↪ 0.00
1   2.5e+01  9.1e-01  1.4e+00  0.00e+00   8.711262850e+00   1.115287830e+01   2.4e+00␣
↪ 0.00
2   2.4e+00  8.8e-02  1.4e-01  -7.33e-01  4.062505701e+01   4.422203730e+01   2.3e-01␣
↪ 0.00
3   9.4e-02  3.4e-03  5.5e-03  1.33e+00   4.250700434e+01   4.258548510e+01   9.1e-03␣
↪ 0.00
4   2.0e-05  7.2e-07  1.1e-06  1.02e+00   4.249996599e+01   4.249998669e+01   1.9e-06␣
↪ 0.00
5   2.0e-09  7.2e-11  1.1e-10  1.00e+00   4.250000000e+01   4.250000000e+01   1.9e-10␣
↪ 0.00
Basis identification started.
Basis identification terminated. Time: 0.00
Optimizer terminated. Time: 0.01

Basic solution summary
  Problem status  : PRIMAL_AND_DUAL_FEASIBLE
  Solution status : OPTIMAL
  Primal.  obj: 4.2500000000e+01    nrm: 6e+02    Viol.  con: 1e-13    var: 0e+00
  Dual.    obj: 4.2499999999e+01    nrm: 2e+00    Viol.  con: 0e+00    var: 9e-11
Optimal objective value of the penalty problem: 4.250000000000e+01

Repairing bounds.
Increasing the upper bound 1.35e+02 on constraint 'c4' (3) with 2.25e+01.
Decreasing the lower bound 6.50e+02 on variable 'x2' (4) with 2.00e+01.
Primal feasibility repair terminated.
```

(continues on next page)

(continued from previous page)

```
Optimizer started.
Optimizer terminated. Time: 0.00


Interior-point solution summary
  Problem status  : PRIMAL_AND_DUAL_FEASIBLE
  Solution status : OPTIMAL
  Primal.  obj: -5.6700000000e+03   nrm: 6e+02    Viol.  con: 0e+00    var: 0e+00
  Dual.    obj: -5.6700000000e+03   nrm: 1e+01    Viol.  con: 0e+00    var: 0e+00

Basic solution summary
  Problem status  : PRIMAL_AND_DUAL_FEASIBLE
  Solution status : OPTIMAL
  Primal.  obj: -5.6700000000e+03   nrm: 6e+02    Viol.  con: 0e+00    var: 0e+00
  Dual.    obj: -5.6700000000e+03   nrm: 1e+01    Viol.  con: 0e+00    var: 0e+00

Optimizer summary
  Optimizer                 -                        time: 0.00
    Interior-point          - iterations : 0         time: 0.00
      Basis identification  -                        time: 0.00
        Primal              - iterations : 0         time: 0.00
        Dual                - iterations : 0         time: 0.00
        Clean primal        - iterations : 0         time: 0.00
        Clean dual          - iterations : 0         time: 0.00
    Simplex                 -                        time: 0.00
      Primal simplex        - iterations : 0         time: 0.00
      Dual simplex          - iterations : 0         time: 0.00
    Mixed integer           - relaxations: 0         time: 0.00
```

It will also modify the task according to the optimal elasticity variables found. In this case the optimal repair it is to increase the upper bound on constraint `c4` by 22.5 and decrease the lower bound on variable `x2` by 20.

14.3 Sensitivity Analysis

Given an optimization problem it is often useful to obtain information about how the optimal objective value changes when the problem parameters are perturbed. E.g, assume that a bound represents the capacity of a machine. Now, it may be possible to expand the capacity for a certain cost and hence it is worthwhile knowing what the value of additional capacity is. This is precisely the type of questions the sensitivity analysis deals with.

Analyzing how the optimal objective value changes when the problem data is changed is called *sensitivity analysis*.

References

The book [Chvatal83] discusses the classical sensitivity analysis in Chapter 10 whereas the book [RTV97] presents a modern introduction to sensitivity analysis. Finally, it is recommended to read the short paper [Wal00] to avoid some of the pitfalls associated with sensitivity analysis.

Warning: Currently, sensitivity analysis is only available for continuous linear optimization problems. Moreover, **MOSEK** can only deal with perturbations of bounds and objective function coefficients.

14.3.1 Sensitivity Analysis for Linear Problems

The Optimal Objective Value Function

Assume that we are given the problem

$$\begin{array}{rcllcll} z(l^c, u^c, l^x, u^x, c) & = & \text{minimize} & & & c^T x & \\ & & \text{subject to} & l^c & \leq & Ax & \leq u^c, \\ & & & l^x & \leq & x & \leq u^x, \end{array} \tag{14.3}$$

and we want to know how the optimal objective value changes as l_i^c is perturbed. To answer this question we define the perturbed problem for l_i^c as follows

$$\begin{array}{rcllcll} f_{l_i^c}(\beta) & = & \text{minimize} & & & c^T x & \\ & & \text{subject to} & l^c + \beta e_i & \leq & Ax & \leq u^c, \\ & & & l^x & \leq & x \leq & u^x, \end{array}$$

where e_i is the i-th column of the identity matrix. The function

$$f_{l_i^c}(\beta) \tag{14.4}$$

shows the optimal objective value as a function of β. Please note that a change in β corresponds to a perturbation in l_i^c and hence (14.4) shows the optimal objective value as a function of varying l_i^c with the other bounds fixed.

It is possible to prove that the function (14.4) is a piecewise linear and convex function, i.e. its graph may look like in Fig. 14.1 and Fig. 14.2.

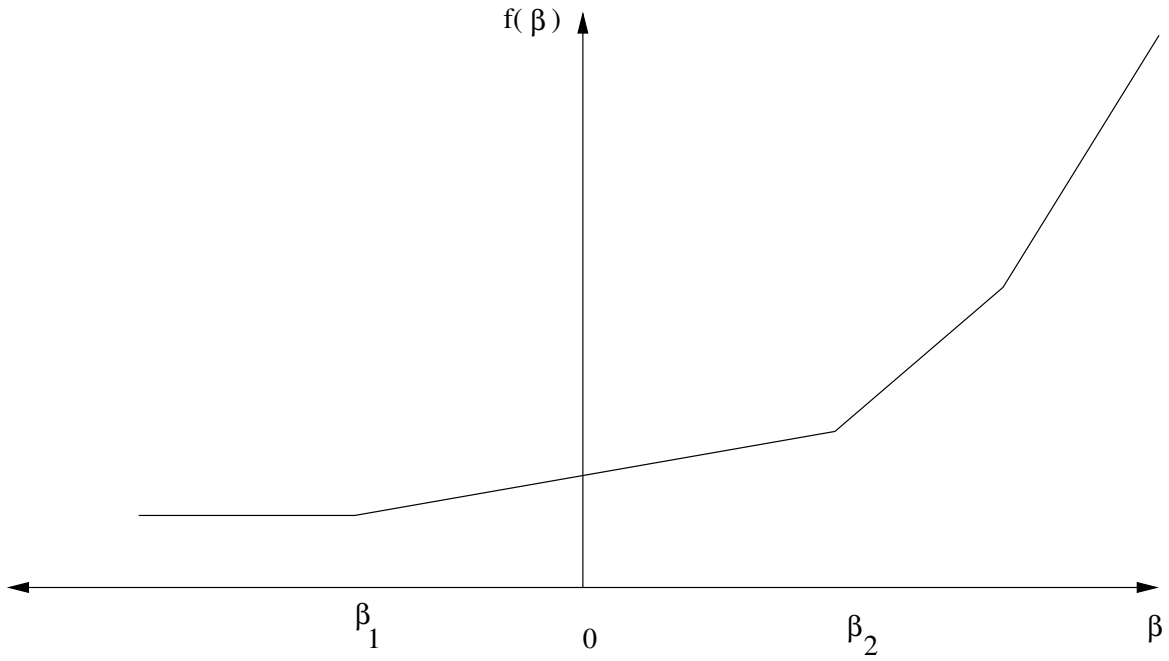


Fig. 14.1: $\beta = 0$ is in the interior of linearity interval.

Clearly, if the function $f_{l_i^c}(\beta)$ does not change much when β is changed, then we can conclude that the optimal objective value is insensitive to changes in l_i^c. Therefore, we are interested in the rate of change in $f_{l_i^c}(\beta)$ for small changes in β — specifically the gradient

$$f'_{l_i^c}(0),$$

which is called the *shadow price* related to l_i^c. The shadow price specifies how the objective value changes for small changes of β around zero. Moreover, we are interested in the *linearity interval*

$$\beta \in [\beta_1, \beta_2]$$

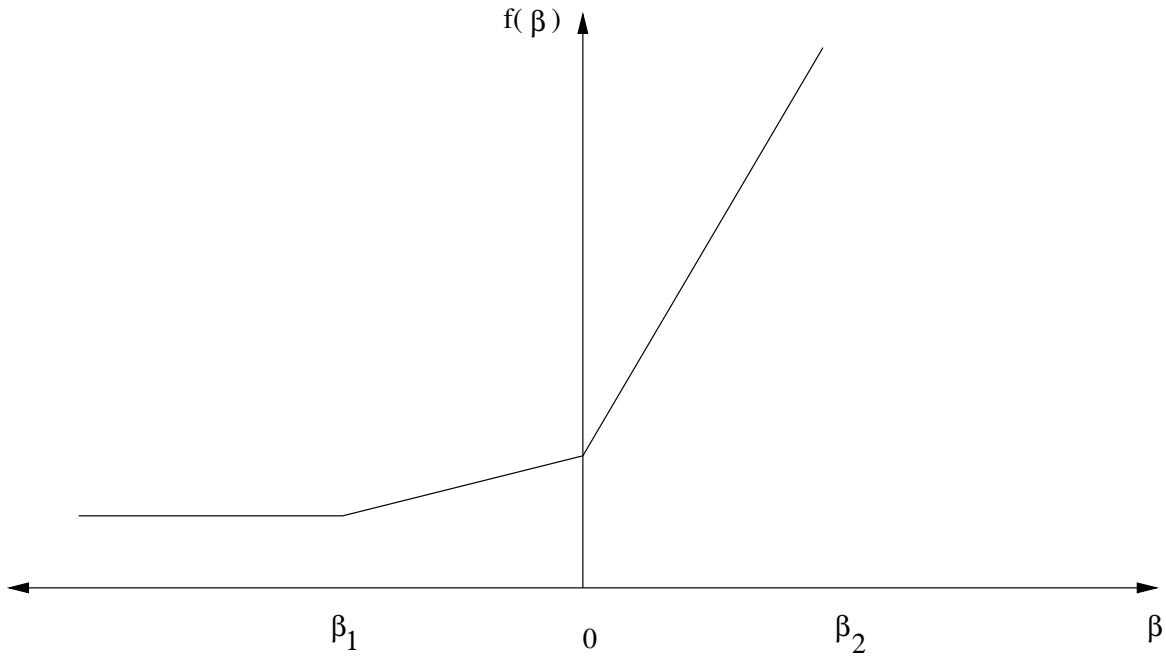


Fig. 14.2: $\beta = 0$ is a breakpoint.

for which

$$f'_{l_i^c}(\beta) = f'_{l_i^c}(0).$$

Since $f_{l_i^c}$ is not a smooth function $f'_{l_i^c}$ may not be defined at 0, as illustrated in Fig. 14.2. In this case we can define a left and a right shadow price and a left and a right linearity interval.

The function $f_{l_i^c}$ considered only changes in l_i^c. We can define similar functions for the remaining parameters of the z defined in (14.3) as well:

$$\begin{array}{rcll} f_{l_i^c}(\beta) & = & z(l^c + \beta e_i, u^c, l^x, u^x, c), & i = 1, \ldots, m, \\ f_{u_i^c}(\beta) & = & z(l^c, u^c + \beta e_i, l^x, u^x, c), & i = 1, \ldots, m, \\ f_{l_j^x}(\beta) & = & z(l^c, u^c, l^x + \beta e_j, u^x, c), & j = 1, \ldots, n, \\ f_{u_j^x}(\beta) & = & z(l^c, u^c, l^x, u^x + \beta e_j, c), & j = 1, \ldots, n, \\ f_{c_j}(\beta) & = & z(l^c, u^c, l^x, u^x, c + \beta e_j), & j = 1, \ldots, n. \end{array}$$

Given these definitions it should be clear how linearity intervals and shadow prices are defined for the parameters u_i^c etc.

Equality Constraints

In **MOSEK** a constraint can be specified as either an equality constraint or a ranged constraint. If some constraint e_i^c is an equality constraint, we define the optimal value function for this constraint as

$$f_{e_i^c}(\beta) = z(l^c + \beta e_i, u^c + \beta e_i, l^x, u^x, c)$$

Thus for an equality constraint the upper and the lower bounds (which are equal) are perturbed simultaneously. Therefore, **MOSEK** will handle sensitivity analysis differently for a ranged constraint with $l_i^c = u_i^c$ and for an equality constraint.

The Basis Type Sensitivity Analysis

The classical sensitivity analysis discussed in most textbooks about linear optimization, e.g. [Chvatal83], is based on an optimal basis. This method may produce misleading results [RTV97] but is computationally cheap. This is the type of sensitivity analysis implemented in **MOSEK**.

We will now briefly discuss the basis type sensitivity analysis. Given an optimal basic solution which provides a partition of variables into basic and non-basic variables, the basis type sensitivity analysis computes the linearity interval $[\beta_1, \beta_2]$ so that the basis remains optimal for the perturbed problem. A shadow price associated with the linearity interval is also computed. However, it is well-known that an optimal basic solution may not be unique and therefore the result depends on the optimal basic solution employed in the sensitivity analysis. If the optimal objective value function has a breakpoint for $\beta = 0$ then the basis type sensitivity method will only provide a subset of either the left or the right linearity interval.

In summary, the basis type sensitivity analysis is computationally cheap but does not provide complete information. Hence, the results of the basis type sensitivity analysis should be used with care.

Example: Sensitivity Analysis

As an example we will use the following transportation problem. Consider the problem of minimizing the transportation cost between a number of production plants and stores. Each plant supplies a number of goods and each store has a given demand that must be met. Supply, demand and cost of transportation per unit are shown in Fig. 14.3.

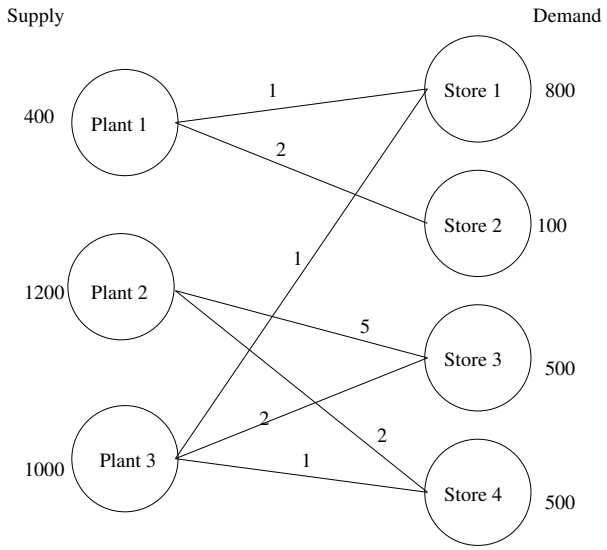


Fig. 14.3: Supply, demand and cost of transportation.

If we denote the number of transported goods from location i to location j by x_{ij}, problem can be formulated as the linear optimization problem of minimizing

$$1x_{11} + 2x_{12} + 5x_{23} + 2x_{24} + 1x_{31} + 2x_{33} + 1x_{34}$$

subject to

$$\begin{array}{rcrcrcrcrcrcl}
x_{11} & + & x_{12} & & & & & & & & & \leq & 400, \\
 & & & x_{23} & + & x_{24} & & & & & & \leq & 1200, \\
 & & & & & & x_{31} & + & x_{33} & + & x_{34} & \leq & 1000, \\
x_{11} & & & & & & + & x_{31} & & & & = & 800, \\
 & & x_{12} & & & & & & & & & = & 100, \\
 & & & x_{23} & + & & & & x_{33} & & & = & 500, \\
 & & & & & x_{24} & + & & & & x_{34} & = & 500, \\
x_{11}, & & x_{12}, & & x_{23}, & & x_{24}, & & x_{31}, & & x_{33}, & & x_{34} \geq 0.
\end{array} \tag{14.5}$$

The sensitivity parameters are shown in Table 14.1 and Table 14.2.

Table 14.1: Ranges and shadow prices related to bounds on constraints and variables.

Con.	β_1	β_2	σ_1	σ_2
1	-300.00	0.00	3.00	3.00
2	-700.00	$+\infty$	0.00	0.00
3	-500.00	0.00	3.00	3.00
4	-0.00	500.00	4.00	4.00
5	-0.00	300.00	5.00	5.00
6	-0.00	700.00	5.00	5.00
7	-500.00	700.00	2.00	2.00
Var.	β_1	β_2	σ_1	σ_2
x_{11}	$-\infty$	300.00	0.00	0.00
x_{12}	$-\infty$	100.00	0.00	0.00
x_{23}	$-\infty$	0.00	0.00	0.00
x_{24}	$-\infty$	500.00	0.00	0.00
x_{31}	$-\infty$	500.00	0.00	0.00
x_{33}	$-\infty$	500.00	0.00	0.00
x_{34}	-0.000000	500.00	2.00	2.00

Table 14.2: Ranges and shadow prices related to the objective coefficients.

Var.	β_1	β_2	σ_1	σ_2
c_1	$-\infty$	3.00	300.00	300.00
c_2	$-\infty$	∞	100.00	100.00
c_3	-2.00	∞	0.00	0.00
c_4	$-\infty$	2.00	500.00	500.00
c_5	-3.00	∞	500.00	500.00
c_6	$-\infty$	2.00	500.00	500.00
c_7	-2.00	∞	0.00	0.00

Examining the results from the sensitivity analysis we see that for constraint number 1 we have $\sigma_1 = 3$ and $\beta_1 = -300,\ \beta_2 = 0$.

If the upper bound on constraint 1 is decreased by

$$\beta \in [0, 300]$$

then the optimal objective value will increase by the value

$$\sigma_1 \beta = 3\beta.$$

14.3.2 Sensitivity Analysis with MOSEK

MOSEK provides the functions *primalsensitivity* and *dualsensitivity* for performing sensitivity analysis. The code in Listing 14.2 gives an example of its use.

Listing 14.2: Example of sensitivity analysis with the **MOSEK** Optimizer API for Julia.

```
using Mosek

prob_ptf = "Task ''
Objective
  Minimize  + 1 x11 + 2 x12 + 5 x23 + 2 x24 + 1 x31 + 2 x33 + 1 x34
Constraints
  c1 [-inf;400 ]  +  x11 +   x12
  c2 [-inf;1200]               +   x23 +   x24
  c3 [-inf;1000]                                +   x31 +   x33 +   x34
  c4 [800]        +  x11                        +   x31
  c5 [100]              +   x12
  c6 [500]                    +   x23                 +   x33
  c7 [500]                          +   x24                 +   x34
Variables
  x11 [0;+inf]
  x12 [0;+inf]
  x23 [0;+inf]
  x24 [0;+inf]
  x31 [0;+inf]
  x33 [0;+inf]
  x34 [0;+inf]
"

maketask() do task
    # Use remote server: putoptserverhost(task,"http://solve.mosek.com:30080")
    putstreamfunc(task,MSK_STREAM_LOG,msg -> print(msg))

    readptfstring(task,prob_ptf)

    # A maximization problem

    optimize(task)
    checkmem(task,@__FILE__,@__LINE__)
    # Analyze upper bound on c1 and the equality constraint on c4
    let subi = [1, 4],
        marki = [MSK_MARK_UP, MSK_MARK_UP],
        # Analyze lower bound on the variables x12 and x31
        subj  = [2, 5],
        markj = [MSK_MARK_LO,MSK_MARK_LO],
        (leftpricei,
         rightpricei,
         leftrangei,
         rightrangei,
         leftpricej,
         rightpricej,
         leftrangej,
         rightrangej) = primalsensitivity(task,
                                          subi,
                                          marki,
                                          subj,
```

(continues on next page)

(continued from previous page)

```
                                          markj)
        checkmem(task,@__FILE__,@__LINE__)
        println("Results from sensitivity analysis on bounds:  ")

        println("For constraints:")
        for i in 1:2
            println("leftprice = $(leftpricei[i]) | rightprice = $(rightpricei[i]) |␣
↪leftrange = $(leftrangei[i]) | rightrange = $(rightrangei[i])");
        end
        println("For variables:")
        for i in 1:2
            println("leftprice = $(leftpricej[i]) rightprice = $(rightpricej[i])␣
↪leftrange = $(leftrangej[i]) rightrange = $(rightrangej[i])")
        end
    end

    let subc  = Int32[2, 5],
        (leftprice,
         rightprice,
         leftrange,
         rightrange) = dualsensitivity(task,subc)

        println("Results from sensitivity analysis on objective coefficients:")

        for i in 1:2
            println("leftprice = $(leftprice[i]) rightprice = $(rightprice[i])␣
↪leftrange = $(leftrange[i]) rightrange = $( rightrange[i])")
        end
    end
end
```

Chapter 15

API Reference

This section contains the complete reference of the **MOSEK** Optimizer API for Julia. It is organized as follows:

- *General API conventions.*
- **Functions:**
  - *Full list*
  - *Browse by topic*
- **Optimizer parameters:**
  - *Double*, *Integer*, *String*
  - *Full list*
  - *Browse by topic*
- **Optimizer information items:**
  - `Double`, `Integer`, `Long`
- *Optimizer response codes*
- *Constants*
- *List of supported domains*
- *Environment variables*

15.1 API Conventions

15.1.1 Function arguments

Naming Convention

In the definition of the **MOSEK** Optimizer API for Julia a consistent naming convention has been used. This implies that whenever for example `numcon` is an argument in a function definition it indicates the number of constraints. In Table 15.1 the variable names used to specify the problem parameters are listed.

Table 15.1: Naming conventions used in the **MOSEK** Optimizer API for Julia.

API name	API type	Dimension	Related problem parameter
`numcon`	`int`		m
`numvar`	`int`		n
`numcone`	`int`		t
`aptrb`	`int[]`	`numvar`	a_{ij}
`aptre`	`int[]`	`numvar`	a_{ij}
`asub`	`int[]`	`aptre[numvar-1]`	a_{ij}
`aval`	`double[]`	`aptre[numvar-1]`	a_{ij}
`c`	`double[]`	`numvar`	c_j
`cfix`	`double`		c^f
`blc`	`double[]`	`numcon`	l_k^c
`buc`	`double[]`	`numcon`	u_k^c
`blx`	`double[]`	`numvar`	l_k^x
`bux`	`double[]`	`numvar`	u_k^x
`numqonz`	`int`		q_{ij}^o
`qosubi`	`int[]`	`numqonz`	q_{ij}^o
`qosubj`	`int[]`	`numqonz`	q_{ij}^o
`qoval`	`double[]`	`numqonz`	q_{ij}^o
`numqcnz`	`int`		q_{ij}^k
`qcsubk`	`int[]`	`numqcnz`	q_{ij}^k
`qcsubi`	`int[]`	`numqcnz`	q_{ij}^k
`qcsubj`	`int[]`	`numqcnz`	q_{ij}^k
`qcval`	`double[]`	`numqcnz`	q_{ij}^k
`bkc`	`int[]`	`numcon`	l_k^c and u_k^c
`bkx`	`int[]`	`numvar`	l_k^x and u_k^x

The relation between the variable names and the problem parameters is as follows:

- The quadratic terms in the objective: $q^o_{\texttt{qosubi[t]},\texttt{qosubj[t]}} = \texttt{qoval[t]}, \quad t = 1, \ldots, \texttt{numqonz}$.
- The linear terms in the objective : $c_j = \texttt{c[j]}, \quad j = 1, \ldots, \texttt{numvar}$
- The fixed term in the objective : $c^f = \texttt{cfix}$.
- The quadratic terms in the constraints: $q^{\texttt{qcsubk[t]}}_{\texttt{qcsubi[t]},\texttt{qcsubj[t]}} = \texttt{qcval[t]}, \quad t = 1, \ldots, \texttt{numqcnz}$
- The linear terms in the constraints: $a_{\texttt{asub[t]},\texttt{j}} = \texttt{aval[t]}, \quad t = \texttt{ptrb[j]}, \ldots, \texttt{ptre[j]} - 1, \; j = 1, \ldots, \texttt{numvar}$

Information about input/output arguments

The following are purely informational tags which indicate how **MOSEK** treats a specific function argument.

- (input) An input argument. It is used to input data to **MOSEK**.
- (output) An output argument. It can be a user-preallocated data structure, a reference, a string buffer etc. where **MOSEK** will output some data.
- (input/output) An input/output argument. **MOSEK** will read the data and overwrite it with new/updated information.

15.1.2 Bounds

The bounds on the constraints and variables are specified using the variables `bkc`, `blc`, and `buc`. The components of the integer array `bkc` specify the bound type according to Table 15.2

Table 15.2: Symbolic key for variable and constraint bounds.

Symbolic constant	Lower bound	Upper bound
MSK_BK_FX	finite	identical to the lower bound
MSK_BK_FR	minus infinity	plus infinity
MSK_BK_LO	finite	plus infinity
MSK_BK_RA	finite	finite
MSK_BK_UP	minus infinity	finite

For instance `bkc[2]`=*MSK_BK_LO* means that $-\infty < l_2^c$ and $u_2^c = \infty$. Even if a variable or constraint is bounded only from below, e.g. $x \geq 0$, both bounds are inputted or extracted; the irrelevant value is ignored.

Finally, the numerical values of the bounds are given by

$$l_k^c = \texttt{blc}[\texttt{k}], \quad k = 1, \ldots, \texttt{numcon}$$

$$u_k^c = \texttt{buc}[\texttt{k}], \quad k = 1, \ldots, \texttt{numcon}.$$

The bounds on the variables are specified using the variables `bkx`, `blx`, and `bux` in the same way. The numerical values for the lower bounds on the variables are given by

$$l_j^x = \texttt{blx}[\texttt{j}], \quad j = 1, \ldots, \texttt{numvar}.$$

$$u_j^x = \texttt{bux}[\texttt{j}], \quad j = 1, \ldots, \texttt{numvar}.$$

15.1.3 Vector Formats

Three different vector formats are used in the **MOSEK** API:

Full (dense) vector

This is simply an array where the first element corresponds to the first item, the second element to the second item etc. For example to get the linear coefficients of the objective in `task` with `numvar` variables, one would write

```
c = getc(task)
```

Vector slice

A vector slice is a range of values from `first` up to and **not including** `last` entry in the vector, i.e. for the set of indices `i` such that `first <= i < last`. For example, to get the bounds associated with constrains 2 through 9 (both inclusive) one would write

```
upper_bound = zeros(Float64,8)
lower_bound = zeros(Float64,8)
bound_key   = fill(MSK_BK_FR,8)

bound_key,lower_bound,upper_bound = getconboundslice(task, 3, 11)
```

Sparse vector

A sparse vector is given as an array of indexes and an array of values. The indexes need not be ordered. For example, to input a set of bounds associated with constraints number 1, 6, 3, and 9, one might write

```
bound_index = Int32[        1,        6,        3,        9]
bound_key   = [MSK_BK_FR,MSK_BK_LO,MSK_BK_UP,MSK_BK_FX]
lower_bound = Float64[     0.0,         -10.0,        0.0,        5.0]
upper_bound = Float64[     0.0,           0.0,        6.0,        5.0]
putconboundlist(task,bound_index,
                bound_key,lower_bound,upper_bound)
```

15.1.4 Matrix Formats

The coefficient matrices in a problem are inputted and extracted in a sparse format. That means only the nonzero entries are listed.

Julia Sparse Matrices

Optimizer API for Julia accepts standard sparse matrices of type `SparseMatrixCSC` whenever the interface expects the linear constraint `A` matrix or its parts. For example, one can input the whole `A` matrix as follows

```
A = sparse([1, 2, 1, 2, 3, 1, 2, 2, 3],
           [1, 1, 2, 2, 2, 3, 3, 4, 4],
           [3.0, 2.0, 1.0, 1.0, 2.0, 2.0, 3.0, 1.0, 3.0 ],
           numcon,numvar)

putacolslice(task, 1, numvar+1, A)
```

Unordered Triplets

In unordered triplet format each entry is defined as a row index, a column index and a coefficient. For example, to input the A matrix coefficients for $a_{1,2} = 1.1, a_{3,3} = 4.3$, and $a_{5,4} = 0.2$, one would write as follows:

```
subi = Int32[   1,   3,   5 ]
subj = Int32[   2,   3,   4 ]
cof  = Float64[ 1.1, 4.3, 0.2 ]
putaijlist(task,subi,subj,cof)
```

Please note that in some cases (like *putaijlist*) *only* the specified indexes are modified — all other are unchanged. In other cases (such as *putqconk*) the triplet format is used to modify *all* entries — entries that are not specified are set to 0.

Column or Row Ordered Sparse Matrix

In a sparse matrix format only the non-zero entries of the matrix are stored. **MOSEK** uses a sparse packed matrix format ordered either by columns or rows. Here we describe the column-wise format. The row-wise format is based on the same principle.

Column ordered sparse format

A sparse matrix in column ordered format is essentially a list of all non-zero entries read column by column from left to right and from top to bottom within each column. The exact representation uses four arrays:

- `asub`: Array of size equal to the number of nonzeros. List of row indexes.
- `aval`: Array of size equal to the number of nonzeros. List of non-zero entries of A ordered by columns.
- `ptrb`: Array of size `numcol`, where `ptrb[j]` is the position of the first value/index in `aval`/ `asub` for the j-th column.
- `ptre`: Array of size `numcol`, where `ptre[j]` is the position of the last value/index plus one in `aval` / `asub` for the j-th column.

With this representation the values of a matrix A with `numcol` columns are assigned using:

$$a_{\texttt{asub}[k],j} = \texttt{aval}[k] \quad \text{for} \quad j = 1, \ldots, \texttt{numcol} - 1, \; k = \texttt{ptrb}[j], \ldots, \texttt{ptre}[j] - 1.$$

As an example consider the matrix

$$A = \begin{bmatrix} 1.1 & & 1.3 & 1.4 & \\ & 2.2 & & & 2.5 \\ 3.1 & & & 3.4 & \\ & & 4.4 & & \end{bmatrix} \tag{15.1}$$

which can be represented in the column ordered sparse matrix format as

$$\begin{array}{lcl} \texttt{ptrb} & = & [1, 3, 4, 6, 8], \\ \texttt{ptre} & = & [3, 4, 6, 8, 9], \\ \texttt{asub} & = & [1, 3, 2, 1, 4, 1, 3, 2], \\ \texttt{aval} & = & [1.1, 3.1, 2.2, 1.3, 4.4, 1.4, 3.4, 2.5]. \end{array}$$

Column ordered sparse format with nonzeros

Note that `nzc[j] := ptre[j]-ptrb[j]` is exactly the number of nonzero elements in the j-th column of A. In some functions a sparse matrix will be represented using the equivalent dataset `asub`, `aval`, `ptrb`, `nzc`. The matrix A (15.1) would now be represented as:

$$\begin{array}{lcl} \texttt{ptrb} & = & [1, 3, 4, 6, 8], \\ \texttt{nzc} & = & [2, 1, 2, 2, 1], \\ \texttt{asub} & = & [1, 3, 2, 1, 4, 1, 3, 2], \\ \texttt{aval} & = & [1.1, 3.1, 2.2, 1.3, 4.4, 1.4, 3.4, 2.5]. \end{array}$$

Row ordered sparse matrix

The matrix A (15.1) can also be represented in the row ordered sparse matrix format as:

$$\begin{array}{lcl} \texttt{ptrb} & = & [1, 4, 6, 8], \\ \texttt{ptre} & = & [4, 6, 8, 9], \\ \texttt{asub} & = & [1, 3, 2, 2, 5, 1, 4, 3], \\ \texttt{aval} & = & [1.1, 1.3, 1.4, 2.2, 2.5, 3.1, 3.4, 4.4]. \end{array}$$

15.2 Functions grouped by topic

Callback

- *clearcallbackfunc* – Clears callbacks.
- *clearstreamfunc* – Clears a stream callback.
- *putcallbackfunc* – Receive callbacks with solver status and information during optimization.
- *putstreamfunc* – Directs all output from a task stream to a callback function.

Environment and task management

- *getdualproblem* – Obtains the dual problem.
- *makeenv* – Creates a new environment.
- *maketask* – Creates a new task.
- *puttaskname* – Assigns a new name to the task.
- *Infrequent:* *commitchanges*, *deletesolution*, *putmaxnumacc*, *putmaxnumafe*, *putmaxnumanz*, *putmaxnumbarvar*, *putmaxnumcon*, *putmaxnumdjc*, *putmaxnumdomain*, *putmaxnumqnz*, *putmaxnumvar*, *resizetask*
- *Deprecated:* ~~*putmaxnumcone*~~

Infeasibility diagnostic

- *getinfeasiblesubproblem* – Obtains an infeasible subproblem.
- *infeasibilityreport* – Prints the infeasibility report to an output stream.
- *primalrepair* – Repairs a primal infeasible optimization problem by adjusting the bounds on the constraints and variables.

Information items and statistics

- *getdouinf* – Obtains a double information item.
- *getintinf* – Obtains an integer information item.
- *getlintinf* – Obtains a long integer information item.
- *updatesolutioninfo* – Update the information items related to the solution.
- *Infrequent:* *getinfname*, *getnadouinf*, *getnaintinf*

Input/Output

- *writedata* – Writes problem data to a file.
- *writesolution* – Write a solution to a file.
- *Infrequent:* *readbsolution*, *readdata*, *readdataformat*, *readjsonsol*, *readjsonstring*, *readlpstring*, *readopfstring*, *readparamfile*, *readptfstring*, *readsolution*, *readsolutionfile*, *readsummary*, *readtask*, *writebsolution*, *writedatastream*, *writejsonsol*, *writeparamfile*, *writesolutionfile*, *writetask*

Inspecting the task

- *analyzeproblem* – Analyze the data of a task.
- *getnumcon* – Obtains the number of constraints.
- *getnumvar* – Obtains the number of variables.
- *Infrequent:* *analyzesolution*, *getaccafeidxlist*, *getaccb*, *getaccbarfnumblocktriplets*, *getaccdomain*, *getaccfnumnz*, *getaccftrip*, *getaccgvector*, *getaccn*, *getaccname*, *getaccnamelen*, *getaccntot*, *getaccs*, *getacol*, *getacolnumnz*, *getacolslice*, *getacolslicenumnz*, *getacolslicetrip*, *getafebarfnumblocktriplets*, *getafebarfnumrowentries*, *getafebarfrow*, *getafebarfrowinfo*, *getafefnumnz*, *getafefrow*, *getafefrownumnz*, *getafeftrip*, *getafeg*, *getafegslice*, *getaij*, *getapiecenumnz*, *getarow*, *getarownumnz*, *getarowslice*, *getarowslicenumnz*, *getarowslicetrip*, *getatrip*, *getbarablocktriplet*, *getbaraidx*, *getbaraidxij*, *getbaraidxinfo*, *getbarasparsity*, *getbarcblocktriplet*, *getbarcidx*, *getbarcidxinfo*, *getbarcidxj*, *getbarcsparsity*, *getbarvarname*, *getbarvarnameindex*, *getbarvarnamelen*, *getc*, *getcfix*, *getcj*, *getclist*, *getconbound*, *getconboundslice*, *getconname*, *getconnameindex*, *getconnamelen*, *getcslice*, *getdimbarvarj*, *getdjcafeidxlist*, *getdjcb*, *getdjcdomainidxlist*, *getdjcname*, *getdjcnamelen*, *getdjcnumafe*, *getdjcnumafetot*, *getdjcnumdomain*, *getdjcnumdomaintot*, *getdjcnumterm*, *getdjcnumtermtot*, *getdjcs*, *getdjctermsizelist*, *getdomainn*, *getdomainname*, *getdomainnamelen*, *getdomaintype*, *getlenbarvarj*, *getmaxnumanz*, *getmaxnumbarvar*, *getmaxnumcon*, *getmaxnumqnz*, *getmaxnumvar*, *getnumacc*, *getnumafe*, *getnumanz*, *getnumbarablocktriplets*, *getnumbaranz*, *getnumbarcblocktriplets*, *getnumbarcnz*, *getnumbarvar*, *getnumdjc*, *getnumdomain*, *getnumintvar*, *getnumparam*, *getnumqconknz*, *getnumqobjnz*, *getnumsymmat*, *getobjname*, *getobjnamelen*, *getpowerdomainalpha*, *getpowerdomaininfo*, *getprobtype*, *getqconk*, *getqobj*, *getqobjij*, *getsparsesymmat*, *getsymmatinfo*, *gettaskname*, *gettasknamelen*, *getvarbound*, *getvarboundslice*, *getvarname*, *getvarnameindex*, *getvarnamelen*, *getvartype*, *getvartypelist*, *printparam*, *probtypetostr*, *readsummary*
- *Deprecated:* ~~*getcone*~~, ~~*getconeinfo*~~, ~~*getconename*~~, ~~*getconenameindex*~~, ~~*getconenamelen*~~, ~~*getmaxnumcone*~~, ~~*getnumcone*~~, ~~*getnumconemem*~~

License system

- *checkoutlicense* – Check out a license feature from the license server ahead of time.
- *putlicensedebug* – Enables debug information for the license system.
- *putlicensepath* – Set the path to the license file.
- *putlicensewait* – Control whether mosek should wait for an available license if no license is available.
- *Infrequent:* *checkinall*, *checkinlicense*, *expirylicenses*, *licensecleanup*, *putlicensecode*, *resetexpirylicenses*

Linear algebra

- *Infrequent:* *computesparsecholesky*, *sparsetriangularsolvedense*

Logging

- *clearcallbackfunc* – Clears callbacks.
- *clearstreamfunc* – Clears a stream callback.
- *linkfiletostream* – Directs all output from a task stream to a file.
- *onesolutionsummary* – Prints a short summary of a specified solution.
- *optimizersummary* – Prints a short summary with optimizer statistics from last optimization.
- *putstreamfunc* – Directs all output from a task stream to a callback function.
- *solutionsummary* – Prints a short summary of the current solutions.
- *Infrequent:* *echointro*, *linkfiletostream*, *printparam*

Names

- *getcodedesc* – Obtains a short description of a response code.
- *putaccname* – Sets the name of an affine conic constraint.
- *putbarvarname* – Sets the name of a semidefinite variable.
- *putconname* – Sets the name of a constraint.
- *putdjcname* – Sets the name of a disjunctive constraint.
- *putdomainname* – Sets the name of a domain.
- *putobjname* – Assigns a new name to the objective.
- *puttaskname* – Assigns a new name to the task.
- *putvarname* – Sets the name of a variable.
- *Infrequent:* *analyzenames*, *bktostr*, *callbackcodetostr*, *dinfitemtostr*, *getaccname*, *getaccnamelen*, *getbarvarname*, *getbarvarnameindex*, *getbarvarnamelen*, *getconname*, *getconnameindex*, *getconnamelen*, *getdjcname*, *getdjcnamelen*, *getdomainname*, *getdomainnamelen*, *getinfname*, *getnastrparam*, *getobjname*, *getobjnamelen*, *getparamname*, *getstrparam*, *getstrparamlen*, *gettaskname*, *gettasknamelen*, *getvarname*, *getvarnameindex*, *getvarnamelen*, *iinfitemtostr*, *isdouparname*, *isintparname*, *isstrparname*, *liinfitemtostr*, *probtypetostr*, *prostatostr*, *rescodetostr*, *strtosk*
- *Deprecated:* ~~*conetypetostr*~~, ~~*getconename*~~, ~~*getconenameindex*~~, ~~*getconenamelen*~~, ~~*putconename*~~, ~~*strtoconetype*~~

Optimization

- *optimize* – Optimizes the problem.
- *optimizebatch* – Optimize a number of tasks in parallel using a specified number of threads.

Parameters

- *putdouparam* – Sets a double parameter.
- *putintparam* – Sets an integer parameter.
- *putlintparam* – Sets an integer parameter.
- *putparam* – Modifies the value of parameter.
- *putstrparam* – Sets a string parameter.
- *resetdouparam* – Resets a double parameter to its default value.
- *resetintparam* – Resets an integer parameter to its default value.
- *resetparameters* – Resets all parameter values.
- *resetstrparam* – Resets a string parameter to its defalt value.
- *Infrequent:* *getatruncatetol*, *getdouparam*, *getintparam*, *getlintparam*, *getnadouparam*, *getnaintparam*, *getnastrparam*, *getnumparam*, *getparamname*, *getstrparam*, *getstrparamlen*, *isdouparname*, *isintparname*, *isstrparname*, *putnadouparam*, *putnaintparam*, *putnastrparam*, *readparamfile*, *writeparamfile*

Problem data - affine conic constraints

- *appendacc* – Appends an affine conic constraint to the task.
- *getaccdoty* – Obtains the doty vector for an affine conic constraint.
- *putaccname* – Sets the name of an affine conic constraint.
- *Infrequent:* *appendaccs*, *appendaccseq*, *appendaccsseq*, *evaluateacc*, *evaluateaccs*, *getaccafeidxlist*, *getaccb*, *getaccbarfnumblocktriplets*, *getaccdomain*, *getaccdotys*, *getaccfnumnz*, *getaccftrip*, *getaccgvector*, *getaccn*, *getaccname*, *getaccnamelen*, *getaccntot*, *getaccs*, *getnumacc*, *putacc*, *putaccb*, *putaccbj*, *putaccdoty*, *putacclist*, *putmaxnumacc*

Problem data - affine expressions

- *appendafes* – Appends a number of empty affine expressions to the optimization task.
- *putafebarfentry* – Inputs one entry in barF.
- *putafebarfentrylist* – Inputs a list of entries in barF.
- *putafebarfrow* – Inputs a row of barF.
- *putafefcol* – Replaces all elements in one column of the F matrix in the affine expressions.
- *putafefentry* – Replaces one entry in F.
- *putafefentrylist* – Replaces a list of entries in F.
- *putafefrow* – Replaces all elements in one row of the F matrix in the affine expressions.
- *putafefrowlist* – Replaces all elements in a number of rows of the F matrix in the affine expressions.
- *putafeg* – Replaces one element in the g vector in the affine expressions.
- *putafegslice* – Modifies a slice of the vector g.
- *Infrequent:* *emptyafebarfrow*, *emptyafebarfrowlist*, *emptyafefcol*, *emptyafefcollist*, *emptyafefrow*, *emptyafefrowlist*, *getaccbarfblocktriplet*, *getafebarfblocktriplet*, *getafebarfnumrowentries*, *getafebarfrow*, *getafebarfrowinfo*, *getafefnumnz*, *getafefrow*, *getafefrownumnz*, *getafeftrip*, *getafeg*, *getafegslice*, *getnumafe*, *putafebarfblocktriplet*, *putafeglist*, *putmaxnumafe*

Problem data - bounds

- *putconbound* – Changes the bound for one constraint.
- *putconboundslice* – Changes the bounds for a slice of the constraints.
- *putvarbound* – Changes the bounds for one variable.
- *putvarboundslice* – Changes the bounds for a slice of the variables.
- *Infrequent:* *chgconbound*, *chgvarbound*, *getconbound*, *getconboundslice*, *getvarbound*, *getvarboundslice*, *inputdata*, *putconboundlist*, *putconboundlistconst*, *putconboundsliceconst*, *putvarboundlist*, *putvarboundlistconst*, *putvarboundsliceconst*

Problem data - cones (deprecated)

- *Deprecated:* ~~*appendcone*~~, ~~*appendconeseq*~~, ~~*appendconesseq*~~, ~~*getcone*~~, ~~*getconeinfo*~~, ~~*getconename*~~, ~~*getconenameindex*~~, ~~*getconenamelen*~~, ~~*getmaxnumcone*~~, ~~*getnumcone*~~, ~~*getnumconemem*~~, ~~*putcone*~~, ~~*putconename*~~, ~~*putmaxnumcone*~~, ~~*removecones*~~

Problem data - constraints

- *appendcons* – Appends a number of constraints to the optimization task.
- *getnumcon* – Obtains the number of constraints.
- *putconbound* – Changes the bound for one constraint.
- *putconboundslice* – Changes the bounds for a slice of the constraints.
- *putconname* – Sets the name of a constraint.
- *removecons* – Removes a number of constraints.
- *Infrequent:* *chgconbound*, *getconbound*, *getconboundslice*, *getconname*, *getconnameindex*, *getconnamelen*, *getmaxnumcon*, *getnumqconknz*, *getqconk*, *inputdata*, *putconboundlist*, *putconboundlistconst*, *putconboundsliceconst*, *putmaxnumcon*

Problem data - disjunctive constraints

- *appenddjcs* – Appends a number of empty disjunctive constraints to the task.
- *putdjc* – Inputs a disjunctive constraint.
- *putdjcname* – Sets the name of a disjunctive constraint.
- *putdjcslice* – Inputs a slice of disjunctive constraints.
- *Infrequent:* *getdjcafeidxlist*, *getdjcb*, *getdjcdomainidxlist*, *getdjcname*, *getdjcnamelen*, *getdjcnumafe*, *getdjcnumafetot*, *getdjcnumdomain*, *getdjcnumdomaintot*, *getdjcnumterm*, *getdjcnumtermtot*, *getdjcs*, *getdjctermsizelist*, *getnumdjc*, *putmaxnumdjc*

Problem data - domain

- *appenddualexpconedomain* – Appends the dual exponential cone domain.
- *appenddualgeomeanconedomain* – Appends the dual geometric mean cone domain.
- *appenddualpowerconedomain* – Appends the dual power cone domain.
- *appenddualpowerconedomainseq* – Appends a sequence of dual power cone domains.
- *appendprimalexpconedomain* – Appends the primal exponential cone domain.
- *appendprimalgeomeanconedomain* – Appends the primal geometric mean cone domain.
- *appendprimalpowerconedomain* – Appends the primal power cone domain.
- *appendprimalpowerconedomainseq* – Appends a sequence of primal power cone domains.
- *appendquadraticconedomain* – Appends the n dimensional quadratic cone domain.
- *appendrdomain* – Appends the n dimensional real number domain.
- *appendrminusdomain* – Appends the n dimensional negative orthant to the list of domains.
- *appendrplusdomain* – Appends the n dimensional positive orthant to the list of domains.
- *appendrquadraticconedomain* – Appends the n dimensional rotated quadratic cone domain.
- *appendrzerodomain* – Appends the n dimensional 0 domain.
- *appendsvecpsdconedomain* – Appends the vectorized SVEC PSD cone domain.
- *putdomainname* – Sets the name of a domain.
- *Infrequent:* *getdomainn*, *getdomainname*, *getdomainnamelen*, *getdomaintype*, *getnumdomain*, *getpowerdomainalpha*, *getpowerdomaininfo*, *putmaxnumdomain*

Problem data - linear part

- *appendcons* – Appends a number of constraints to the optimization task.
- *appendvars* – Appends a number of variables to the optimization task.
- *getnumcon* – Obtains the number of constraints.
- *putacol* – Replaces all elements in one column of the linear constraint matrix.
- *putacolslice* – Replaces all elements in a sequence of columns the linear constraint matrix.
- *putaij* – Changes a single value in the linear coefficient matrix.
- *putaijlist* – Changes one or more coefficients in the linear constraint matrix.
- *putarow* – Replaces all elements in one row of the linear constraint matrix.
- *putarowslice* – Replaces all elements in several rows the linear constraint matrix.
- *putcfix* – Replaces the fixed term in the objective.
- *putcj* – Modifies one linear coefficient in the objective.
- *putconbound* – Changes the bound for one constraint.
- *putconboundslice* – Changes the bounds for a slice of the constraints.
- *putconname* – Sets the name of a constraint.
- *putcslice* – Modifies a slice of the linear objective coefficients.

- *putobjname* – Assigns a new name to the objective.
- *putobjsense* – Sets the objective sense.
- *putvarbound* – Changes the bounds for one variable.
- *putvarboundslice* – Changes the bounds for a slice of the variables.
- *putvarname* – Sets the name of a variable.
- *removecons* – Removes a number of constraints.
- *removevars* – Removes a number of variables.
- *Infrequent:* *chgconbound*, *chgvarbound*, *getacol*, *getacolnumnz*, *getacolslice*, *getacolslicenumnz*, *getacolslicetrip*, *getaij*, *getapiecenumnz*, *getarow*, *getarownumnz*, *getarowslice*, *getarowslicenumnz*, *getarowslicetrip*, *getatrip*, *getatruncatetol*, *getc*, *getcfix*, *getcj*, *getclist*, *getconbound*, *getconboundslice*, *getconname*, *getconnameindex*, *getconnamelen*, *getcslice*, *getmaxnumanz*, *getmaxnumcon*, *getmaxnumvar*, *getnumanz*, *getobjsense*, *getvarbound*, *getvarboundslice*, *getvarname*, *getvarnameindex*, *getvarnamelen*, *inputdata*, *putacollist*, *putarowlist*, *putatruncatetol*, *putclist*, *putconboundlist*, *putconboundlistconst*, *putconboundsliceconst*, *putmaxnumanz*, *putvarboundlist*, *putvarboundlistconst*, *putvarboundsliceconst*

Problem data - objective

- *putbarcj* – Changes one element in barc.
- *putcfix* – Replaces the fixed term in the objective.
- *putcj* – Modifies one linear coefficient in the objective.
- *putcslice* – Modifies a slice of the linear objective coefficients.
- *putobjname* – Assigns a new name to the objective.
- *putobjsense* – Sets the objective sense.
- *putqobj* – Replaces all quadratic terms in the objective.
- *putqobjij* – Replaces one coefficient in the quadratic term in the objective.
- *Infrequent:* *putclist*

Problem data - quadratic part

- *putqcon* – Replaces all quadratic terms in constraints.
- *putqconk* – Replaces all quadratic terms in a single constraint.
- *putqobj* – Replaces all quadratic terms in the objective.
- *putqobjij* – Replaces one coefficient in the quadratic term in the objective.
- *Infrequent:* *getmaxnumqnz*, *getnumqconknz*, *getnumqobjnz*, *getqconk*, *getqobj*, *getqobjij*, *putmaxnumqnz*

Problem data - semidefinite

- *appendbarvars* – Appends semidefinite variables to the problem.
- *appendsparsesymmat* – Appends a general sparse symmetric matrix to the storage of symmetric matrices.
- *appendsparsesymmatlist* – Appends a general sparse symmetric matrix to the storage of symmetric matrices.
- *putafebarfentry* – Inputs one entry in barF.
- *putafebarfentrylist* – Inputs a list of entries in barF.
- *putafebarfrow* – Inputs a row of barF.
- *putbaraij* – Inputs an element of barA.
- *putbaraijlist* – Inputs list of elements of barA.
- *putbararowlist* – Replace a set of rows of barA
- *putbarcj* – Changes one element in barc.
- *putbarvarname* – Sets the name of a semidefinite variable.
- *Infrequent:* *emptyafebarfrow*, *emptyafebarfrowlist*, *getaccbarfblocktriplet*, *getaccbarfnumblocktriplets*, *getafebarfblocktriplet*, *getafebarfnumblocktriplets*, *getafebarfnumrowentries*, *getafebarfrow*, *getafebarfrowinfo*, *getbarablocktriplet*, *getbaraidx*, *getbaraidxij*, *getbaraidxinfo*, *getbarasparsity*, *getbarcblocktriplet*, *getbarcidx*, *getbarcidxinfo*, *getbarcidxj*, *getbarcsparsity*, *getdimbarvarj*, *getlenbarvarj*, *getmaxnumbarvar*, *getnumbarablocktriplets*, *getnumbaranz*, *getnumbarcblocktriplets*, *getnumbarcnz*, *getnumbarvar*, *getnumsymmat*, *getsparsesymmat*, *getsymmatinfo*, *putafebarfblocktriplet*, *putbarablocktriplet*, *putbarcblocktriplet*, *putmaxnumbarvar*, *removebarvars*

Problem data - variables

- *appendvars* – Appends a number of variables to the optimization task.
- *getnumvar* – Obtains the number of variables.
- *putvarbound* – Changes the bounds for one variable.
- *putvarboundslice* – Changes the bounds for a slice of the variables.
- *putvarname* – Sets the name of a variable.
- *putvartype* – Sets the variable type of one variable.
- *removevars* – Removes a number of variables.
- *Infrequent:* *chgvarbound*, *getc*, *getcj*, *getmaxnumvar*, *getnumintvar*, *getvarbound*, *getvarboundslice*, *getvarname*, *getvarnameindex*, *getvarnamelen*, *getvartype*, *getvartypelist*, *putclist*, *putmaxnumvar*, *putvarboundlist*, *putvarboundlistconst*, *putvarboundsliceconst*, *putvartypelist*

Remote optimization

- *asyncgetlog* – Get the optimizer log from a remote job.
- *asyncgetresult* – Request a solution from a remote job.
- *asyncoptimize* – Offload the optimization task to a solver server in asynchronous mode.
- *asyncpoll* – Requests information about the status of the remote job.
- *asyncstop* – Request that the job identified by the token is terminated.
- *optimizermt* – Offload the optimization task to a solver server and wait for the solution.
- *putoptserverhost* – Specify an OptServer for remote calls.

Responses, errors and warnings

- *getcodedesc* – Obtains a short description of a response code.
- *Infrequent:* *getlasterror*

Sensitivity analysis

- *dualsensitivity* – Performs sensitivity analysis on objective coefficients.
- *primalsensitivity* – Perform sensitivity analysis on bounds.
- *sensitivityreport* – Creates a sensitivity report.

Solution - dual

- *getaccdoty* – Obtains the doty vector for an affine conic constraint.
- *getdualobj* – Computes the dual objective value associated with the solution.
- *gety* – Obtains the y vector for a solution.
- *getyslice* – Obtains a slice of the y vector for a solution.
- *Infrequent:* *getaccdotys*, *getreducedcosts*, *getslc*, *getslcslice*, *getslx*, *getslxslice*, *getsnx*, *getsnxslice*, *getsolution*, *getsolutionnew*, *getsolutionslice*, *getsuc*, *getsucslice*, *getsux*, *getsuxslice*, *putaccdoty*, *putconsolutioni*, *putslc*, *putslcslice*, *putslx*, *putslxslice*, *putsnx*, *putsnxslice*, *putsolution*, *putsolutionnew*, *putsolutionyi*, *putsuc*, *putsucslice*, *putsux*, *putsuxslice*, *putvarsolutionj*, *putyslice*

Solution - primal

- *getprimalobj* – Computes the primal objective value for the desired solution.
- *getxx* – Obtains the xx vector for a solution.
- *getxxslice* – Obtains a slice of the xx vector for a solution.
- *putxx* – Sets the xx vector for a solution.
- *putxxslice* – Sets a slice of the xx vector for a solution.
- *Infrequent:* *evaluateacc*, *evaluateaccs*, *getsolution*, *getsolutionnew*, *getsolutionslice*, *getxc*, *getxcslice*, *putconsolutioni*, *putsolution*, *putsolutionnew*, *putvarsolutionj*, *putxc*, *putxcslice*, *puty*

Solution - semidefinite

- *getbarsj* – Obtains the dual solution for a semidefinite variable.
- *getbarsslice* – Obtains the dual solution for a sequence of semidefinite variables.
- *getbarxj* – Obtains the primal solution for a semidefinite variable.
- *getbarxslice* – Obtains the primal solution for a sequence of semidefinite variables.
- *Infrequent:* *putbarsj*, *putbarxj*

Solution information

- *getdualobj* – Computes the dual objective value associated with the solution.
- *getprimalobj* – Computes the primal objective value for the desired solution.
- *getprosta* – Obtains the problem status.
- *getpviolcon* – Computes the violation of a primal solution associated to a constraint.
- *getpviolvar* – Computes the violation of a primal solution for a list of scalar variables.
- *getsolsta* – Obtains the solution status.
- *getsolutioninfo* – Obtains information about of a solution.
- *getsolutioninfonew* – Obtains information about of a solution.
- *onesolutionsummary* – Prints a short summary of a specified solution.
- *solutiondef* – Checks whether a solution is defined.
- *solutionsummary* – Prints a short summary of the current solutions.
- *Infrequent:* *analyzesolution*, *deletesolution*, *getdualsolutionnorms*, *getdviolacc*, *getdviolbarvar*, *getdviolcon*, *getdviolvar*, *getprimalsolutionnorms*, *getpviolacc*, *getpviolbarvar*, *getpvioldjc*, *getskc*, *getskcslice*, *getskn*, *getskx*, *getskxslice*, *getsolution*, *getsolutionnew*, *getsolutionslice*, *prostatostr*, *putconsolutioni*, *putskc*, *putskcslice*, *putskx*, *putskxslice*, *putsolution*, *putsolutionnew*, *putsolutionyi*, *putvarsolutionj*
- *Deprecated:* ~~*getdviolcones*~~, ~~*getpviolcones*~~

Solving systems with basis matrix

- *Infrequent:* *basiscond*, *initbasissolve*, *solvewithbasis*

System, memory and debugging

- *Infrequent:* *checkmem*, *getmemusage*

Versions

- *getversion* – Obtains MOSEK version information.

15.3 Functions in alphabetical order

`analyzenames`

```
function analyzenames(task::MSKtask,
                      whichstream::Streamtype,
                      nametype::Nametype)
```

The function analyzes the names and issues an error if a name is invalid.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*Streamtype*) – Index of the stream. (input)
- `nametype` (*Nametype*) – The type of names e.g. valid in MPS or LP files. (input)

Groups

Names

`analyzeproblem`

```
function analyzeproblem(task::MSKtask,
                        whichstream::Streamtype)
```

The function analyzes the data of a task and writes out a report.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*Streamtype*) – Index of the stream. (input)

Groups

Inspecting the task

`analyzesolution`

```
function analyzesolution(task::MSKtask,
                         whichstream::Streamtype,
                         whichsol::Soltype)
```

Print information related to the quality of the solution and other solution statistics.

By default this function prints information about the largest infeasibilites in the solution, the primal (and possibly dual) objective value and the solution status.

Following parameters can be used to configure the printed statistics:

- *MSK_IPAR_ANA_SOL_BASIS* enables or disables printing of statistics specific to the basis solution (condition number, number of basic variables etc.). Default is on.
- *MSK_IPAR_ANA_SOL_PRINT_VIOLATED* enables or disables listing names of all constraints (both primal and dual) which are violated by the solution. Default is off.
- *MSK_DPAR_ANA_SOL_INFEAS_TOL* is the tolerance defining when a constraint is considered violated. If a constraint is violated more than this, it will be listed in the summary.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*Streamtype*) – Index of the stream. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)

Groups

Solution information, *Inspecting the task*

`appendacc`

```
function appendacc(task::MSKtask,
                   domidx::Int64,
                   afeidxlist::Vector{Int64},
                   b::Union{Nothing,Vector{Float64}})
function appendacc(task::MSKtask,
                   domidx::T0,
                   afeidxlist::T1,
                   b::T2)
    where { T0<:Integer,
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
```

Appends an affine conic constraint to the task. The affine constraint has the form *a sequence of affine expressions belongs to a domain*.

The domain index is specified with `domidx` and should refer to a domain previously appended with one of the `append...domain` functions.

The length of the affine expression list `afeidxlist` must be equal to the dimension n of the domain. The elements of `afeidxlist` are indexes to the store of affine expressions, i.e. the affine expressions appearing in the affine conic constraint are:

$$F_{\texttt{afeidxlist}[k],:}x + g_{\texttt{afeidxlist}[k]} \quad \text{for } k = 1, \ldots, n.$$

If an optional vector `b` of the same length as `afeidxlist` is specified then the expressions appearing in the affine constraint will instead be taken as:

$$F_{\texttt{afeidxlist}[k],:}x + g_{\texttt{afeidxlist}[k]} - b_k \quad \text{for } k = 1, \ldots, n.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidx` (`Int64`) – Domain index. (input)
- `afeidxlist` (`Int64[]`) – List of affine expression indexes. (input)
- `b` (`Float64[]`) – The vector of constant terms modifying affine expressions. Optional, can be `nothing` if not required. (input)

Groups

Problem data - affine conic constraints

`appendaccs`

```
function appendaccs(task::MSKtask,
                    domidxs::Vector{Int64},
                    afeidxlist::Vector{Int64},
                    b::Union{Nothing,Vector{Float64}})
function appendaccs(task::MSKtask,
                    domidxs::T0,
                    afeidxlist::T1,
                    b::T2)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
```

Appends `numaccs` affine conic constraint to the task. Each single affine conic constraint should be specified as in *`appendacc`* and the input of this function should contain the concatenation of all these descriptions.

In particular, the length of `afeidxlist` must equal the sum of dimensions of domains indexed in `domainsidxs`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidxs` (`Int64[]`) – Domain indices. (input)
- `afeidxlist` (`Int64[]`) – List of affine expression indexes. (input)
- `b` (`Float64[]`) – The vector of constant terms modifying affine expressions. Optional, can be `nothing` if not required. (input)

Groups

Problem data - affine conic constraints

`appendaccseq`

```
function appendaccseq(task::MSKtask,
                      domidx::Int64,
                      afeidxfirst::Int64,
                      b::Union{Nothing,Vector{Float64}})
function appendaccseq(task::MSKtask,
                      domidx::T0,
                      afeidxfirst::T1,
                      b::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Appends an affine conic constraint to the task, as in *appendacc*. The function assumes the affine expressions forming the constraint are sequential. The affine constraint has the form *a sequence of affine expressions belongs to a domain*.

The domain index is specified with `domidx` and should refer to a domain previously appended with one of the `append...domain` functions.

The number of affine expressions should be equal to the dimension n of the domain. The affine expressions forming the affine constraint are arranged sequentially in a contiguous block of the affine expression store starting from position `afeidxfirst`. That is, the affine expressions appearing in the affine conic constraint are:

$$F_{\texttt{afeidxfirst}+k,:}x + g_{\texttt{afeidxfirst}+k} \quad \text{for } k = 1, \ldots, n.$$

If an optional vector `b` of length `numafeidx` is specified then the expressions appearing in the affine constraint will instead be taken as

$$F_{\texttt{afeidxfirst}+k,:}x + g_{\texttt{afeidxfirst}+k} - b_k \quad \text{for } k = 1, \ldots, n.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidx` (`Int64`) – Domain index. (input)
- `afeidxfirst` (`Int64`) – Index of the first affine expression. (input)
- `b` (`Float64[]`) – The vector of constant terms modifying affine expressions. Optional, can be `nothing` if not required. (input)

Groups

Problem data - affine conic constraints

`appendaccsseq`

```
function appendaccsseq(task::MSKtask,
                       domidxs::Vector{Int64},
                       numafeidx::Int64,
                       afeidxfirst::Int64,
                       b::Union{Nothing,Vector{Float64}})
function appendaccsseq(task::MSKtask,
```

(continues on next page)

(continued from previous page)

```
                        domidxs::T0,
                        numafeidx::T1,
                        afeidxfirst::T2,
                        b::T3)
    where { T0<:AbstractVector{<:Integer},
            T1<:Integer,
            T2<:Integer,
            T3<:AbstractVector{<:Number} }
```

Appends `numaccs` affine conic constraint to the task. It is the block variant of *appendaccs*, that is it assumes that the affine expressions appearing in the affine conic constraints are sequential in the affine expression store, starting from position `afeidxfirst`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidxs` (`Int64[]`) – Domain indices. (input)
- `numafeidx` (`Int64`) – Number of affine expressions in the affine expression list (must equal the sum of dimensions of the domains). (input)
- `afeidxfirst` (`Int64`) – Index of the first affine expression. (input)
- `b` (`Float64[]`) – The vector of constant terms modifying affine expressions. Optional, can be `nothing` if not required. (input)

Groups

Problem data - affine conic constraints

`appendafes`

```
function appendafes(task::MSKtask,
                    num::Int64)
function appendafes(task::MSKtask,
                    num::T0)
    where { T0<:Integer }
```

Appends a number of empty affine expressions to the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `num` (`Int64`) – Number of empty affine expressions which should be appended. (input)

Groups

Problem data - affine expressions

`appendbarvars`

```
function appendbarvars(task::MSKtask,
                       dim::Vector{Int32})
function appendbarvars(task::MSKtask,
                       dim::T0)
    where { T0<:AbstractVector{<:Integer} }
```

Appends positive semidefinite matrix variables of dimensions given by `dim` to the problem.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `dim` (`Int32[]`) – Dimensions of symmetric matrix variables to be added. (input)

Groups

Problem data - semidefinite

~~appendcone~~ *Deprecated*

```
function appendcone(task::MSKtask,
                    ct::Conetype,
                    conepar::Float64,
                    submem::Vector{Int32})
function appendcone(task::MSKtask,
                    ct::Conetype,
                    conepar::T0,
                    submem::T1)
    where { T0<:Number,
            T1<:AbstractVector{<:Integer} }
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Appends a new conic constraint to the problem. Hence, add a constraint

$$\hat{x} \in \mathcal{K}$$

to the problem, where $\mathcal{K}$ is a convex cone. $\hat{x}$ is a subset of the variables which will be specified by the argument `submem`. Cone type is specified by `ct`.

Define

$$\hat{x} = x_{\texttt{submem}[1]}, \ldots, x_{\texttt{submem}[\texttt{nummem}]}.$$

Depending on the value of `ct` this function appends one of the constraints:

- Quadratic cone (*MSK_CT_QUAD*, requires $\texttt{nummem} \geq 1$):

$$\hat{x}_0 \geq \sqrt{\sum_{i=1}^{i<\texttt{nummem}} \hat{x}_i^2}$$

- Rotated quadratic cone (*MSK_CT_RQUAD*, requires $\texttt{nummem} \geq 2$):

$$2\hat{x}_0\hat{x}_1 \geq \sum_{i=2}^{i<\texttt{nummem}} \hat{x}_i^2, \quad \hat{x}_0, \hat{x}_1 \geq 0$$

- Primal exponential cone (*MSK_CT_PEXP*, requires $\texttt{nummem} = 3$):

$$\hat{x}_0 \geq \hat{x}_1 \exp(\hat{x}_2/\hat{x}_1), \quad \hat{x}_0, \hat{x}_1 \geq 0$$

- Primal power cone (*MSK_CT_PPOW*, requires $\texttt{nummem} \geq 2$):

$$\hat{x}_0^{\alpha}\hat{x}_1^{1-\alpha} \geq \sqrt{\sum_{i=2}^{i<\texttt{nummem}} \hat{x}_i^2}, \quad \hat{x}_0, \hat{x}_1 \geq 0$$

  where α is the cone parameter specified by `conepar`.

- Dual exponential cone (*MSK_CT_DEXP*, requires $\texttt{nummem} = 3$):

$$\hat{x}_0 \geq -\hat{x}_2 e^{-1} \exp(\hat{x}_1/\hat{x}_2), \quad \hat{x}_2 \leq 0, \hat{x}_0 \geq 0$$

- Dual power cone (*MSK_CT_DPOW*, requires $\texttt{nummem} \geq 2$):

$$\left(\frac{\hat{x}_0}{\alpha}\right)^{\alpha} \left(\frac{\hat{x}_1}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{i=2}^{i<\texttt{nummem}} \hat{x}_i^2}, \quad \hat{x}_0, \hat{x}_1 \geq 0$$

  where α is the cone parameter specified by `conepar`.

- Zero cone (*MSK_CT_ZERO*):

$$\hat{x}_i = 0 \text{ for all } i$$

Please note that the sets of variables appearing in different conic constraints must be disjoint.

For an explained code example see Sec. 6.3, Sec. 6.5 or Sec. 6.4.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `ct` (*Conetype*) – Specifies the type of the cone. (input)
- `conepar` (`Float64`) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0. (input)
- `submem` (`Int32[]`) – Variable subscripts of the members in the cone. (input)

Groups

Problem data - cones (deprecated)

~~`appendconeseq`~~ *Deprecated*

```
function appendconeseq(task::MSKtask,
                       ct::Conetype,
                       conepar::Float64,
                       nummem::Int32,
                       j::Int32)
function appendconeseq(task::MSKtask,
                       ct::Conetype,
                       conepar::T0,
                       nummem::T1,
                       j::T2)
    where { T0<:Number,
            T1<:Integer,
            T2<:Integer }
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Appends a new conic constraint to the problem, as in ~~*appendcone*~~. The function assumes the members of cone are sequential where the first member has index `j` and the last `j+nummem-1`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `ct` (*Conetype*) – Specifies the type of the cone. (input)
- `conepar` (`Float64`) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0. (input)
- `nummem` (`Int32`) – Number of member variables in the cone. (input)
- `j` (`Int32`) – Index of the first variable in the conic constraint. (input)

Groups

Problem data - cones (deprecated)

~~`appendconesseq`~~ *Deprecated*

```
function appendconesseq(task::MSKtask,
                        ct::Vector{Conetype},
                        conepar::Vector{Float64},
                        nummem::Vector{Int32},
                        j::Int32)
function appendconesseq(task::MSKtask,
```

(continues on next page)

(continued from previous page)

```
                        ct::Vector{Conetype},
                        conepar::T0,
                        nummem::T1,
                        j::T2)
    where { T0<:AbstractVector{<:Number},
            T1<:AbstractVector{<:Integer},
            T2<:Integer }
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Appends a number of conic constraints to the problem, as in ***appendcone***. The kth cone is assumed to be of dimension `nummem[k]`. Moreover, it is assumed that the first variable of the first cone has index j and starting from there the sequentially following variables belong to the first cone, then to the second cone and so on.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `ct` (*`Conetype`*`[]`) – Specifies the type of the cone. (input)
- `conepar` (`Float64[]`) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0. (input)
- `nummem` (`Int32[]`) – Numbers of member variables in the cones. (input)
- `j` (`Int32`) – Index of the first variable in the first cone to be appended. (input)

Groups

Problem data - cones (deprecated)

`appendcons`

```
function appendcons(task::MSKtask,
                    num::Int32)
function appendcons(task::MSKtask,
                    num::T0)
    where { T0<:Integer }
```

Appends a number of constraints to the model. Appended constraints will be declared free. Please note that **MOSEK** will automatically expand the problem dimension to accommodate the additional constraints.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `num` (`Int32`) – Number of constraints which should be appended. (input)

Groups

Problem data - linear part, *Problem data - constraints*

`appenddjcs`

```
function appenddjcs(task::MSKtask,
                    num::Int64)
function appenddjcs(task::MSKtask,
                    num::T0)
    where { T0<:Integer }
```

Appends a number of empty disjunctive constraints to the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `num` (`Int64`) – Number of empty disjunctive constraints which should be appended. (input)

Groups

Problem data - disjunctive constraints

`appenddualexpconedomain`

```
function appenddualexpconedomain(task::MSKtask) -> domidx :: Int64
```

Appends the dual exponential cone $\left\{x \in \mathbb{R}^3 \ : \ x_0 \geq -x_2 e^{-1} e^{x_1/x_2}, \ x_0 > 0, \ x_2 < 0\right\}$ to the list of domains.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appenddualgeomeanconedomain`

```
function appenddualgeomeanconedomain(task::MSKtask,
                                     n::Int64) -> domidx :: Int64
function appenddualgeomeanconedomain(task::MSKtask,
                                     n::T0)
    where { T0<:Integer }
    -> domidx :: Int64
```

Appends the dual geometric mean cone $\left\{x \in \mathbb{R}^n \ : \ (n-1)\left(\prod_{i=0}^{n-2} x_i\right)^{1/(n-1)} \geq |x_{n-1}|, \ x_0, \ldots, x_{n-2} \geq 0\right\}$ to the list of domains.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appenddualpowerconedomain`

```
function appenddualpowerconedomain(task::MSKtask,
                                   n::Int64,
                                   alpha::Vector{Float64}) -> domidx :: Int64
function appenddualpowerconedomain(task::MSKtask,
                                   n::T0,
                                   alpha::T1)
    where { T0<:Integer,
            T1<:AbstractVector{<:Number} }
    -> domidx :: Int64
```

Appends the dual power cone domain of dimension n, with n_ℓ variables appearing on the left-hand side, where n_ℓ is the length of α, and with a homogenous sequence of exponents $\alpha_0, \ldots, \alpha_{n_\ell - 1}$. Formally, let $s = \sum_i \alpha_i$ and $\beta_i = \alpha_i / s$, so that $\sum_i \beta_i = 1$. Then the dual power cone is defined as follows:

$$\left\{x \in \mathbb{R}^n \ : \ \prod_{i=0}^{n_\ell - 1} \left(\frac{x_i}{\beta_i}\right)^{\beta_i} \geq \sqrt{\sum_{j=n_\ell}^{n-1} x_j^2}, \ x_0 \ldots, x_{n_\ell - 1} \geq 0\right\}$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain. (input)
- `alpha` (`Float64[]`) – The sequence proportional to exponents. Must be positive. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appenddualpowerconedomainseq`

```
function appenddualpowerconedomainseq(task::MSKtask,
                                      n::Vector{Int64},
                                      nleft::Vector{Int64},
                                      alpha::Vector{Float64}) -> domidxlist ::␣
↪Vector{Int64}
function appenddualpowerconedomainseq(task::MSKtask,
                                      n::T0,
                                      nleft::T1,
                                      alpha::T2)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
    -> domidxlist :: Vector{Int64}
```

Appends a sequence of dual power cone domains. The input arrays contain concatenated descriptions of individual domains, where each domain is defined as in *appenddualpowerconedomain*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64[]`) – Dimensions of the domains. (input)
- `nleft` (`Int64[]`) – Number of variables on the left hand sides. (input)
- `alpha` (`Float64[]`) – The sequences proportional to exponents, concatenated for all domains. Must be positive. (input)

Return

`domidxlist` (`Int64[]`) – Indexes of the domains.

Groups

Problem data - domain

`appendprimalexpconedomain`

```
function appendprimalexpconedomain(task::MSKtask) -> domidx :: Int64
```

Appends the primal exponential cone $\left\{x \in \mathbb{R}^3 \ : \ x_0 \geq x_1 e^{x_2/x_1}, \ x_0, x_1 > 0\right\}$ to the list of domains.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appendprimalgeomeanconedomain`

```
function appendprimalgeomeanconedomain(task::MSKtask,
                                       n::Int64) -> domidx :: Int64
function appendprimalgeomeanconedomain(task::MSKtask,
```

(continues on next page)

(continued from previous page)

```
                                        n::T0)
    where { T0<:Integer }
    -> domidx :: Int64
```

Appends the primal geometric mean cone $\left\{x \in \mathbb{R}^n \ : \ \left(\prod_{i=0}^{n-2} x_i\right)^{1/(n-1)} \geq |x_{n-1}|, \ x_0 \ldots, x_{n-2} \geq 0\right\}$ to the list of domains.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appendprimalpowerconedomain`

```
function appendprimalpowerconedomain(task::MSKtask,
                                     n::Int64,
                                     alpha::Vector{Float64}) -> domidx :: Int64
function appendprimalpowerconedomain(task::MSKtask,
                                     n::T0,
                                     alpha::T1)
    where { T0<:Integer,
            T1<:AbstractVector{<:Number} }
    -> domidx :: Int64
```

Appends the primal power cone domain of dimension n, with n_ℓ variables appearing on the left-hand side, where n_ℓ is the length of α, and with a homogenous sequence of exponents $\alpha_0, \ldots, \alpha_{n_\ell - 1}$. Formally, let $s = \sum_i \alpha_i$ and $\beta_i = \alpha_i / s$, so that $\sum_i \beta_i = 1$. Then the primal power cone is defined as follows:

$$\left\{x \in \mathbb{R}^n \ : \ \prod_{i=0}^{n_\ell - 1} x_i^{\beta_i} \geq \sqrt{\sum_{j=n_\ell}^{n-1} x_j^2}, \ x_0 \ldots, x_{n_\ell - 1} \geq 0\right\}$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain. (input)
- `alpha` (`Float64[]`) – The sequence proportional to exponents. Must be positive. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appendprimalpowerconedomainseq`

```
function appendprimalpowerconedomainseq(task::MSKtask,
                                        n::Vector{Int64},
                                        nleft::Vector{Int64},
                                        alpha::Vector{Float64}) -> domidxlist ::␣
↪Vector{Int64}
function appendprimalpowerconedomainseq(task::MSKtask,
                                        n::T0,
```

(continues on next page)

(continued from previous page)

```
                                        nleft::T1,
                                        alpha::T2)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
    -> domidxlist :: Vector{Int64}
```

Appends a sequence of primal power cone domains. The input arrays contain concatenated descriptions of individual domains, where each domain is defined as in *appendprimalpowerconedomain*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64[]`) – Dimensions of the domains. (input)
- `nleft` (`Int64[]`) – Number of variables on the left hand sides. (input)
- `alpha` (`Float64[]`) – The sequences proportional to exponents, concatenated for all domains. Must be positive. (input)

Return

`domidxlist` (`Int64[]`) – Indexes of the domains.

Groups

Problem data - domain

`appendquadraticconedomain`

```
function appendquadraticconedomain(task::MSKtask,
                                   n::Int64) -> domidx :: Int64
function appendquadraticconedomain(task::MSKtask,
                                   n::T0)
    where { T0<:Integer }
    -> domidx :: Int64
```

Appends the n-dimensional quadratic cone $\left\{x \in \mathbb{R}^n \ : \ x_0 \geq \sqrt{\sum_{i=1}^{n-1} x_i^2}\right\}$ to the list of domains.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appendrdomain`

```
function appendrdomain(task::MSKtask,
                       n::Int64) -> domidx :: Int64
function appendrdomain(task::MSKtask,
                       n::T0)
    where { T0<:Integer }
    -> domidx :: Int64
```

Appends the n-dimensional real space $\{x \in \mathbb{R}^n\}$ to the list of domains.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appendrminusdomain`

```
function appendrminusdomain(task::MSKtask,
                            n::Int64) -> domidx :: Int64
function appendrminusdomain(task::MSKtask,
                            n::T0)
    where { T0<:Integer }
    -> domidx :: Int64
```

Appends the n-dimensional negative orthant $\{x \in \mathbb{R}^n : x \leq 0\}$ to the list of domains.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appendrplusdomain`

```
function appendrplusdomain(task::MSKtask,
                           n::Int64) -> domidx :: Int64
function appendrplusdomain(task::MSKtask,
                           n::T0)
    where { T0<:Integer }
    -> domidx :: Int64
```

Appends the n-dimensional positive orthant $\{x \in \mathbb{R}^n : x \geq 0\}$ to the list of domains.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appendrquadraticconedomain`

```
function appendrquadraticconedomain(task::MSKtask,
                                    n::Int64) -> domidx :: Int64
function appendrquadraticconedomain(task::MSKtask,
                                    n::T0)
    where { T0<:Integer }
    -> domidx :: Int64
```

Appends the n-dimensional rotated quadratic cone $\left\{x \in \mathbb{R}^n \ : \ 2x_0x_1 \geq \sum_{i=2}^{n-1} x_i^2, \ x_0, x_1 \geq 0\right\}$ to the list of domains.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

appendrzerodomain

```
function appendrzerodomain(task::MSKtask,
                           n::Int64) -> domidx :: Int64
function appendrzerodomain(task::MSKtask,
                           n::T0)
    where { T0<:Integer }
    -> domidx :: Int64
```

Appends the zero in n-dimensional real space $\{x \in \mathbb{R}^n : x = 0\}$ to the list of domains.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

appendsparsesymmat

```
function appendsparsesymmat(task::MSKtask,
                            dim::Int32,
                            subi::Vector{Int32},
                            subj::Vector{Int32},
                            valij::Vector{Float64}) -> idx :: Int64
function appendsparsesymmat(task::MSKtask,
                            dim::T0,
                            subi::T1,
                            subj::T2,
                            valij::T3)
    where { T0<:Integer,
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number} }
    -> idx :: Int64
function appendsparsesymmat(task::MSKtask,
                            dim::T0,
                            data:: SparseMatrixCSC{Float64})
    where { T0<:Integer }
    -> idx :: Int64
```

MOSEK maintains a storage of symmetric data matrices that is used to build $\overline{C}$ and $\overline{A}$. The storage can be thought of as a vector of symmetric matrices denoted E. Hence, E_i is a symmetric matrix of certain dimension.

This function appends a general sparse symmetric matrix on triplet form to the vector E of symmetric matrices. The vectors `subi`, `subj`, and `valij` contains the row subscripts, column subscripts and values of each element in the symmetric matrix to be appended. Since the matrix that is appended is symmetric, only the lower triangular part should be specified. Moreover, duplicates are not allowed.

Observe the function reports the index (position) of the appended matrix in E. This index should be used for later references to the appended matrix.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `dim` (`Int32`) – Dimension of the symmetric matrix that is appended. (input)
- `subi` (`Int32[]`) – Row subscript in the triplets. (input)
- `subj` (`Int32[]`) – Column subscripts in the triplets. (input)

- `valij` (`Float64[]`) – Values of each triplet. (input)
- `data` (`SparseMatrixCSC{Float64}`) – Sparse matrix defining the column values (input)

Return

`idx` (`Int64`) – Unique index assigned to the inputted matrix that can be used for later reference.

Groups

Problem data - semidefinite

`appendsparsesymmatlist`

```
function appendsparsesymmatlist(task::MSKtask,
                                dims::Vector{Int32},
                                nz::Vector{Int64},
                                subi::Vector{Int32},
                                subj::Vector{Int32},
                                valij::Vector{Float64}) -> idx :: Vector{Int64}
function appendsparsesymmatlist(task::MSKtask,
                                dims::T0,
                                nz::T1,
                                subi::T2,
                                subj::T3,
                                valij::T4)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Number} }
    -> idx :: Vector{Int64}
```

MOSEK maintains a storage of symmetric data matrices that is used to build $\overline{C}$ and $\overline{A}$. The storage can be thought of as a vector of symmetric matrices denoted E. Hence, E_i is a symmetric matrix of certain dimension.

This function appends general sparse symmetric matrixes on triplet form to the vector E of symmetric matrices. The vectors `subi`, `subj`, and `valij` contains the row subscripts, column subscripts and values of each element in the symmetric matrix to be appended. Since the matrix that is appended is symmetric, only the lower triangular part should be specified. Moreover, duplicates are not allowed.

Observe the function reports the index (position) of the appended matrix in E. This index should be used for later references to the appended matrix.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `dims` (`Int32[]`) – Dimensions of the symmetric matrixes. (input)
- `nz` (`Int64[]`) – Number of nonzeros for each matrix. (input)
- `subi` (`Int32[]`) – Row subscript in the triplets. (input)
- `subj` (`Int32[]`) – Column subscripts in the triplets. (input)
- `valij` (`Float64[]`) – Values of each triplet. (input)

Return

`idx` (`Int64[]`) – Unique index assigned to the inputted matrix that can be used for later reference.

Groups

Problem data - semidefinite

`appendsvecpsdconedomain`

```
function appendsvecpsdconedomain(task::MSKtask,
                                 n::Int64) -> domidx :: Int64
function appendsvecpsdconedomain(task::MSKtask,
                                 n::T0)
    where { T0<:Integer }
    -> domidx :: Int64
```

Appends the domain consisting of vectors of length $n = d(d+1)/2$ defined as follows

$$\{(x_1, \ldots, x_{d(d+1)/2}) \in \mathbb{R}^n \ : \ \mathrm{sMat}(x) \in \mathcal{S}_+^d\} = \{\mathrm{sVec}(X) \ : \ X \in \mathcal{S}_+^d\},$$

where

$$\mathrm{sVec}(X) = (X_{11}, \sqrt{2}X_{21}, \ldots, \sqrt{2}X_{d1}, X_{22}, \sqrt{2}X_{32}, \ldots, X_{dd}),$$

and

$$\mathrm{sMat}(x) = \begin{bmatrix} x_1 & x_2/\sqrt{2} & \cdots & x_d/\sqrt{2} \\ x_2/\sqrt{2} & x_{d+1} & \cdots & x_{2d-1}/\sqrt{2} \\ \cdots & \cdots & \cdots & \cdots \\ x_d/\sqrt{2} & x_{2d-1}/\sqrt{2} & \cdots & x_{d(d+1)/2} \end{bmatrix}.$$

In other words, the domain consists of vectorizations of the lower-triangular part of a positive semidefinite matrix, with the non-diagonal elements additionally rescaled.

This domain is a self-dual cone.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `n` (`Int64`) – Dimension of the domain, must be of the form $d(d+1)/2$. (input)

Return

`domidx` (`Int64`) – Index of the domain.

Groups

Problem data - domain

`appendvars`

```
function appendvars(task::MSKtask,
                    num::Int32)
function appendvars(task::MSKtask,
                    num::T0)
    where { T0<:Integer }
```

Appends a number of variables to the model. Appended variables will be fixed at zero. Please note that **MOSEK** will automatically expand the problem dimension to accommodate the additional variables.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `num` (`Int32`) – Number of variables which should be appended. (input)

Groups

Problem data - linear part, *Problem data - variables*

`asyncgetlog`

```
function asyncgetlog(task::MSKtask,
                     addr::AbstractString,
                     accesstoken::Union{Nothing,AbstractString},
                     token::AbstractString)
```

Get the optimizer log from a remote job.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `addr` (`AbstractString`) – Address of the solver server (input)
- `accesstoken` (`AbstractString`) – Access token string or `nothing` if no token is given. (input)
- `token` (`AbstractString`) – Job token (input)

Groups

Remote optimization

`asyncgetresult`

```
function asyncgetresult(task::MSKtask,
                        address::AbstractString,
                        accesstoken::AbstractString,
                        token::AbstractString) -> (respavailable :: Bool,resp ::␣
↪Rescode,trm :: Rescode)
```

Request a solution from a remote job identified by the argument `token`. For other arguments see *`asyncoptimize`*. If the solution is available it will be retrieved and loaded into the local task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `address` (`AbstractString`) – Address of the OptServer. (input)
- `accesstoken` (`AbstractString`) – Access token. (input)
- `token` (`AbstractString`) – The task token. (input)

Return

- `respavailable` (`Bool`) – Indicates if a remote response is available. If this is not true, `resp` and `trm` should be ignored.
- `resp` (*`Rescode`*) – Is the response code from the remote solver.
- `trm` (*`Rescode`*) – Is either *`MSK_RES_OK`* or a termination response code.

Groups

Remote optimization

`asyncoptimize`

```
function asyncoptimize(task::MSKtask,
                       address::AbstractString,
                       accesstoken::AbstractString) -> token :: String
```

Offload the optimization task to an instance of OptServer specified by `addr`, which should be a valid URL, for example `http://server:port` or `https://server:port`. The call will exit immediately.

If the server requires authentication, the authentication token can be passed in the `accesstoken` argument.

If the server requires encryption, the keys can be passed using one of the solver parameters *`MSK_SPAR_REMOTE_TLS_CERT`* or *`MSK_SPAR_REMOTE_TLS_CERT_PATH`*.

The function returns a token which should be used in future calls to identify the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `address` (`AbstractString`) – Address of the OptServer. (input)
- `accesstoken` (`AbstractString`) – Access token. (input)

Return

`token` (`String`) – Returns the task token.

Groups

Remote optimization

asyncpoll

```
function asyncpoll(task::MSKtask,
                   address::AbstractString,
                   accesstoken::AbstractString,
                   token::AbstractString) -> (respavailable :: Bool,resp ::␣
↪Rescode,trm :: Rescode)
```

Requests information about the status of the remote job identified by the argument `token`. For other arguments see *asyncoptimize*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `address` (`AbstractString`) – Address of the OptServer. (input)
- `accesstoken` (`AbstractString`) – Access token. (input)
- `token` (`AbstractString`) – The task token. (input)

Return

- `respavailable` (`Bool`) – Indicates if a remote response is available. If this is not true, `resp` and `trm` should be ignored.
- `resp` (*Rescode*) – Is the response code from the remote solver.
- `trm` (*Rescode*) – Is either *MSK_RES_OK* or a termination response code.

Groups

Remote optimization

asyncstop

```
function asyncstop(task::MSKtask,
                   address::AbstractString,
                   accesstoken::AbstractString,
                   token::AbstractString)
```

Request that the remote job identified by `token` is terminated. For other arguments see *asyncoptimize*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `address` (`AbstractString`) – Address of the OptServer. (input)
- `accesstoken` (`AbstractString`) – Access token. (input)
- `token` (`AbstractString`) – The task token. (input)

Groups

Remote optimization

basiscond

```
function basiscond(task::MSKtask) -> (nrmbasis :: Float64,nrminvbasis :: Float64)
```

If a basic solution is available and it defines a nonsingular basis, then this function computes the 1-norm estimate of the basis matrix and a 1-norm estimate for the inverse of the basis matrix. The 1-norm estimates are computed using the method outlined in [Ste98], pp. 388-391.

By definition the 1-norm condition number of a matrix B is defined as

$$\kappa_1(B) := \|B\|_1 \|B^{-1}\|_1.$$

Moreover, the larger the condition number is the harder it is to solve linear equation systems involving B. Given estimates for $\|B\|_1$ and $\|B^{-1}\|_1$ it is also possible to estimate $\kappa_1(B)$.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `nrmbasis` (`Float64`) – An estimate for the 1-norm of the basis.
- `nrminvbasis` (`Float64`) – An estimate for the 1-norm of the inverse of the basis.

Groups

Solving systems with basis matrix

`bktostr`

```
function bktostr(task::MSKtask,
                 bk::Boundkey) -> str :: String
```

Obtains an identifier string corresponding to a bound key.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `bk` (*`Boundkey`*) – Bound key. (input)

Return

`str` (`String`) – String corresponding to the bound key code `bk`.

Groups

Names

`callbackcodetostr`

```
function callbackcodetostr(code::Callbackcode) -> callbackcodestr :: String
```

Obtains the string representation of a callback code.

Parameters

`code` (*`Callbackcode`*) – A callback code. (input)

Return

`callbackcodestr` (`String`) – String corresponding to the callback code.

Groups

Names

`checkinall`

```
function checkinall(env::MSKenv)
function checkinall()
```

Check in all unused license features to the license token server.

Parameters

`env` (`MSKenv`) – The MOSEK environment. (input)

Groups

License system

`checkinlicense`

```
function checkinlicense(env::MSKenv,
                        feature::Feature)
function checkinlicense(feature::Feature)
```

Check in a license feature to the license server. By default all licenses consumed by functions using a single environment are kept checked out for the lifetime of the **MOSEK** environment. This function checks in a given license feature back to the license server immediately.

If the given license feature is not checked out at all, or it is in use by a call to *`optimize`*, calling this function has no effect.

Please note that returning a license to the license server incurs a small overhead, so frequent calls to this function should be avoided.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)
- `feature` (*`Feature`*) – Feature to check in to the license system. (input)

Groups

License system

`checkmem`

```
function checkmem(task::MSKtask,
                  file::AbstractString,
                  line::Int32)
function checkmem(task::MSKtask,
                  file::Union{Nothing,AbstractString},
                  line::T0)
    where { T0<:Integer }
```

Checks the memory allocated by the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `file` (`AbstractString`) – File from which the function is called. (input)
- `line` (`Int32`) – Line in the file from which the function is called. (input)

Groups

System, memory and debugging

`checkoutlicense`

```
function checkoutlicense(env::MSKenv,
                         feature::Feature)
function checkoutlicense(feature::Feature)
```

Checks out a license feature from the license server. Normally the required license features will be automatically checked out the first time they are needed by the function *`optimize`*. This function can be used to check out one or more features ahead of time.

The feature will remain checked out until the environment is deleted or the function *`checkinlicense`* is called.

If a given feature is already checked out when this function is called, the call has no effect.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)
- `feature` (*`Feature`*) – Feature to check out from the license system. (input)

Groups

License system

`chgconbound`

```
function chgconbound(task::MSKtask,
                     i::Int32,
                     lower::Int32,
                     finite::Int32,
                     value::Float64)
function chgconbound(task::MSKtask,
                     i::T0,
                     lower::T1,
                     finite::T2,
                     value::T3)
    where { T0<:Integer,
```

(continues on next page)

(continued from previous page)

```
            T1<:Integer,
            T2<:Integer,
            T3<:Number }
```

Changes a bound for one constraint.

If `lower` is non-zero, then the lower bound is changed as follows:

$$\text{new lower bound} = \begin{cases} -\infty, & \texttt{finite} = 0, \\ \texttt{value} & \text{otherwise.} \end{cases}$$

Otherwise if `lower` is zero, then

$$\text{new upper bound} = \begin{cases} \infty, & \texttt{finite} = 0, \\ \texttt{value} & \text{otherwise.} \end{cases}$$

Please note that this function automatically updates the bound key for the bound, in particular, if the lower and upper bounds are identical, the bound key is changed to `fixed`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the constraint for which the bounds should be changed. (input)
- `lower` (`Int32`) – If non-zero, then the lower bound is changed, otherwise the upper bound is changed. (input)
- `finite` (`Int32`) – If non-zero, then `value` is assumed to be finite. (input)
- `value` (`Float64`) – New value for the bound. (input)

Groups

Problem data - bounds, *Problem data - constraints*, *Problem data - linear part*

`chgvarbound`

```
function chgvarbound(task::MSKtask,
                     j::Int32,
                     lower::Int32,
                     finite::Int32,
                     value::Float64)
function chgvarbound(task::MSKtask,
                     j::T0,
                     lower::T1,
                     finite::T2,
                     value::T3)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Integer,
            T3<:Number }
```

Changes a bound for one variable.

If `lower` is non-zero, then the lower bound is changed as follows:

$$\text{new lower bound} = \begin{cases} -\infty, & \texttt{finite} = 0, \\ \texttt{value} & \text{otherwise.} \end{cases}$$

Otherwise if `lower` is zero, then

$$\text{new upper bound} = \begin{cases} \infty, & \texttt{finite} = 0, \\ \texttt{value} & \text{otherwise.} \end{cases}$$

Please note that this function automatically updates the bound key for the bound, in particular, if the lower and upper bounds are identical, the bound key is changed to `fixed`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the variable for which the bounds should be changed. (input)
- `lower` (`Int32`) – If non-zero, then the lower bound is changed, otherwise the upper bound is changed. (input)
- `finite` (`Int32`) – If non-zero, then `value` is assumed to be finite. (input)
- `value` (`Float64`) – New value for the bound. (input)

Groups

Problem data - bounds, *Problem data - variables*, *Problem data - linear part*

`clearcallbackfunc`

```
function clearcallbackfunc(task::MSKtask)
```

Detaches a callback function.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Groups

Logging, *Callback*

`clearstreamfunc`

```
function clearstreamfunc(task::MSKtask,
                         whichstream::Streamtype)
```

Detaches a stream callback function.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*`streamtype`*) – Index of the stream. (input)

Groups

Logging, *Callback*

`commitchanges`

```
function commitchanges(task::MSKtask)
```

Commits all cached problem changes to the task. It is usually not necessary to call this function explicitly since changes will be committed automatically when required.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Groups

Environment and task management

`computesparsecholesky`

```
function computesparsecholesky(env::MSKenv,
                               numthreads::Int32,
                               ordermethod::Int32,
                               tolsingular::Float64,
                               anzc::Vector{Int32},
                               aptrc::Vector{Int64},
                               asubc::Vector{Int32},
                               avalc::Vector{Float64}) -> (perm :: Int32,diag ::␣
↪Float64,lnzc :: Int32,lptrc :: Int64,lensubnval :: Int64,lsubc :: Int32,lvalc :
```

(continues on next page)

(continued from previous page)

```
↪: Float64)
function computesparsecholesky(env::MSKenv,
                               numthreads::T0,
                               ordermethod::T1,
                               tolsingular::T2,
                               anzc::T3,
                               aptrc::T4,
                               asubc::T5,
                               avalc::T6)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Number,
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Integer},
            T5<:AbstractVector{<:Integer},
            T6<:AbstractVector{<:Number} }
    -> (perm :: Int32,diag :: Float64,lnzc :: Int32,lptrc :: Int64,lensubnval ::␣
↪Int64,lsubc :: Int32,lvalc :: Float64)
function computesparsecholesky(numthreads::Int32,
                               ordermethod::Int32,
                               tolsingular::Float64,
                               anzc::Vector{Int32},
                               aptrc::Vector{Int64},
                               asubc::Vector{Int32},
                               avalc::Vector{Float64}) -> (perm :: Int32,diag ::␣
↪Float64,lnzc :: Int32,lptrc :: Int64,lensubnval :: Int64,lsubc :: Int32,lvalc :
↪: Float64)
function computesparsecholesky(numthreads::T0,
                               ordermethod::T1,
                               tolsingular::T2,
                               anzc::T3,
                               aptrc::T4,
                               asubc::T5,
                               avalc::T6)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Number,
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Integer},
            T5<:AbstractVector{<:Integer},
            T6<:AbstractVector{<:Number} }
    -> (perm :: Int32,diag :: Float64,lnzc :: Int32,lptrc :: Int64,lensubnval ::␣
↪Int64,lsubc :: Int32,lvalc :: Float64)
```

The function computes a Cholesky factorization of a sparse positive semidefinite matrix. Sparsity is exploited during the computations to reduce the amount of space and work required. Both the input and output matrices are represented using the sparse format.

To be precise, given a symmetric matrix $A \in \mathbb{R}^{n \times n}$ the function computes a nonsingular lower triangular matrix L, a diagonal matrix D and a permutation matrix P such that

$$LL^T - D = PAP^T.$$

If `ordermethod` is zero then reordering heuristics are not employed and P is the identity.

If a pivot during the computation of the Cholesky factorization is less than

$$-\rho \cdot \max((PAP^T)_{jj}, 1.0)$$

then the matrix is declared negative semidefinite. On the hand if a pivot is smaller than

$$\rho \cdot \max((PAP^T)_{jj}, 1.0),$$

then D_{jj} is increased from zero to

$$\rho \cdot \max((PAP^T)_{jj}, 1.0).$$

Therefore, if A is sufficiently positive definite then D will be the zero matrix. Here ρ is set equal to value of `tolsingular`.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)
- `numthreads` (`Int32`) – The number threads that can be used to do the computation. 0 means the code makes the choice. NOTE: API change in version 10: in versions up to 9 the argument in this position indicated whether to use multithreading or not. (input)
- `ordermethod` (`Int32`) – If nonzero, then a sparsity preserving ordering will be employed. (input)
- `tolsingular` (`Float64`) – A positive parameter controlling when a pivot is declared zero. (input)
- `anzc` (`Int32[]`) – `anzc[j]` is the number of nonzeros in the j-th column of A. (input)
- `aptrc` (`Int64[]`) – `aptrc[j]` is a pointer to the first element in column j of A. (input)
- `asubc` (`Int32[]`) – Row indexes for each column stored in increasing order. (input)
- `avalc` (`Float64[]`) – The value corresponding to row indexed stored in `asubc`. (input)

Return

- `perm` (`Int32`) – Permutation array used to specify the permutation matrix P computed by the function.
- `diag` (`Float64`) – The diagonal elements of matrix D.
- `lnzc` (`Int32`) – `lnzc[j]` is the number of non zero elements in column j of L.
- `lptrc` (`Int64`) – `lptrc[j]` is a pointer to the first row index and value in column j of L.
- `lensubnval` (`Int64`) – Number of elements in `lsubc` and `lvalc`.
- `lsubc` (`Int32`) – Row indexes for each column stored in increasing order.
- `lvalc` (`Float64`) – The values corresponding to row indexed stored in `lsubc`.

Groups

Linear algebra

~~`conetypetostr`~~ *Deprecated*

```
function conetypetostr(task::MSKtask,
                       ct::Conetype) -> str :: String
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Obtains the cone string identifier corresponding to a cone type.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `ct` (*`Conetype`*) – Specifies the type of the cone. (input)

Return

`str` (`String`) – String corresponding to the cone type code `ct`.

Groups
Names

`deletesolution`

```
function deletesolution(task::MSKtask,
                        whichsol::Soltype)
```

Undefine a solution and free the memory it uses.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Groups
Environment and task management, *Solution information*

`dinfitemtostr`

```
function dinfitemtostr(item::Dinfitem) -> str :: String
```

Obtains an identifier string corresponding to a information item.

Parameters
`item` (*`Dinfitem`*) – Information item. (input)

Return
`str` (`String`) – String corresponding to the bound information item `item`.

Groups
Names

`dualsensitivity`

```
function dualsensitivity(task::MSKtask,
                         subj::Vector{Int32}) -> (leftpricej :: Vector{Float64},
↪rightpricej :: Vector{Float64},leftrangej :: Vector{Float64},rightrangej ::␣
↪Vector{Float64})
function dualsensitivity(task::MSKtask,
                         subj::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> (leftpricej :: Vector{Float64},rightpricej :: Vector{Float64},leftrangej :
↪: Vector{Float64},rightrangej :: Vector{Float64})
```

Calculates sensitivity information for objective coefficients. The indexes of the coefficients to analyze are

$$\{\texttt{subj}[i] \mid i = 1, \ldots, \texttt{numj}\}$$

The type of sensitivity analysis to perform (basis or optimal partition) is controlled by the parameter *`MSK_IPAR_SENSITIVITY_TYPE`*.

For an example, please see Section *Example: Sensitivity Analysis*.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `subj` (`Int32[]`) – Indexes of objective coefficients to analyze. (input)

Return
- `leftpricej` (`Float64[]`) – `leftpricej`$[j]$ is the left shadow price for the coefficient with index `subj`[j].
- `rightpricej` (`Float64[]`) – `rightpricej`$[j]$ is the right shadow price for the coefficient with index `subj`[j].

- `leftrangej` (`Float64[]`) – `leftrangej`[j] is the left range β_1 for the coefficient with index `subj`[j].
- `rightrangej` (`Float64[]`) – `rightrangej`[j] is the right range β_2 for the coefficient with index `subj`[j].

Groups

Sensitivity analysis

`echointro`

```
function echointro(env::MSKenv,
                   longver::Int32)
function echointro(env::MSKenv,
                   longver::T0)
    where { T0<:Integer }
function echointro(longver::Int32)
function echointro(longver::T0)
    where { T0<:Integer }
```

Prints an intro to message stream.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)
- `longver` (`Int32`) – If non-zero, then the intro is slightly longer. (input)

Groups

Logging

`emptyafebarfrow`

```
function emptyafebarfrow(task::MSKtask,
                         afeidx::Int64)
function emptyafebarfrow(task::MSKtask,
                         afeidx::T0)
    where { T0<:Integer }
```

Clears a row in $\overline{F}$ i.e. sets $\overline{F}_{\mathrm{afeidx},*} = 0$.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Row index of $\overline{F}$. (input)

Groups

Problem data - affine expressions, *Problem data - semidefinite*

`emptyafebarfrowlist`

```
function emptyafebarfrowlist(task::MSKtask,
                             afeidxlist::Vector{Int64})
function emptyafebarfrowlist(task::MSKtask,
                             afeidxlist::T0)
    where { T0<:AbstractVector{<:Integer} }
```

Clears a number of rows in $\overline{F}$ i.e. sets $\overline{F}_{i,*} = 0$ for all indices i in `afeidxlist`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidxlist` (`Int64[]`) – Indices of rows in $\overline{F}$ to clear. (input)

Groups

Problem data - affine expressions, *Problem data - semidefinite*

`emptyafefcol`

```
function emptyafefcol(task::MSKtask,
                      varidx::Int32)
function emptyafefcol(task::MSKtask,
                      varidx::T0)
    where { T0<:Integer }
```

Clears one column in the affine constraint matrix F, that is sets $F_{*,\mathtt{varidx}} = 0$.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `varidx` (`Int32`) – Index of a variable (column in F). (input)

Groups

Problem data - affine expressions

`emptyafefcollist`

```
function emptyafefcollist(task::MSKtask,
                          varidx::Vector{Int32})
function emptyafefcollist(task::MSKtask,
                          varidx::T0)
    where { T0<:AbstractVector{<:Integer} }
```

Clears a number of columns in F i.e. sets $F_{*,j} = 0$ for all indices j in `varidx`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `varidx` (`Int32[]`) – Indices of variables (columns) in F to clear. (input)

Groups

Problem data - affine expressions

`emptyafefrow`

```
function emptyafefrow(task::MSKtask,
                      afeidx::Int64)
function emptyafefrow(task::MSKtask,
                      afeidx::T0)
    where { T0<:Integer }
```

Clears one row in the affine constraint matrix F, that is sets $F_{\mathtt{afeidx},*} = 0$.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Index of a row in F. (input)

Groups

Problem data - affine expressions

`emptyafefrowlist`

```
function emptyafefrowlist(task::MSKtask,
                          afeidx::Vector{Int64})
function emptyafefrowlist(task::MSKtask,
                          afeidx::T0)
    where { T0<:AbstractVector{<:Integer} }
```

Clears a number of rows in F i.e. sets $F_{i,*} = 0$ for all indices i in `afeidx`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)

- `afeidx` (`Int64[]`) – Indices of rows in F to clear. (input)

Groups

Problem data - affine expressions

`evaluateacc`

```
function evaluateacc(task::MSKtask,
                     whichsol::Soltype,
                     accidx::Int64) -> activity :: Vector{Float64}
function evaluateacc(task::MSKtask,
                     whichsol::Soltype,
                     accidx::T0)
    where { T0<:Integer }
    -> activity :: Vector{Float64}
```

Evaluates the activity of an affine conic constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `accidx` (`Int64`) – The index of the affine conic constraint. (input)

Return

`activity` (`Float64[]`) – The activity of the affine conic constraint. The array should have length equal to the dimension of the constraint.

Groups

Solution - primal, *Problem data - affine conic constraints*

`evaluateaccs`

```
function evaluateaccs(task::MSKtask,
                      whichsol::Soltype) -> activity :: Vector{Float64}
```

Evaluates the activities of all affine conic constraints.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`activity` (`Float64[]`) – The activity of affine conic constraints. The array should have length equal to the sum of dimensions of all affine conic constraints.

Groups

Solution - primal, *Problem data - affine conic constraints*

`expirylicenses`

```
function expirylicenses(env::MSKenv) -> expiry :: Int64
function expirylicenses() -> expiry :: Int64
```

Reports when the first license feature expires. It reports the number of days to the expiry of the first feature of all the features that were ever checked out from the start of the process, or from the last call to *`resetexpirylicenses`*, until now.

Parameters

`env` (`MSKenv`) – The MOSEK environment. (input)

Return

`expiry` (`Int64`) – If nonnegative, then it is the minimum number days to expiry of any feature that has been checked out.

Groups

License system

getaccafeidxlist

```
function getaccafeidxlist(task::MSKtask,
                          accidx::Int64) -> afeidxlist :: Vector{Int64}
function getaccafeidxlist(task::MSKtask,
                          accidx::T0)
    where { T0<:Integer }
    -> afeidxlist :: Vector{Int64}
```

Obtains the list of affine expressions appearing in the affine conic constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `accidx` (`Int64`) – Index of the affine conic constraint. (input)

Return

`afeidxlist` (`Int64[]`) – List of indexes of affine expressions appearing in the constraint.

Groups

Problem data - affine conic constraints, *Inspecting the task*

getaccb

```
function getaccb(task::MSKtask,
                 accidx::Int64) -> b :: Vector{Float64}
function getaccb(task::MSKtask,
                 accidx::T0)
    where { T0<:Integer }
    -> b :: Vector{Float64}
```

Obtains the additional constant term vector appearing in the affine conic constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `accidx` (`Int64`) – Index of the affine conic constraint. (input)

Return

`b` (`Float64[]`) – The vector b appearing in the constraint.

Groups

Problem data - affine conic constraints, *Inspecting the task*

getaccbarfblocktriplet

```
function getaccbarfblocktriplet(task::MSKtask) -> (numtrip :: Int64,acc_afe ::␣
↪Vector{Int64},bar_var :: Vector{Int32},blk_row :: Vector{Int32},blk_col ::␣
↪Vector{Int32},blk_val :: Vector{Float64})
```

Obtains $\overline{F}$, implied by the ACCs, in block triplet form. If the AFEs passed to the ACCs were out of order, then this function can be used to obtain the barF as seen by the ACCs.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `numtrip` (`Int64`) – Number of elements in the block triplet form.
- `acc_afe` (`Int64[]`) – Index of the AFE within the concatenated list of AFEs in ACCs.
- `bar_var` (`Int32[]`) – Symmetric matrix variable index.
- `blk_row` (`Int32[]`) – Block row index.
- `blk_col` (`Int32[]`) – Block column index.
- `blk_val` (`Float64[]`) – The numerical value associated with each block triplet.

Groups
Problem data - affine expressions, *Problem data - semidefinite*

`getaccbarfnumblocktriplets`

```
function getaccbarfnumblocktriplets(task::MSKtask) -> numtrip :: Int64
```

Obtains an upper bound on the number of elements in the block triplet form of $\overline{F}$, as used within the ACCs.

Parameters
`task` (`MSKtask`) – An optimization task. (input)
Return
`numtrip` (`Int64`) – An upper bound on the number of elements in the block triplet form of $\overline{F}$., as used within the ACCs.
Groups
Problem data - semidefinite, *Problem data - affine conic constraints*, *Inspecting the task*

`getaccdomain`

```
function getaccdomain(task::MSKtask,
                      accidx::Int64) -> domidx :: Int64
function getaccdomain(task::MSKtask,
                      accidx::T0)
    where { T0<:Integer }
    -> domidx :: Int64
```

Obtains the domain appearing in the affine conic constraint.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `accidx` (`Int64`) – The index of the affine conic constraint. (input)

Return
`domidx` (`Int64`) – The index of domain in the affine conic constraint.
Groups
Problem data - affine conic constraints, *Inspecting the task*

`getaccdoty`

```
function getaccdoty(task::MSKtask,
                    whichsol::Soltype,
                    accidx::Int64) -> doty :: Vector{Float64}
function getaccdoty(task::MSKtask,
                    whichsol::Soltype,
                    accidx::T0)
    where { T0<:Integer }
    -> doty :: Vector{Float64}
```

Obtains the $\dot{y}$ vector for a solution (the dual values of an affine conic constraint).

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `accidx` (`Int64`) – The index of the affine conic constraint. (input)

Return
`doty` (`Float64[]`) – The dual values for this affine conic constraint. The array should have length equal to the dimension of the constraint.
Groups
Solution - dual, *Problem data - affine conic constraints*

getaccdotys

```
function getaccdotys(task::MSKtask,
                     whichsol::Soltype) -> doty :: Vector{Float64}
```

Obtains the $\dot{y}$ vector for a solution (the dual values of all affine conic constraint).

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`doty` (`Float64[]`) – The dual values of affine conic constraints. The array should have length equal to the sum of dimensions of all affine conic constraints.

Groups

Solution - dual, *Problem data - affine conic constraints*

getaccfnumnz

```
function getaccfnumnz(task::MSKtask) -> accfnnz :: Int64
```

If the AFEs are not added sequentially to the ACCs, then the present function gives the number of nonzero elements in the F matrix that would be implied by the ordering of AFEs within ACCs.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`accfnnz` (`Int64`) – Number of non-zeros in F implied by ACCs.

Groups

Problem data - affine conic constraints, *Inspecting the task*

getaccftrip

```
function getaccftrip(task::MSKtask) -> (frow :: Vector{Int64},fcol :: Vector
↪{Int32},fval :: Vector{Float64})
```

Obtains the F (that would be implied by the ordering of the AFEs within the ACCs) in triplet format.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `frow` (`Int64[]`) – Row indices of nonzeros in the implied F matrix.
- `fcol` (`Int32[]`) – Column indices of nonzeros in the implied F matrix.
- `fval` (`Float64[]`) – Values of nonzero entries in the implied F matrix.

Groups

Problem data - affine conic constraints, *Inspecting the task*

getaccgvector

```
function getaccgvector(task::MSKtask) -> g :: Vector{Float64}
```

If the AFEs are passed out of sequence to the ACCs, then this function can be used to obtain the vector g of constant terms used within the ACCs.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`g` (`Float64[]`) – The g used within the ACCs as a dense vector. The length is sum of the dimensions of the ACCs.

Groups
Inspecting the task, *Problem data - affine conic constraints*

`getaccn`

```
function getaccn(task::MSKtask,
                 accidx::Int64) -> n :: Int64
function getaccn(task::MSKtask,
                 accidx::T0)
    where { T0<:Integer }
    -> n :: Int64
```

Obtains the dimension of the affine conic constraint.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `accidx` (`Int64`) – The index of the affine conic constraint. (input)

Return
`n` (`Int64`) – The dimension of the affine conic constraint (equal to the dimension of its domain).

Groups
Problem data - affine conic constraints, *Inspecting the task*

`getaccname`

```
function getaccname(task::MSKtask,
                    accidx::Int64) -> name :: String
function getaccname(task::MSKtask,
                    accidx::T0)
    where { T0<:Integer }
    -> name :: String
```

Obtains the name of an affine conic constraint.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `accidx` (`Int64`) – Index of an affine conic constraint. (input)

Return
`name` (`String`) – Returns the required name.

Groups
Names, *Problem data - affine conic constraints*, *Inspecting the task*

`getaccnamelen`

```
function getaccnamelen(task::MSKtask,
                       accidx::Int64) -> len :: Int32
function getaccnamelen(task::MSKtask,
                       accidx::T0)
    where { T0<:Integer }
    -> len :: Int32
```

Obtains the length of the name of an affine conic constraint.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `accidx` (`Int64`) – Index of an affine conic constraint. (input)

Return
`len` (`Int32`) – Returns the length of the indicated name.

Groups
Names, *Problem data - affine conic constraints*, *Inspecting the task*

getaccntot

```
function getaccntot(task::MSKtask) -> n :: Int64
```

Obtains the total dimension of all affine conic constraints (the sum of all their dimensions).

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`n` (`Int64`) – The total dimension of all affine conic constraints.

Groups
Problem data - affine conic constraints, *Inspecting the task*

getaccs

```
function getaccs(task::MSKtask) -> (domidxlist :: Vector{Int64},afeidxlist ::␣
↪Vector{Int64},b :: Vector{Float64})
```

Obtains full data of all affine conic constraints. The output array `domainidxlist` must have at least length determined by *getnumacc*. The output arrays `afeidxlist` and `b` must have at least length determined by *getaccntot*.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
- `domidxlist` (`Int64[]`) – The list of domains appearing in all affine conic constraints.
- `afeidxlist` (`Int64[]`) – The concatenation of index lists of affine expressions appearing in all affine conic constraints.
- `b` (`Float64[]`) – The concatenation of vectors b appearing in all affine conic constraints.

Groups
Problem data - affine conic constraints, *Inspecting the task*

getacol

```
function getacol(task::MSKtask,
                 j::Int32) -> (nzj :: Int32,subj :: Vector{Int32},valj :: Vector
↪{Float64})
function getacol(task::MSKtask,
                 j::T0)
    where { T0<:Integer }
    -> (nzj :: Int32,subj :: Vector{Int32},valj :: Vector{Float64})
```

Obtains one column of A in a sparse format.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the column. (input)

Return
- `nzj` (`Int32`) – Number of non-zeros in the column obtained.
- `subj` (`Int32[]`) – Row indices of the non-zeros in the column obtained.
- `valj` (`Float64[]`) – Numerical values in the column obtained.

Groups
Problem data - linear part, *Inspecting the task*

getacolnumnz

```
function getacolnumnz(task::MSKtask,
                      i::Int32) -> nzj :: Int32
function getacolnumnz(task::MSKtask,
                      i::T0)
    where { T0<:Integer }
    -> nzj :: Int32
```

Obtains the number of non-zero elements in one column of A.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the column. (input)

Return

`nzj` (`Int32`) – Number of non-zeros in the j-th column of A.

Groups

Problem data - linear part, *Inspecting the task*

`getacolslice`

```
function getacolslice(task::MSKtask,
                      first::Int32,
                      last::Int32) -> (ptrb :: Vector{Int64},ptre :: Vector
↪{Int64},sub :: Vector{Int32},val :: Vector{Float64})
function getacolslice(task::MSKtask,
                      first::T0,
                      last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> (ptrb :: Vector{Int64},ptre :: Vector{Int64},sub :: Vector{Int32},val ::␣
↪Vector{Float64})
```

Obtains a sequence of columns from A in sparse format.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – Index of the first column in the sequence. (input)
- `last` (`Int32`) – Index of the last column in the sequence **plus one**. (input)

Return

- `ptrb` (`Int64[]`) – `ptrb[t]` is an index pointing to the first element in the t-th column obtained.
- `ptre` (`Int64[]`) – `ptre[t]` is an index pointing to the last element plus one in the t-th column obtained.
- `sub` (`Int32[]`) – Contains the row subscripts.
- `val` (`Float64[]`) – Contains the coefficient values.

Groups

Problem data - linear part, *Inspecting the task*

`getacolslicenumnz`

```
function getacolslicenumnz(task::MSKtask,
                           first::Int32,
                           last::Int32) -> numnz :: Int64
function getacolslicenumnz(task::MSKtask,
                           first::T0,
                           last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> numnz :: Int64
```

Obtains the number of non-zeros in a slice of columns of A.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – Index of the first column in the sequence. (input)
- `last` (`Int32`) – Index of the last column **plus one** in the sequence. (input)

Return

`numnz` (`Int64`) – Number of non-zeros in the slice.

Groups

Problem data - linear part, *Inspecting the task*

`getacolslicetrip`

```
function getacolslicetrip(task::MSKtask,
                          first::Int32,
                          last::Int32) -> (subi :: Vector{Int32},subj :: Vector
↪{Int32},val :: Vector{Float64})
function getacolslicetrip(task::MSKtask,
                          first::T0,
                          last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> (subi :: Vector{Int32},subj :: Vector{Int32},val :: Vector{Float64})
```

Obtains a sequence of columns from A in sparse triplet format. The function returns the content of all columns whose index `j` satisfies `first <= j < last`. The triplets corresponding to nonzero entries are stored in the arrays `subi`, `subj` and `val`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – Index of the first column in the sequence. (input)
- `last` (`Int32`) – Index of the last column in the sequence **plus one**. (input)

Return

- `subi` (`Int32[]`) – Constraint subscripts.
- `subj` (`Int32[]`) – Column subscripts.
- `val` (`Float64[]`) – Values.

Groups

Problem data - linear part, *Inspecting the task*

`getafebarfblocktriplet`

```
function getafebarfblocktriplet(task::MSKtask) -> (numtrip :: Int64,afeidx ::␣
↪Vector{Int64},barvaridx :: Vector{Int32},subk :: Vector{Int32},subl :: Vector
↪{Int32},valkl :: Vector{Float64})
```

Obtains $\overline{F}$ in block triplet form.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `numtrip` (`Int64`) – Number of elements in the block triplet form.
- `afeidx` (`Int64[]`) – Constraint index.
- `barvaridx` (`Int32[]`) – Symmetric matrix variable index.
- `subk` (`Int32[]`) – Block row index.
- `subl` (`Int32[]`) – Block column index.
- `valkl` (`Float64[]`) – The numerical value associated with each block triplet.

Groups

Problem data - affine expressions, *Problem data - semidefinite*

getafebarfnumblocktriplets

```
function getafebarfnumblocktriplets(task::MSKtask) -> numtrip :: Int64
```

Obtains an upper bound on the number of elements in the block triplet form of $\overline{F}$.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`numtrip` (`Int64`) – An upper bound on the number of elements in the block triplet form of $\overline{F}$.

Groups

Problem data - semidefinite, *Inspecting the task*

getafebarfnumrowentries

```
function getafebarfnumrowentries(task::MSKtask,
                                 afeidx::Int64) -> numentr :: Int32
function getafebarfnumrowentries(task::MSKtask,
                                 afeidx::T0)
    where { T0<:Integer }
    -> numentr :: Int32
```

Obtains the number of nonzero entries in one row of $\overline{F}$, that is the number of j such that $\overline{F}_{\mathrm{afeidx},j}$ is not the zero matrix.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Row index of $\overline{F}$. (input)

Return

`numentr` (`Int32`) – Number of nonzero entries in a row of $\overline{F}$.

Groups

Problem data - affine expressions, *Problem data - semidefinite*, *Inspecting the task*

getafebarfrow

```
function getafebarfrow(task::MSKtask,
                       afeidx::Int64) -> (barvaridx :: Vector{Int32},ptrterm ::␣
↪Vector{Int64},numterm :: Vector{Int64},termidx :: Vector{Int64},termweight ::␣
↪Vector{Float64})
function getafebarfrow(task::MSKtask,
                       afeidx::T0)
    where { T0<:Integer }
    -> (barvaridx :: Vector{Int32},ptrterm :: Vector{Int64},numterm :: Vector
↪{Int64},termidx :: Vector{Int64},termweight :: Vector{Float64})
```

Obtains all nonzero entries in one row $\overline{F}_{\mathrm{afeidx},*}$ of $\overline{F}$. For every k there is a nonzero entry $\overline{F}_{\mathrm{afeidx},\mathrm{barvaridx}[k]}$, which is represented as a weighted sum of numterm$[k]$ terms. The indices in the matrix store E and their weights for the k-th entry appear in the arrays `termidx` and `termweight` in positions

$$\mathrm{ptrterm}[k], \ldots, \mathrm{ptrterm}[k] + (\mathrm{numterm}[k] - 1).$$

The arrays should be long enough to accommodate the data; their required lengths can be obtained with *getafebarfrowinfo*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Row index of $\overline{F}$. (input)

Return

- `barvaridx` (`Int32[]`) – Semidefinite variable indices of nonzero entries in the row of $\overline{F}$.
- `ptrterm` (`Int64[]`) – Pointers to the start of each entry's description.
- `numterm` (`Int64[]`) – Number of terms in the weighted sum representation of each entry.
- `termidx` (`Int64[]`) – Indices of semidefinite matrices from the matrix store E.
- `termweight` (`Float64[]`) – Weights appearing in the weighted sum representations of all entries.

Groups

Problem data - affine expressions, *Problem data - semidefinite*, *Inspecting the task*

`getafebarfrowinfo`

```
function getafebarfrowinfo(task::MSKtask,
                           afeidx::Int64) -> (numentr :: Int32,numterm :: Int64)
function getafebarfrowinfo(task::MSKtask,
                           afeidx::T0)
    where { T0<:Integer }
    -> (numentr :: Int32,numterm :: Int64)
```

Obtains information about one row of $\overline{F}$: the number of nonzero entries, that is the number of j such that $\overline{F}_{\text{afeidx},j}$ is not the zero matrix, as well as the total number of terms in the representations of all these entries as weighted sums of matrices from E. This information provides the data sizes required for a call to *getafebarfrow*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Row index of $\overline{F}$. (input)

Return

- `numentr` (`Int32`) – Number of nonzero entries in a row of $\overline{F}$.
- `numterm` (`Int64`) – Number of terms in the weighted sums representation of the row of $\overline{F}$.

Groups

Problem data - affine expressions, *Problem data - semidefinite*, *Inspecting the task*

`getafefnumnz`

```
function getafefnumnz(task::MSKtask) -> numnz :: Int64
```

Obtains the total number of nonzeros in F.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`numnz` (`Int64`) – Number of non-zeros in F.

Groups

Problem data - affine expressions, *Inspecting the task*

`getafefrow`

```
function getafefrow(task::MSKtask,
                    afeidx::Int64) -> (numnz :: Int32,varidx :: Vector{Int32},
↪val :: Vector{Float64})
function getafefrow(task::MSKtask,
                    afeidx::T0)
    where { T0<:Integer }
    -> (numnz :: Int32,varidx :: Vector{Int32},val :: Vector{Float64})
```

Obtains one row of F in sparse format.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Index of a row in F. (input)

Return

- `numnz` (`Int32`) – Number of non-zeros in the row obtained.
- `varidx` (`Int32[]`) – Column indices of the non-zeros in the row obtained.
- `val` (`Float64[]`) – Values of the non-zeros in the row obtained.

Groups

Problem data - affine expressions, *Inspecting the task*

`getafefrownumnz`

```
function getafefrownumnz(task::MSKtask,
                         afeidx::Int64) -> numnz :: Int32
function getafefrownumnz(task::MSKtask,
                         afeidx::T0)
    where { T0<:Integer }
    -> numnz :: Int32
```

Obtains the number of nonzeros in one row of F.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Index of a row in F. (input)

Return

`numnz` (`Int32`) – Number of non-zeros in row `afeidx` of F.

Groups

Problem data - affine expressions, *Inspecting the task*

`getafeftrip`

```
function getafeftrip(task::MSKtask) -> (afeidx :: Vector{Int64},varidx :: Vector
↪{Int32},val :: Vector{Float64})
```

Obtains the F in triplet format.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `afeidx` (`Int64[]`) – Row indices of nonzeros.
- `varidx` (`Int32[]`) – Column indices of nonzeros.
- `val` (`Float64[]`) – Values of nonzero entries.

Groups

Problem data - affine expressions, *Inspecting the task*

`getafeg`

```
function getafeg(task::MSKtask,
                 afeidx::Int64) -> g :: Float64
function getafeg(task::MSKtask,
                 afeidx::T0)
    where { T0<:Integer }
    -> g :: Float64
```

Obtains a single coefficient in g.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Index of an element in g. (input)

Return

`g` (`Float64`) – The value of g_{afeidx}.

Groups

Problem data - affine expressions, *Inspecting the task*

`getafegslice`

```
function getafegslice(task::MSKtask,
                      first::Int64,
                      last::Int64) -> g :: Vector{Float64}
function getafegslice(task::MSKtask,
                      first::T0,
                      last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> g :: Vector{Float64}
```

Obtains a sequence of elements from the vector g of constant terms in the affine expressions list.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int64`) – First index in the sequence. (input)
- `last` (`Int64`) – Last index plus 1 in the sequence. (input)

Return

`g` (`Float64[]`) – The slice g as a dense vector. The length is `last-first`.

Groups

Inspecting the task, *Problem data - affine expressions*

`getaij`

```
function getaij(task::MSKtask,
                i::Int32,
                j::Int32) -> aij :: Float64
function getaij(task::MSKtask,
                i::T0,
                j::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> aij :: Float64
```

Obtains a single coefficient in A.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Row index of the coefficient to be returned. (input)
- `j` (`Int32`) – Column index of the coefficient to be returned. (input)

Return

`aij` (`Float64`) – The required coefficient $a_{i,j}$.

Groups

Problem data - linear part, *Inspecting the task*

`getapiecenumnz`

```
function getapiecenumnz(task::MSKtask,
                        firsti::Int32,
                        lasti::Int32,
                        firstj::Int32,
                        lastj::Int32) -> numnz :: Int32
function getapiecenumnz(task::MSKtask,
                        firsti::T0,
                        lasti::T1,
                        firstj::T2,
                        lastj::T3)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Integer,
            T3<:Integer }
    -> numnz :: Int32
```

Obtains the number non-zeros in a rectangular piece of A, i.e. the number of elements in the set

$$\{(i,j) \ : \ a_{i,j} \neq 0, \ \texttt{firsti} \leq i \leq \texttt{lasti} - 1, \ \texttt{firstj} \leq j \leq \texttt{lastj} - 1\}$$

This function is not an efficient way to obtain the number of non-zeros in one row or column. In that case use the function *getarownumnz* or *getacolnumnz*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `firsti` (`Int32`) – Index of the first row in the rectangular piece. (input)
- `lasti` (`Int32`) – Index of the last row plus one in the rectangular piece. (input)
- `firstj` (`Int32`) – Index of the first column in the rectangular piece. (input)
- `lastj` (`Int32`) – Index of the last column plus one in the rectangular piece. (input)

Return

`numnz` (`Int32`) – Number of non-zero A elements in the rectangular piece.

Groups

Problem data - linear part, *Inspecting the task*

`getarow`

```
function getarow(task::MSKtask,
                 i::Int32) -> (nzi :: Int32,subi :: Vector{Int32},vali :: Vector
↪{Float64})
function getarow(task::MSKtask,
                 i::T0)
    where { T0<:Integer }
    -> (nzi :: Int32,subi :: Vector{Int32},vali :: Vector{Float64})
```

Obtains one row of A in a sparse format.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the row. (input)

Return

- `nzi` (`Int32`) – Number of non-zeros in the row obtained.
- `subi` (`Int32[]`) – Column indices of the non-zeros in the row obtained.
- `vali` (`Float64[]`) – Numerical values of the row obtained.

Groups

Problem data - linear part, *Inspecting the task*

getarownumnz

```
function getarownumnz(task::MSKtask,
                      i::Int32) -> nzi :: Int32
function getarownumnz(task::MSKtask,
                      i::T0)
    where { T0<:Integer }
    -> nzi :: Int32
```

Obtains the number of non-zero elements in one row of A.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the row. (input)

Return

`nzi` (`Int32`) – Number of non-zeros in the i-th row of A.

Groups

Problem data - linear part, *Inspecting the task*

getarowslice

```
function getarowslice(task::MSKtask,
                      first::Int32,
                      last::Int32) -> (ptrb :: Vector{Int64},ptre :: Vector
↪{Int64},sub :: Vector{Int32},val :: Vector{Float64})
function getarowslice(task::MSKtask,
                      first::T0,
                      last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> (ptrb :: Vector{Int64},ptre :: Vector{Int64},sub :: Vector{Int32},val ::␣
↪Vector{Float64})
```

Obtains a sequence of rows from A in sparse format.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – Index of the first row in the sequence. (input)
- `last` (`Int32`) – Index of the last row in the sequence **plus one**. (input)

Return

- `ptrb` (`Int64[]`) – `ptrb[t]` is an index pointing to the first element in the t-th row obtained.
- `ptre` (`Int64[]`) – `ptre[t]` is an index pointing to the last element plus one in the t-th row obtained.
- `sub` (`Int32[]`) – Contains the column subscripts.
- `val` (`Float64[]`) – Contains the coefficient values.

Groups

Problem data - linear part, *Inspecting the task*

getarowslicenumnz

```
function getarowslicenumnz(task::MSKtask,
                           first::Int32,
                           last::Int32) -> numnz :: Int64
function getarowslicenumnz(task::MSKtask,
                           first::T0,
                           last::T1)
```

(continues on next page)

(continued from previous page)

```
    where { T0<:Integer,
            T1<:Integer }
    -> numnz :: Int64
```

Obtains the number of non-zeros in a slice of rows of A.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – Index of the first row in the sequence. (input)
- `last` (`Int32`) – Index of the last row **plus one** in the sequence. (input)

Return

`numnz` (`Int64`) – Number of non-zeros in the slice.

Groups

Problem data - linear part, *Inspecting the task*

`getarowslicetrip`

```
function getarowslicetrip(task::MSKtask,
                          first::Int32,
                          last::Int32) -> (subi :: Vector{Int32},subj :: Vector
↪{Int32},val :: Vector{Float64})
function getarowslicetrip(task::MSKtask,
                          first::T0,
                          last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> (subi :: Vector{Int32},subj :: Vector{Int32},val :: Vector{Float64})
```

Obtains a sequence of rows from A in sparse triplet format. The function returns the content of all rows whose index `i` satisfies `first <= i < last`. The triplets corresponding to nonzero entries are stored in the arrays `subi`, `subj` and `val`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – Index of the first row in the sequence. (input)
- `last` (`Int32`) – Index of the last row in the sequence **plus one**. (input)

Return

- `subi` (`Int32[]`) – Constraint subscripts.
- `subj` (`Int32[]`) – Column subscripts.
- `val` (`Float64[]`) – Values.

Groups

Problem data - linear part, *Inspecting the task*

`getatrip`

```
function getatrip(task::MSKtask) -> (subi :: Vector{Int32},subj :: Vector{Int32},
↪val :: Vector{Float64})
```

Obtains A in sparse triplet format. The triplets corresponding to nonzero entries are stored in the arrays `subi`, `subj` and `val`.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `subi` (`Int32[]`) – Constraint subscripts.
- `subj` (`Int32[]`) – Column subscripts.

- `val` (`Float64[]`) – Values.

Groups

Problem data - linear part, *Inspecting the task*

`getatruncatetol`

```
function getatruncatetol(task::MSKtask) -> tolzero :: Vector{Float64}
```

Obtains the tolerance value set with *putatruncatetol*.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`tolzero` (`Float64[]`) – All elements $|a_{i,j}|$ less than this tolerance is truncated to zero.

Groups

Parameters, *Problem data - linear part*

`getbarablocktriplet`

```
function getbarablocktriplet(task::MSKtask) -> (num :: Int64,subi :: Vector
↪{Int32},subj :: Vector{Int32},subk :: Vector{Int32},subl :: Vector{Int32},
↪valijkl :: Vector{Float64})
```

Obtains $\overline{A}$ in block triplet form.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `num` (`Int64`) – Number of elements in the block triplet form.
- `subi` (`Int32[]`) – Constraint index.
- `subj` (`Int32[]`) – Symmetric matrix variable index.
- `subk` (`Int32[]`) – Block row index.
- `subl` (`Int32[]`) – Block column index.
- `valijkl` (`Float64[]`) – The numerical value associated with each block triplet.

Groups

Problem data - semidefinite, *Inspecting the task*

`getbaraidx`

```
function getbaraidx(task::MSKtask,
                    idx::Int64) -> (i :: Int32,j :: Int32,num :: Int64,sub ::␣
↪Vector{Int64},weights :: Vector{Float64})
function getbaraidx(task::MSKtask,
                    idx::T0)
    where { T0<:Integer }
    -> (i :: Int32,j :: Int32,num :: Int64,sub :: Vector{Int64},weights :: Vector
↪{Float64})
```

Obtains information about an element in $\overline{A}$. Since $\overline{A}$ is a sparse matrix of symmetric matrices, only the nonzero elements in $\overline{A}$ are stored in order to save space. Now $\overline{A}$ is stored vectorized i.e. as one long vector. This function makes it possible to obtain information such as the row index and the column index of a particular element of the vectorized form of $\overline{A}$.

Please observe if one element of $\overline{A}$ is inputted multiple times then it may be stored several times in vectorized form. In that case the element with the highest index is the one that is used.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)

- `idx` (`Int64`) – Position of the element in the vectorized form. (input)

Return

- `i` (`Int32`) – Row index of the element at position `idx`.
- `j` (`Int32`) – Column index of the element at position `idx`.
- `num` (`Int64`) – Number of terms in weighted sum that forms the element.
- `sub` (`Int64[]`) – A list indexes of the elements from symmetric matrix storage that appear in the weighted sum.
- `weights` (`Float64[]`) – The weights associated with each term in the weighted sum.

Groups

Problem data - semidefinite, *Inspecting the task*

`getbaraidxij`

```
function getbaraidxij(task::MSKtask,
                      idx::Int64) -> (i :: Int32,j :: Int32)
function getbaraidxij(task::MSKtask,
                      idx::T0)
    where { T0<:Integer }
    -> (i :: Int32,j :: Int32)
```

Obtains information about an element in $\overline{A}$. Since $\overline{A}$ is a sparse matrix of symmetric matrices, only the nonzero elements in $\overline{A}$ are stored in order to save space. Now $\overline{A}$ is stored vectorized i.e. as one long vector. This function makes it possible to obtain information such as the row index and the column index of a particular element of the vectorized form of $\overline{A}$.

Please note that if one element of $\overline{A}$ is inputted multiple times then it may be stored several times in vectorized form. In that case the element with the highest index is the one that is used.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `idx` (`Int64`) – Position of the element in the vectorized form. (input)

Return

- `i` (`Int32`) – Row index of the element at position `idx`.
- `j` (`Int32`) – Column index of the element at position `idx`.

Groups

Problem data - semidefinite, *Inspecting the task*

`getbaraidxinfo`

```
function getbaraidxinfo(task::MSKtask,
                        idx::Int64) -> num :: Int64
function getbaraidxinfo(task::MSKtask,
                        idx::T0)
    where { T0<:Integer }
    -> num :: Int64
```

Each nonzero element in $\overline{A}_{ij}$ is formed as a weighted sum of symmetric matrices. Using this function the number of terms in the weighted sum can be obtained. See description of *appendsparsesymmat* for details about the weighted sum.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `idx` (`Int64`) – The internal position of the element for which information should be obtained. (input)

Return

`num` (`Int64`) – Number of terms in the weighted sum that form the specified element in $\overline{A}$.

Groups
Problem data - semidefinite, *Inspecting the task*

`getbarasparsity`

```
function getbarasparsity(task::MSKtask) -> (numnz :: Int64,idxij :: Vector{Int64}
↪)
```

The matrix $\overline{A}$ is assumed to be a sparse matrix of symmetric matrices. This implies that many of the elements in $\overline{A}$ are likely to be zero matrices. Therefore, in order to save space, only nonzero elements in $\overline{A}$ are stored on vectorized form. This function is used to obtain the sparsity pattern of $\overline{A}$ and the position of each nonzero element in the vectorized form of $\overline{A}$. From the index detailed information about each nonzero $\overline{A}_{i,j}$ can be obtained using *getbaraidxinfo* and *getbaraidx*.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
- `numnz` (`Int64`) – Number of nonzero elements in $\overline{A}$.
- `idxij` (`Int64[]`) – Position of each nonzero element in the vectorized form of $\overline{A}$.

Groups
Problem data - semidefinite, *Inspecting the task*

`getbarcblocktriplet`

```
function getbarcblocktriplet(task::MSKtask) -> (num :: Int64,subj :: Vector
↪{Int32},subk :: Vector{Int32},subl :: Vector{Int32},valjkl :: Vector{Float64})
```

Obtains $\overline{C}$ in block triplet form.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
- `num` (`Int64`) – Number of elements in the block triplet form.
- `subj` (`Int32[]`) – Symmetric matrix variable index.
- `subk` (`Int32[]`) – Block row index.
- `subl` (`Int32[]`) – Block column index.
- `valjkl` (`Float64[]`) – The numerical value associated with each block triplet.

Groups
Problem data - semidefinite, *Inspecting the task*

`getbarcidx`

```
function getbarcidx(task::MSKtask,
                    idx::Int64) -> (j :: Int32,num :: Int64,sub :: Vector{Int64},
↪weights :: Vector{Float64})
function getbarcidx(task::MSKtask,
                    idx::T0)
    where { T0<:Integer }
    -> (j :: Int32,num :: Int64,sub :: Vector{Int64},weights :: Vector{Float64})
```

Obtains information about an element in $\overline{C}$.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `idx` (`Int64`) – Index of the element for which information should be obtained. (input)

Return
- `j` (`Int32`) – Row index in $\overline{C}$.

- `num` (`Int64`) – Number of terms in the weighted sum.
- `sub` (`Int64[]`) – Elements appearing the weighted sum.
- `weights` (`Float64[]`) – Weights of terms in the weighted sum.

Groups

Problem data - semidefinite, *Inspecting the task*

`getbarcidxinfo`

```
function getbarcidxinfo(task::MSKtask,
                        idx::Int64) -> num :: Int64
function getbarcidxinfo(task::MSKtask,
                        idx::T0)
    where { T0<:Integer }
    -> num :: Int64
```

Obtains the number of terms in the weighted sum that forms a particular element in $\overline{C}$.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `idx` (`Int64`) – Index of the element for which information should be obtained. The value is an index of a symmetric sparse variable. (input)

Return

`num` (`Int64`) – Number of terms that appear in the weighted sum that forms the requested element.

Groups

Problem data - semidefinite, *Inspecting the task*

`getbarcidxj`

```
function getbarcidxj(task::MSKtask,
                     idx::Int64) -> j :: Int32
function getbarcidxj(task::MSKtask,
                     idx::T0)
    where { T0<:Integer }
    -> j :: Int32
```

Obtains the row index of an element in $\overline{C}$.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `idx` (`Int64`) – Index of the element for which information should be obtained. (input)

Return

`j` (`Int32`) – Row index in $\overline{C}$.

Groups

Problem data - semidefinite, *Inspecting the task*

`getbarcsparsity`

```
function getbarcsparsity(task::MSKtask) -> (numnz :: Int64,idxj :: Vector{Int64})
```

Internally only the nonzero elements of $\overline{C}$ are stored in a vector. This function is used to obtain the nonzero elements of $\overline{C}$ and their indexes in the internal vector representation (in `idx`). From the index detailed information about each nonzero $\overline{C}_j$ can be obtained using *getbarcidxinfo* and *getbarcidx*.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `numnz` (`Int64`) – Number of nonzero elements in $\overline{C}$.
- `idxj` (`Int64[]`) – Internal positions of the nonzeros elements in $\overline{C}$.

Groups

Problem data - semidefinite, *Inspecting the task*

`getbarsj`

```
function getbarsj(task::MSKtask,
                  whichsol::Soltype,
                  j::Int32) -> barsj :: Vector{Float64}
function getbarsj(task::MSKtask,
                  whichsol::Soltype,
                  j::T0)
    where { T0<:Integer }
    -> barsj :: Vector{Float64}
```

Obtains the dual solution for a semidefinite variable. Only the lower triangular part of $\overline{S}_j$ is returned because the matrix by construction is symmetric. The format is that the columns are stored sequentially in the natural order.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `j` (`Int32`) – Index of the semidefinite variable. (input)

Return

`barsj` (`Float64[]`) – Value of $\overline{S}_j$.

Groups

Solution - semidefinite

`getbarsslice`

```
function getbarsslice(task::MSKtask,
                      whichsol::Soltype,
                      first::Int32,
                      last::Int32,
                      slicesize::Int64) -> barsslice :: Vector{Float64}
function getbarsslice(task::MSKtask,
                      whichsol::Soltype,
                      first::T0,
                      last::T1,
                      slicesize::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Integer }
    -> barsslice :: Vector{Float64}
```

Obtains the dual solution for a sequence of semidefinite variables. The format is that matrices are stored sequentially, and in each matrix the columns are stored as in *`getbarsj`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – Index of the first semidefinite variable in the slice. (input)
- `last` (`Int32`) – Index of the last semidefinite variable in the slice plus one. (input)
- `slicesize` (`Int64`) – Denotes the length of the array `barsslice`. (input)

Return

`barsslice` (`Float64[]`) – Dual solution values of symmetric matrix variables in the slice, stored sequentially.

Groups

Solution - semidefinite

`getbarvarname`

```
function getbarvarname(task::MSKtask,
                       i::Int32) -> name :: String
function getbarvarname(task::MSKtask,
                       i::T0)
    where { T0<:Integer }
    -> name :: String
```

Obtains the name of a semidefinite variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the variable. (input)

Return

`name` (`String`) – The requested name is copied to this buffer.

Groups

Names, *Inspecting the task*

`getbarvarnameindex`

```
function getbarvarnameindex(task::MSKtask,
                            somename::AbstractString) -> (asgn :: Int32,index ::␣
↪Int32)
```

Obtains the index of semidefinite variable from its name.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `somename` (`AbstractString`) – The name of the variable. (input)

Return

- `asgn` (`Int32`) – Non-zero if the name `somename` is assigned to some semidefinite variable.
- `index` (`Int32`) – The index of a semidefinite variable with the name `somename` (if one exists).

Groups

Names, *Inspecting the task*

`getbarvarnamelen`

```
function getbarvarnamelen(task::MSKtask,
                          i::Int32) -> len :: Int32
function getbarvarnamelen(task::MSKtask,
                          i::T0)
    where { T0<:Integer }
    -> len :: Int32
```

Obtains the length of the name of a semidefinite variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the variable. (input)

Return

`len` (`Int32`) – Returns the length of the indicated name.

Groups

Names, *Inspecting the task*

getbarxj

```
function getbarxj(task::MSKtask,
                  whichsol::Soltype,
                  j::Int32) -> barxj :: Vector{Float64}
function getbarxj(task::MSKtask,
                  whichsol::Soltype,
                  j::T0)
    where { T0<:Integer }
    -> barxj :: Vector{Float64}
```

Obtains the primal solution for a semidefinite variable. Only the lower triangular part of $\overline{X}_j$ is returned because the matrix by construction is symmetric. The format is that the columns are stored sequentially in the natural order.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `j` (`Int32`) – Index of the semidefinite variable. (input)

Return

`barxj` (`Float64[]`) – Value of $\overline{X}_j$.

Groups

Solution - semidefinite

getbarxslice

```
function getbarxslice(task::MSKtask,
                      whichsol::Soltype,
                      first::Int32,
                      last::Int32,
                      slicesize::Int64) -> barxslice :: Vector{Float64}
function getbarxslice(task::MSKtask,
                      whichsol::Soltype,
                      first::T0,
                      last::T1,
                      slicesize::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Integer }
    -> barxslice :: Vector{Float64}
```

Obtains the primal solution for a sequence of semidefinite variables. The format is that matrices are stored sequentially, and in each matrix the columns are stored as in *`getbarxj`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – Index of the first semidefinite variable in the slice. (input)
- `last` (`Int32`) – Index of the last semidefinite variable in the slice plus one. (input)
- `slicesize` (`Int64`) – Denotes the length of the array `barxslice`. (input)

Return

`barxslice` (`Float64[]`) – Solution values of symmetric matrix variables in the slice, stored sequentially.

Groups

Solution - semidefinite

getc

```
function getc(task::MSKtask) -> c :: Vector{Float64}
```

Obtains all objective coefficients c.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`c` (`Float64[]`) – Linear terms of the objective as a dense vector. The length is the number of variables.

Groups
Problem data - linear part, *Inspecting the task*, *Problem data - variables*

getcfix

```
function getcfix(task::MSKtask) -> cfix :: Float64
```

Obtains the fixed term in the objective.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`cfix` (`Float64`) – Fixed term in the objective.

Groups
Problem data - linear part, *Inspecting the task*

getcj

```
function getcj(task::MSKtask,
               j::Int32) -> cj :: Float64
function getcj(task::MSKtask,
               j::T0)
    where { T0<:Integer }
    -> cj :: Float64
```

Obtains one coefficient of c.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the variable for which the c coefficient should be obtained. (input)

Return
`cj` (`Float64`) – The value of c_j.

Groups
Problem data - linear part, *Inspecting the task*, *Problem data - variables*

getclist

```
function getclist(task::MSKtask,
                  subj::Vector{Int32}) -> c :: Vector{Float64}
function getclist(task::MSKtask,
                  subj::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> c :: Vector{Float64}
```

Obtains a sequence of elements in c.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)

- `subj` (`Int32[]`) – A list of variable indexes. (input)

Return

`c` (`Float64[]`) – Linear terms of the requested list of the objective as a dense vector.

Groups

Inspecting the task, *Problem data - linear part*

`getcodedesc`

```
function getcodedesc(code::Rescode) -> (symname :: String,str :: String)
```

Obtains a short description of the meaning of the response code given by `code`.

Parameters

`code` (*`Rescode`*) – A valid **MOSEK** response code. (input)

Return

- `symname` (`String`) – Symbolic name corresponding to `code`.
- `str` (`String`) – Obtains a short description of a response code.

Groups

Names, *Responses, errors and warnings*

`getconbound`

```
function getconbound(task::MSKtask,
                     i::Int32) -> (bk :: Boundkey,bl :: Float64,bu :: Float64)
function getconbound(task::MSKtask,
                     i::T0)
    where { T0<:Integer }
    -> (bk :: Boundkey,bl :: Float64,bu :: Float64)
```

Obtains bound information for one constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the constraint for which the bound information should be obtained. (input)

Return

- `bk` (*`Boundkey`*) – Bound keys.
- `bl` (`Float64`) – Values for lower bounds.
- `bu` (`Float64`) – Values for upper bounds.

Groups

Problem data - linear part, *Inspecting the task*, *Problem data - bounds*, *Problem data - constraints*

`getconboundslice`

```
function getconboundslice(task::MSKtask,
                          first::Int32,
                          last::Int32) -> (bk :: Vector{Boundkey},bl :: Vector
↪{Float64},bu :: Vector{Float64})
function getconboundslice(task::MSKtask,
                          first::T0,
                          last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> (bk :: Vector{Boundkey},bl :: Vector{Float64},bu :: Vector{Float64})
```

Obtains bounds information for a slice of the constraints.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

- `bk` (*`Boundkey`*`[]`) – Bound keys.
- `bl` (`Float64[]`) – Values for lower bounds.
- `bu` (`Float64[]`) – Values for upper bounds.

Groups

Problem data - linear part, *Inspecting the task*, *Problem data - bounds*, *Problem data - constraints*

~~`getcone`~~ *Deprecated*

```
function getcone(task::MSKtask,
                 k::Int32) -> (ct :: Conetype,conepar :: Float64,nummem :: Int32,
↪submem :: Vector{Int32})
function getcone(task::MSKtask,
                 k::T0)
    where { T0<:Integer }
    -> (ct :: Conetype,conepar :: Float64,nummem :: Int32,submem :: Vector{Int32}
↪)
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `k` (`Int32`) – Index of the cone. (input)

Return

- `ct` (*`Conetype`*) – Specifies the type of the cone.
- `conepar` (`Float64`) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0.
- `nummem` (`Int32`) – Number of member variables in the cone.
- `submem` (`Int32[]`) – Variable subscripts of the members in the cone.

Groups

Inspecting the task, *Problem data - cones (deprecated)*

~~`getconeinfo`~~ *Deprecated*

```
function getconeinfo(task::MSKtask,
                     k::Int32) -> (ct :: Conetype,conepar :: Float64,nummem ::␣
↪Int32)
function getconeinfo(task::MSKtask,
                     k::T0)
    where { T0<:Integer }
    -> (ct :: Conetype,conepar :: Float64,nummem :: Int32)
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `k` (`Int32`) – Index of the cone. (input)

Return

- `ct` (*`Conetype`*) – Specifies the type of the cone.

- `conepar` (`Float64`) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0.
- `nummem` (`Int32`) – Number of member variables in the cone.

Groups

Inspecting the task, *Problem data - cones (deprecated)*

~~`getconename`~~ *Deprecated*

```
function getconename(task::MSKtask,
                     i::Int32) -> name :: String
function getconename(task::MSKtask,
                     i::T0)
    where { T0<:Integer }
    -> name :: String
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the cone. (input)

Return

`name` (`String`) – The required name.

Groups

Names, *Problem data - cones (deprecated)*, *Inspecting the task*

~~`getconenameindex`~~ *Deprecated*

```
function getconenameindex(task::MSKtask,
                          somename::AbstractString) -> (asgn :: Int32,index ::␣
↪Int32)
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Checks whether the name `somename` has been assigned to any cone. If it has been assigned to a cone, then the index of the cone is reported.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `somename` (`AbstractString`) – The name which should be checked. (input)

Return

- `asgn` (`Int32`) – Is non-zero if the name `somename` is assigned to some cone.
- `index` (`Int32`) – If the name `somename` is assigned to some cone, then `index` is the index of the cone.

Groups

Names, *Problem data - cones (deprecated)*, *Inspecting the task*

~~`getconenamelen`~~ *Deprecated*

```
function getconenamelen(task::MSKtask,
                        i::Int32) -> len :: Int32
function getconenamelen(task::MSKtask,
                        i::T0)
    where { T0<:Integer }
    -> len :: Int32
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the cone. (input)

Return

`len` (`Int32`) – Returns the length of the indicated name.

Groups

Names, *Problem data - cones (deprecated)*, *Inspecting the task*

`getconname`

```
function getconname(task::MSKtask,
                    i::Int32) -> name :: String
function getconname(task::MSKtask,
                    i::T0)
    where { T0<:Integer }
    -> name :: String
```

Obtains the name of a constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the constraint. (input)

Return

`name` (`String`) – The required name.

Groups

Names, *Problem data - linear part*, *Problem data - constraints*, *Inspecting the task*

`getconnameindex`

```
function getconnameindex(task::MSKtask,
                         somename::AbstractString) -> (asgn :: Int32,index ::␣
↪Int32)
```

Checks whether the name `somename` has been assigned to any constraint. If so, the index of the constraint is reported.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `somename` (`AbstractString`) – The name which should be checked. (input)

Return

- `asgn` (`Int32`) – Is non-zero if the name `somename` is assigned to some constraint.
- `index` (`Int32`) – If the name `somename` is assigned to a constraint, then `index` is the index of the constraint.

Groups

Names, *Problem data - linear part*, *Problem data - constraints*, *Inspecting the task*

`getconnamelen`

```
function getconnamelen(task::MSKtask,
                       i::Int32) -> len :: Int32
function getconnamelen(task::MSKtask,
                       i::T0)
    where { T0<:Integer }
    -> len :: Int32
```

Obtains the length of the name of a constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the constraint. (input)

Return

`len` (`Int32`) – Returns the length of the indicated name.

Groups

Names, *Problem data - linear part*, *Problem data - constraints*, *Inspecting the task*

`getcslice`

```
function getcslice(task::MSKtask,
                   first::Int32,
                   last::Int32) -> c :: Vector{Float64}
function getcslice(task::MSKtask,
                   first::T0,
                   last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> c :: Vector{Float64}
```

Obtains a sequence of elements in c.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`c` (`Float64[]`) – Linear terms of the requested slice of the objective as a dense vector. The length is `last-first`.

Groups

Inspecting the task, *Problem data - linear part*

`getdimbarvarj`

```
function getdimbarvarj(task::MSKtask,
                       j::Int32) -> dimbarvarj :: Int32
function getdimbarvarj(task::MSKtask,
                       j::T0)
    where { T0<:Integer }
    -> dimbarvarj :: Int32
```

Obtains the dimension of a symmetric matrix variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the semidefinite variable whose dimension is requested. (input)

Return

`dimbarvarj` (`Int32`) – The dimension of the j-th semidefinite variable.

Groups

Inspecting the task, *Problem data - semidefinite*

`getdjcafeidxlist`

```
function getdjcafeidxlist(task::MSKtask,
                          djcidx::Int64) -> afeidxlist :: Vector{Int64}
function getdjcafeidxlist(task::MSKtask,
                          djcidx::T0)
    where { T0<:Integer }
    -> afeidxlist :: Vector{Int64}
```

Obtains the list of affine expression indexes in a disjunctive constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of the disjunctive constraint. (input)

Return

`afeidxlist` (`Int64[]`) – List of affine expression indexes.

Groups

Problem data - disjunctive constraints, *Inspecting the task*

`getdjcb`

```
function getdjcb(task::MSKtask,
                 djcidx::Int64) -> b :: Vector{Float64}
function getdjcb(task::MSKtask,
                 djcidx::T0)
    where { T0<:Integer }
    -> b :: Vector{Float64}
```

Obtains the optional constant term vector of a disjunctive constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of the disjunctive constraint. (input)

Return

`b` (`Float64[]`) – The vector b.

Groups

Problem data - disjunctive constraints, *Inspecting the task*

`getdjcdomainidxlist`

```
function getdjcdomainidxlist(task::MSKtask,
                             djcidx::Int64) -> domidxlist :: Vector{Int64}
function getdjcdomainidxlist(task::MSKtask,
                             djcidx::T0)
    where { T0<:Integer }
    -> domidxlist :: Vector{Int64}
```

Obtains the list of domain indexes in a disjunctive constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of the disjunctive constraint. (input)

Return

`domidxlist` (`Int64[]`) – List of term sizes.

Groups

Problem data - disjunctive constraints, *Inspecting the task*

`getdjcname`

```
function getdjcname(task::MSKtask,
                    djcidx::Int64) -> name :: String
function getdjcname(task::MSKtask,
                    djcidx::T0)
    where { T0<:Integer }
    -> name :: String
```

Obtains the name of a disjunctive constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of a disjunctive constraint. (input)

Return

`name` (`String`) – Returns the required name.

Groups

Names, *Problem data - disjunctive constraints*, *Inspecting the task*

`getdjcnamelen`

```
function getdjcnamelen(task::MSKtask,
                       djcidx::Int64) -> len :: Int32
function getdjcnamelen(task::MSKtask,
                       djcidx::T0)
    where { T0<:Integer }
    -> len :: Int32
```

Obtains the length of the name of a disjunctive constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of a disjunctive constraint. (input)

Return

`len` (`Int32`) – Returns the length of the indicated name.

Groups

Names, *Problem data - disjunctive constraints*, *Inspecting the task*

`getdjcnumafe`

```
function getdjcnumafe(task::MSKtask,
                      djcidx::Int64) -> numafe :: Int64
function getdjcnumafe(task::MSKtask,
                      djcidx::T0)
    where { T0<:Integer }
    -> numafe :: Int64
```

Obtains the number of affine expressions in the disjunctive constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of the disjunctive constraint. (input)

Return

`numafe` (`Int64`) – Number of affine expressions in the disjunctive constraint.

Groups

Problem data - disjunctive constraints, *Inspecting the task*

`getdjcnumafetot`

```
function getdjcnumafetot(task::MSKtask) -> numafetot :: Int64
```

Obtains the total number of affine expressions in all disjunctive constraints.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`numafetot` (`Int64`) – Number of affine expressions in all disjunctive constraints.

Groups
Problem data - disjunctive constraints, *Inspecting the task*

`getdjcnumdomain`

```
function getdjcnumdomain(task::MSKtask,
                         djcidx::Int64) -> numdomain :: Int64
function getdjcnumdomain(task::MSKtask,
                         djcidx::T0)
    where { T0<:Integer }
    -> numdomain :: Int64
```

Obtains the number of domains in the disjunctive constraint.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of the disjunctive constraint. (input)

Return
`numdomain` (`Int64`) – Number of domains in the disjunctive constraint.

Groups
Problem data - disjunctive constraints, *Inspecting the task*

`getdjcnumdomaintot`

```
function getdjcnumdomaintot(task::MSKtask) -> numdomaintot :: Int64
```

Obtains the total number of domains in all disjunctive constraints.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`numdomaintot` (`Int64`) – Number of domains in all disjunctive constraints.

Groups
Problem data - disjunctive constraints, *Inspecting the task*

`getdjcnumterm`

```
function getdjcnumterm(task::MSKtask,
                       djcidx::Int64) -> numterm :: Int64
function getdjcnumterm(task::MSKtask,
                       djcidx::T0)
    where { T0<:Integer }
    -> numterm :: Int64
```

Obtains the number terms in the disjunctive constraint.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of the disjunctive constraint. (input)

Return
`numterm` (`Int64`) – Number of terms in the disjunctive constraint.

Groups
Problem data - disjunctive constraints, *Inspecting the task*

`getdjcnumtermtot`

```
function getdjcnumtermtot(task::MSKtask) -> numtermtot :: Int64
```

Obtains the total number of terms in all disjunctive constraints.

Parameters
`task` (`MSKtask`) – An optimization task. (input)
Return
`numtermtot` (`Int64`) – Total number of terms in all disjunctive constraints.
Groups
Problem data - disjunctive constraints, *Inspecting the task*

`getdjcs`

```
function getdjcs(task::MSKtask) -> (domidxlist :: Vector{Int64},afeidxlist ::␣
↪Vector{Int64},b :: Vector{Float64},termsizelist :: Vector{Int64},numterms ::␣
↪Vector{Int64})
```

Obtains full data of all disjunctive constraints. The output arrays must have minimal lengths determined by the following methods: `domainidxlist` by *getdjcnumdomaintot*, `afeidxlist` and `b` by *getdjcnumafetot*, `termsizelist` by *getdjcnumtermtot* and `numterms` by *getnumdomain*.

Parameters
`task` (`MSKtask`) – An optimization task. (input)
Return
- `domidxlist` (`Int64[]`) – The concatenation of index lists of domains appearing in all disjunctive constraints.
- `afeidxlist` (`Int64[]`) – The concatenation of index lists of affine expressions appearing in all disjunctive constraints.
- `b` (`Float64[]`) – The concatenation of vectors b appearing in all disjunctive constraints.
- `termsizelist` (`Int64[]`) – The concatenation of lists of term sizes appearing in all disjunctive constraints.
- `numterms` (`Int64[]`) – The number of terms in each of the disjunctive constraints.

Groups
Problem data - disjunctive constraints, *Inspecting the task*

`getdjctermsizelist`

```
function getdjctermsizelist(task::MSKtask,
                            djcidx::Int64) -> termsizelist :: Vector{Int64}
function getdjctermsizelist(task::MSKtask,
                            djcidx::T0)
    where { T0<:Integer }
    -> termsizelist :: Vector{Int64}
```

Obtains the list of term sizes in a disjunctive constraint.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of the disjunctive constraint. (input)

Return
`termsizelist` (`Int64[]`) – List of term sizes.
Groups
Problem data - disjunctive constraints, *Inspecting the task*

`getdomainn`

```
function getdomainn(task::MSKtask,
                    domidx::Int64) -> n :: Int64
function getdomainn(task::MSKtask,
                    domidx::T0)
    where { T0<:Integer }
    -> n :: Int64
```

Obtains the dimension of the domain.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidx` (`Int64`) – Index of the domain. (input)

Return

`n` (`Int64`) – Dimension of the domain.

Groups

Problem data - domain, *Inspecting the task*

`getdomainname`

```
function getdomainname(task::MSKtask,
                       domidx::Int64) -> name :: String
function getdomainname(task::MSKtask,
                       domidx::T0)
    where { T0<:Integer }
    -> name :: String
```

Obtains the name of a domain.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidx` (`Int64`) – Index of a domain. (input)

Return

`name` (`String`) – Returns the required name.

Groups

Names, *Problem data - domain*, *Inspecting the task*

`getdomainnamelen`

```
function getdomainnamelen(task::MSKtask,
                          domidx::Int64) -> len :: Int32
function getdomainnamelen(task::MSKtask,
                          domidx::T0)
    where { T0<:Integer }
    -> len :: Int32
```

Obtains the length of the name of a domain.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidx` (`Int64`) – Index of a domain. (input)

Return

`len` (`Int32`) – Returns the length of the indicated name.

Groups

Names, *Problem data - domain*, *Inspecting the task*

`getdomaintype`

```
function getdomaintype(task::MSKtask,
                       domidx::Int64) -> domtype :: Domaintype
function getdomaintype(task::MSKtask,
                       domidx::T0)
    where { T0<:Integer }
    -> domtype :: Domaintype
```

Returns the type of the domain.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidx` (`Int64`) – Index of the domain. (input)

Return

`domtype` (*`Domaintype`*) – The type of the domain.

Groups

Problem data - domain, *Inspecting the task*

`getdouinf`

```
function getdouinf(task::MSKtask,
                   whichdinf::Dinfitem) -> dvalue :: Float64
```

Obtains a double information item from the task information database.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichdinf` (*`Dinfitem`*) – Specifies a double information item. (input)

Return

`dvalue` (`Float64`) – The value of the required double information item.

Groups

Information items and statistics

`getdouparam`

```
function getdouparam(task::MSKtask,
                     param::Dparam) -> parvalue :: Float64
```

Obtains the value of a double parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*`Dparam`*) – Which parameter. (input)

Return

`parvalue` (`Float64`) – Parameter value.

Groups

Parameters

`getdualobj`

```
function getdualobj(task::MSKtask,
                    whichsol::Soltype) -> dualobj :: Float64
```

Computes the dual objective value associated with the solution. Note that if the solution is a primal infeasibility certificate, then the fixed term in the objective value is not included.

Moreover, since there is no dual solution associated with an integer solution, an error will be reported if the dual objective value is requested for the integer solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`dualobj` (`Float64`) – Objective value corresponding to the dual solution.

Groups

Solution information, *Solution - dual*

`getdualproblem`

```
function getdualproblem(task::MSKtask) -> dualtask :: MSKtask
```

Returns the dual problem as a task. The dual computed by this function is intended for demonstration, educational and debugging purposes. It closely follows the dual form introduced in the documentation, but need not exactly reflect dualization taking place internally in the solver.

The function attempts to detect sparse LMIs and remove redundant linear constraints, trying to formulate the most efficient dual LMI (i.e. SDP in primal form), unless *`MSK_IPAR_GETDUAL_CONVERT_LMIS`* is turned off.

Problems with quadratic objective or constraints are not supported. Integer variables and disjunctive constraints are ignored and only a warning is issued.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`dualtask` (`MSKtask`) – A new task containing the dualized problem.

Groups

Environment and task management

`getdualsolutionnorms`

```
function getdualsolutionnorms(task::MSKtask,
                              whichsol::Soltype) -> (nrmy :: Float64,nrmslc ::␣
↪Float64,nrmsuc :: Float64,nrmslx :: Float64,nrmsux :: Float64,nrmsnx ::␣
↪Float64,nrmbars :: Float64)
```

Compute norms of the dual solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

- `nrmy` (`Float64`) – The norm of the y vector.
- `nrmslc` (`Float64`) – The norm of the s_l^c vector.
- `nrmsuc` (`Float64`) – The norm of the s_u^c vector.
- `nrmslx` (`Float64`) – The norm of the s_l^x vector.
- `nrmsux` (`Float64`) – The norm of the s_u^x vector.
- `nrmsnx` (`Float64`) – The norm of the s_n^x vector.
- `nrmbars` (`Float64`) – The norm of the $\overline{S}$ vector.

Groups

Solution information

`getdviolacc`

```
function getdviolacc(task::MSKtask,
                     whichsol::Soltype,
                     accidxlist::Vector{Int64}) -> viol :: Vector{Float64}
function getdviolacc(task::MSKtask,
```

(continues on next page)

(continued from previous page)

```
                    whichsol::Soltype,
                    accidxlist::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

Let $(s_n^x)^*$ be the value of variable (s_n^x) for the specified solution. For simplicity let us assume that s_n^x is a member of a quadratic cone, then the violation is computed as follows

$$\begin{cases} \max(0, (\|s_n^x\|_{2:n}^* - (s_n^x)_1^*)/\sqrt{2}, & (s_n^x)^* \geq -\|(s_n^x)_{2:n}^*\|, \\ \|(s_n^x)^*\|, & \text{otherwise.} \end{cases}$$

Both when the solution is a certificate of primal infeasibility or when it is a dual feasible solution the violation should be small.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `accidxlist` (`Int64[]`) – An array of indexes of conic constraints. (input)

Return

`viol` (`Float64[]`) – `viol[k]` is the violation of the dual solution associated with the conic constraint `sub[k]`.

Groups

Solution information

`getdviolbarvar`

```
function getdviolbarvar(task::MSKtask,
                        whichsol::Soltype,
                        sub::Vector{Int32}) -> viol :: Vector{Float64}
function getdviolbarvar(task::MSKtask,
                        whichsol::Soltype,
                        sub::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

Let $(\overline{S}_j)^*$ be the value of variable $\overline{S}_j$ for the specified solution. Then the dual violation of the solution associated with variable $\overline{S}_j$ is given by

$$\max(-\lambda_{\min}(\overline{S}_j),\ 0.0).$$

Both when the solution is a certificate of primal infeasibility and when it is dual feasible solution the violation should be small.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `sub` (`Int32[]`) – An array of indexes of $\overline{X}$ variables. (input)

Return

`viol` (`Float64[]`) – `viol[k]` is the violation of the solution for the constraint $\overline{S}_{\texttt{sub}[k]} \in \mathcal{S}_+$.

Groups

Solution information

`getdviolcon`

```
function getdviolcon(task::MSKtask,
                     whichsol::Soltype,
                     sub::Vector{Int32}) -> viol :: Vector{Float64}
function getdviolcon(task::MSKtask,
                     whichsol::Soltype,
                     sub::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

The violation of the dual solution associated with the i-th constraint is computed as follows

$$\max(\rho((s_l^c)_i^*, (b_l^c)_i),\ \rho((s_u^c)_i^*, -(b_u^c)_i),\ |-y_i + (s_l^c)_i^* - (s_u^c)_i^*|)$$

where

$$\rho(x, l) = \begin{cases} -x, & l > -\infty, \\ |x|, & \text{otherwise.} \end{cases}$$

Both when the solution is a certificate of primal infeasibility or it is a dual feasible solution the violation should be small.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `sub` (`Int32[]`) – An array of indexes of constraints. (input)

Return

`viol` (`Float64[]`) – `viol[k]` is the violation of dual solution associated with the constraint `sub[k]`.

Groups

Solution information

~~`getdviolcones`~~ *Deprecated*

```
function getdviolcones(task::MSKtask,
                       whichsol::Soltype,
                       sub::Vector{Int32}) -> viol :: Vector{Float64}
function getdviolcones(task::MSKtask,
                       whichsol::Soltype,
                       sub::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Let $(s_n^x)^*$ be the value of variable (s_n^x) for the specified solution. For simplicity let us assume that s_n^x is a member of a quadratic cone, then the violation is computed as follows

$$\begin{cases} \max(0, (\|s_n^x\|_{2:n}^* - (s_n^x)_1^*)/\sqrt{2}, & (s_n^x)^* \geq -\|(s_n^x)_{2:n}^*\|, \\ \|(s_n^x)^*\|, & \text{otherwise.} \end{cases}$$

Both when the solution is a certificate of primal infeasibility or when it is a dual feasible solution the violation should be small.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `sub` (`Int32[]`) – An array of indexes of conic constraints. (input)

Return

`viol` (`Float64[]`) – `viol[k]` is the violation of the dual solution associated with the conic constraint `sub[k]`.

Groups

Solution information

`getdviolvar`

```
function getdviolvar(task::MSKtask,
                     whichsol::Soltype,
                     sub::Vector{Int32}) -> viol :: Vector{Float64}
function getdviolvar(task::MSKtask,
                     whichsol::Soltype,
                     sub::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

The violation of the dual solution associated with the j-th variable is computed as follows

$$\max\left(\rho((s_l^x)_j^*,(b_l^x)_j),\ \rho((s_u^x)_j^*,-(b_u^x)_j),\ |\sum_{i=1}^{\texttt{numcon}} a_{ij}y_i + (s_l^x)_j^* - (s_u^x)_j^* - \tau c_j|\right)$$

where

$$\rho(x,l) = \begin{cases} -x, & l > -\infty, \\ |x|, & \text{otherwise} \end{cases}$$

and $\tau = 0$ if the solution is a certificate of primal infeasibility and $\tau = 1$ otherwise. The formula for computing the violation is only shown for the linear case but is generalized appropriately for the more general problems. Both when the solution is a certificate of primal infeasibility or when it is a dual feasible solution the violation should be small.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `sub` (`Int32[]`) – An array of indexes of x variables. (input)

Return

`viol` (`Float64[]`) – `viol[k]` is the violation of dual solution associated with the variable `sub[k]`.

Groups

Solution information

`getinfeasiblesubproblem`

```
function getinfeasiblesubproblem(task::MSKtask,
                                 whichsol::Soltype) -> inftask :: MSKtask
```

Given the solution is a certificate of primal or dual infeasibility then a primal or dual infeasible subproblem is obtained respectively. The subproblem tends to be much smaller than the original problem and hence it is easier to locate the infeasibility inspecting the subproblem than the original problem.

For the procedure to be useful it is important to assign meaningful names to constraints, variables etc. in the original task because those names will be duplicated in the subproblem.

The function is only applicable to linear and conic quadratic optimization problems.

For more information see Sec. 8.3 and Sec. 14.2.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)

- `whichsol` (*Soltype*) – Which solution to use when determining the infeasible subproblem. (input)

Return

`inftask` (`MSKtask`) – A new task containing the infeasible subproblem.

Groups

Infeasibility diagnostic

`getinfname`

```
function getinfname(task::MSKtask,
                    inftype::Inftype,
                    whichinf::Int32) -> infname :: String
function getinfname(task::MSKtask,
                    inftype::Inftype,
                    whichinf::T0)
    where { T0<:Integer }
    -> infname :: String
```

Obtains the name of an information item.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `inftype` (*Inftype*) – Type of the information item. (input)
- `whichinf` (`Int32`) – An information item. (input)

Return

`infname` (`String`) – Name of the information item.

Groups

Information items and statistics, *Names*

`getintinf`

```
function getintinf(task::MSKtask,
                   whichiinf::Iinfitem) -> ivalue :: Int32
```

Obtains an integer information item from the task information database.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichiinf` (*Iinfitem*) – Specifies an integer information item. (input)

Return

`ivalue` (`Int32`) – The value of the required integer information item.

Groups

Information items and statistics

`getintparam`

```
function getintparam(task::MSKtask,
                     param::Iparam) -> parvalue :: Int32
```

Obtains the value of an integer parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*Iparam*) – Which parameter. (input)

Return

`parvalue` (`Int32`) – Parameter value.

Groups

Parameters

`getlasterror`

```
function getlasterror(task::MSKtask) -> (lastrescode :: Rescode,lastmsglen ::␣
↪Int64,lastmsg :: String)
```

Obtains the last response code and corresponding message reported in **MOSEK**.

If there is no previous error, warning or termination code for this task, `lastrescode` returns *MSK_RES_OK* and `lastmsg` returns an empty string, otherwise the last response code different from *MSK_RES_OK* and the corresponding message are returned.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `lastrescode` (*Rescode*) – Returns the last error code reported in the task.
- `lastmsglen` (`Int64`) – Returns the length of the last error message reported in the task.
- `lastmsg` (`String`) – Returns the last error message reported in the task.

Groups

Responses, errors and warnings

`getlenbarvarj`

```
function getlenbarvarj(task::MSKtask,
                       j::Int32) -> lenbarvarj :: Int64
function getlenbarvarj(task::MSKtask,
                       j::T0)
    where { T0<:Integer }
    -> lenbarvarj :: Int64
```

Obtains the length of the j-th semidefinite variable i.e. the number of elements in the lower triangular part.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the semidefinite variable whose length if requested. (input)

Return

`lenbarvarj` (`Int64`) – Number of scalar elements in the lower triangular part of the semidefinite variable.

Groups

Inspecting the task, *Problem data - semidefinite*

`getlintinf`

```
function getlintinf(task::MSKtask,
                    whichliinf::Liinfitem) -> ivalue :: Int64
```

Obtains a long integer information item from the task information database.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichliinf` (*Liinfitem*) – Specifies a long information item. (input)

Return

`ivalue` (`Int64`) – The value of the required long integer information item.

Groups

Information items and statistics

`getlintparam`

```
function getlintparam(task::MSKtask,
                      param::Iparam) -> parvalue :: Int64
```

Obtains the value of an integer parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*Iparam*) – Which parameter. (input)

Return

`parvalue` (`Int64`) – Parameter value.

Groups

Parameters

`getmaxnumanz`

```
function getmaxnumanz(task::MSKtask) -> maxnumanz :: Int64
```

Obtains number of preallocated non-zeros in A. When this number of non-zeros is reached **MOSEK** will automatically allocate more space for A.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`maxnumanz` (`Int64`) – Number of preallocated non-zero linear matrix elements.

Groups

Inspecting the task, *Problem data - linear part*

`getmaxnumbarvar`

```
function getmaxnumbarvar(task::MSKtask) -> maxnumbarvar :: Int32
```

Obtains maximum number of symmetric matrix variables for which space is currently preallocated.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`maxnumbarvar` (`Int32`) – Maximum number of symmetric matrix variables for which space is currently preallocated.

Groups

Inspecting the task, *Problem data - semidefinite*

`getmaxnumcon`

```
function getmaxnumcon(task::MSKtask) -> maxnumcon :: Int32
```

Obtains the number of preallocated constraints in the optimization task. When this number of constraints is reached **MOSEK** will automatically allocate more space for constraints.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`maxnumcon` (`Int32`) – Number of preallocated constraints in the optimization task.

Groups

Inspecting the task, *Problem data - linear part*, *Problem data - constraints*

~~`getmaxnumcone`~~ *Deprecated*

```
function getmaxnumcone(task::MSKtask) -> maxnumcone :: Int32
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Obtains the number of preallocated cones in the optimization task. When this number of cones is reached **MOSEK** will automatically allocate space for more cones.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`maxnumcone` (`Int32`) – Number of preallocated conic constraints in the optimization task.

Groups

Inspecting the task, *Problem data - cones (deprecated)*

`getmaxnumqnz`

```
function getmaxnumqnz(task::MSKtask) -> maxnumqnz :: Int64
```

Obtains the number of preallocated non-zeros for Q (both objective and constraints). When this number of non-zeros is reached **MOSEK** will automatically allocate more space for Q.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`maxnumqnz` (`Int64`) – Number of non-zero elements preallocated in quadratic coefficient matrices.

Groups

Inspecting the task, *Problem data - quadratic part*

`getmaxnumvar`

```
function getmaxnumvar(task::MSKtask) -> maxnumvar :: Int32
```

Obtains the number of preallocated variables in the optimization task. When this number of variables is reached **MOSEK** will automatically allocate more space for variables.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`maxnumvar` (`Int32`) – Number of preallocated variables in the optimization task.

Groups

Inspecting the task, *Problem data - linear part*, *Problem data - variables*

`getmemusage`

```
function getmemusage(task::MSKtask) -> (meminuse :: Int64,maxmemuse :: Int64)
```

Obtains information about the amount of memory used by a task.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `meminuse` (`Int64`) – Amount of memory currently used by the `task`.
- `maxmemuse` (`Int64`) – Maximum amount of memory used by the `task` until now.

Groups

System, memory and debugging

`getnadouinf`

```
function getnadouinf(task::MSKtask,
                     infitemname::AbstractString) -> dvalue :: Float64
```

Obtains a named double information item from task information database.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `infitemname` (`AbstractString`) – The name of a double information item. (input)

Return

`dvalue` (`Float64`) – The value of the required double information item.

Groups

Information items and statistics

`getnadouparam`

```
function getnadouparam(task::MSKtask,
                       paramname::AbstractString) -> parvalue :: Float64
```

Obtains the value of a named double parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `paramname` (`AbstractString`) – Name of a parameter. (input)

Return

`parvalue` (`Float64`) – Parameter value.

Groups

Parameters

`getnaintinf`

```
function getnaintinf(task::MSKtask,
                     infitemname::AbstractString) -> ivalue :: Int32
```

Obtains a named integer information item from the task information database.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `infitemname` (`AbstractString`) – The name of an integer information item. (input)

Return

`ivalue` (`Int32`) – The value of the required integer information item.

Groups

Information items and statistics

`getnaintparam`

```
function getnaintparam(task::MSKtask,
                       paramname::AbstractString) -> parvalue :: Int32
```

Obtains the value of a named integer parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `paramname` (`AbstractString`) – Name of a parameter. (input)

Return

`parvalue` (`Int32`) – Parameter value.

Groups

Parameters

`getnastrparam`

```
function getnastrparam(task::MSKtask,
                       paramname::AbstractString,
                       sizeparamname::Int32) -> (len :: Int32,parvalue :: String)
function getnastrparam(task::MSKtask,
                       paramname::Union{Nothing,AbstractString},
                       sizeparamname::T0)
    where { T0<:Integer }
    -> (len :: Int32,parvalue :: String)
```

Obtains the value of a named string parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `paramname` (`AbstractString`) – Name of a parameter. (input)
- `sizeparamname` (`Int32`) – Size of the name buffer `parvalue`. (input)

Return

- `len` (`Int32`) – Length of the string in `parvalue`.
- `parvalue` (`String`) – Parameter value.

Groups

Parameters, *Names*

`getnumacc`

```
function getnumacc(task::MSKtask) -> num :: Int64
```

Obtains the number of affine conic constraints.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`num` (`Int64`) – The number of affine conic constraints.

Groups

Problem data - affine conic constraints, *Inspecting the task*

`getnumafe`

```
function getnumafe(task::MSKtask) -> numafe :: Int64
```

Obtains the number of affine expressions.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`numafe` (`Int64`) – Number of affine expressions.

Groups

Problem data - affine expressions, *Inspecting the task*

`getnumanz`

```
function getnumanz(task::MSKtask) -> numanz :: Int64
```

Obtains the number of non-zeros in A.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`numanz` (`Int64`) – Number of non-zero elements in the linear constraint matrix.

Groups
Inspecting the task, *Problem data - linear part*

`getnumbarablocktriplets`

```
function getnumbarablocktriplets(task::MSKtask) -> num :: Int64
```

Obtains an upper bound on the number of elements in the block triplet form of $\overline{A}$.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`num` (`Int64`) – An upper bound on the number of elements in the block triplet form of $\overline{A}$.

Groups
Problem data - semidefinite, *Inspecting the task*

`getnumbaranz`

```
function getnumbaranz(task::MSKtask) -> nz :: Int64
```

Get the number of nonzero elements in $\overline{A}$.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`nz` (`Int64`) – The number of nonzero block elements in $\overline{A}$ i.e. the number of $\overline{A}_{ij}$ elements that are nonzero.

Groups
Problem data - semidefinite, *Inspecting the task*

`getnumbarcblocktriplets`

```
function getnumbarcblocktriplets(task::MSKtask) -> num :: Int64
```

Obtains an upper bound on the number of elements in the block triplet form of $\overline{C}$.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`num` (`Int64`) – An upper bound on the number of elements in the block triplet form of $\overline{C}$.

Groups
Problem data - semidefinite, *Inspecting the task*

`getnumbarcnz`

```
function getnumbarcnz(task::MSKtask) -> nz :: Int64
```

Obtains the number of nonzero elements in $\overline{C}$.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`nz` (`Int64`) – The number of nonzeros in $\overline{C}$ i.e. the number of elements $\overline{C}_j$ that are nonzero.

Groups
Problem data - semidefinite, *Inspecting the task*

`getnumbarvar`

```
function getnumbarvar(task::MSKtask) -> numbarvar :: Int32
```

Obtains the number of semidefinite variables.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`numbarvar` (`Int32`) – Number of semidefinite variables in the problem.

Groups
Inspecting the task, *Problem data - semidefinite*

`getnumcon`

```
function getnumcon(task::MSKtask) -> numcon :: Int32
```

Obtains the number of constraints.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`numcon` (`Int32`) – Number of constraints.

Groups
Problem data - linear part, *Problem data - constraints*, *Inspecting the task*

~~`getnumcone`~~ *Deprecated*

```
function getnumcone(task::MSKtask) -> numcone :: Int32
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`numcone` (`Int32`) – Number of conic constraints.

Groups
Problem data - cones (deprecated), *Inspecting the task*

~~`getnumconemem`~~ *Deprecated*

```
function getnumconemem(task::MSKtask,
                       k::Int32) -> nummem :: Int32
function getnumconemem(task::MSKtask,
                       k::T0)
    where { T0<:Integer }
    -> nummem :: Int32
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `k` (`Int32`) – Index of the cone. (input)

Return
`nummem` (`Int32`) – Number of member variables in the cone.

Groups
Problem data - cones (deprecated), *Inspecting the task*

`getnumdjc`

```
function getnumdjc(task::MSKtask) -> num :: Int64
```

Obtains the number of disjunctive constraints.

Parameters
`task` (`MSKtask`) – An optimization task. (input)
Return
`num` (`Int64`) – The number of disjunctive constraints.
Groups
Problem data - disjunctive constraints, *Inspecting the task*

`getnumdomain`

```
function getnumdomain(task::MSKtask) -> numdomain :: Int64
```

Obtain the number of domains defined.

Parameters
`task` (`MSKtask`) – An optimization task. (input)
Return
`numdomain` (`Int64`) – Number of domains in the task.
Groups
Inspecting the task, *Problem data - domain*

`getnumintvar`

```
function getnumintvar(task::MSKtask) -> numintvar :: Int32
```

Obtains the number of integer-constrained variables.

Parameters
`task` (`MSKtask`) – An optimization task. (input)
Return
`numintvar` (`Int32`) – Number of integer variables.
Groups
Inspecting the task, *Problem data - variables*

`getnumparam`

```
function getnumparam(task::MSKtask,
                     partype::Parametertype) -> numparam :: Int32
```

Obtains the number of parameters of a given type.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `partype` (*`Parametertype`*) – Parameter type. (input)

Return
`numparam` (`Int32`) – The number of parameters of type `partype`.
Groups
Inspecting the task, *Parameters*

`getnumqconknz`

```
function getnumqconknz(task::MSKtask,
                       k::Int32) -> numqcnz :: Int64
function getnumqconknz(task::MSKtask,
                       k::T0)
    where { T0<:Integer }
    -> numqcnz :: Int64
```

Obtains the number of non-zero quadratic terms in a constraint.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `k` (`Int32`) – Index of the constraint for which the number quadratic terms should be obtained. (input)

Return
`numqcnz` (`Int64`) – Number of quadratic terms.

Groups
Inspecting the task, *Problem data - constraints*, *Problem data - quadratic part*

`getnumqobjnz`

```
function getnumqobjnz(task::MSKtask) -> numqonz :: Int64
```

Obtains the number of non-zero quadratic terms in the objective.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`numqonz` (`Int64`) – Number of non-zero elements in the quadratic objective terms.

Groups
Inspecting the task, *Problem data - quadratic part*

`getnumsymmat`

```
function getnumsymmat(task::MSKtask) -> num :: Int64
```

Obtains the number of symmetric matrices stored in the vector E.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`num` (`Int64`) – The number of symmetric sparse matrices.

Groups
Problem data - semidefinite, *Inspecting the task*

`getnumvar`

```
function getnumvar(task::MSKtask) -> numvar :: Int32
```

Obtains the number of variables.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`numvar` (`Int32`) – Number of variables.

Groups
Inspecting the task, *Problem data - variables*

`getobjname`

```
function getobjname(task::MSKtask) -> objname :: String
```

Obtains the name assigned to the objective function.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Return
`objname` (`String`) – Assigned the objective name.

Groups
Inspecting the task, *Names*

`getobjnamelen`

```
function getobjnamelen(task::MSKtask) -> len :: Int32
```

Obtains the length of the name assigned to the objective function.

Parameters
`task` (`MSKtask`) – An optimization task. (input)
Return
`len` (`Int32`) – Assigned the length of the objective name.
Groups
Inspecting the task, *Names*

`getobjsense`

```
function getobjsense(task::MSKtask) -> sense :: Objsense
```

Gets the objective sense of the task.

Parameters
`task` (`MSKtask`) – An optimization task. (input)
Return
`sense` (*`Objsense`*) – The returned objective sense.
Groups
Problem data - linear part

`getparamname`

```
function getparamname(task::MSKtask,
                      partype::Parametertype,
                      param::Int32) -> parname :: String
function getparamname(task::MSKtask,
                      partype::Parametertype,
                      param::T0)
    where { T0<:Integer }
    -> parname :: String
```

Obtains the name for a parameter `param` of type `partype`.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `partype` (*`Parametertype`*) – Parameter type. (input)
- `param` (`Int32`) – Which parameter. (input)

Return
`parname` (`String`) – Parameter name.
Groups
Names, *Parameters*

`getpowerdomainalpha`

```
function getpowerdomainalpha(task::MSKtask,
                             domidx::Int64) -> alpha :: Vector{Float64}
function getpowerdomainalpha(task::MSKtask,
                             domidx::T0)
    where { T0<:Integer }
    -> alpha :: Vector{Float64}
```

Obtains the exponent vector α of a primal or dual power cone domain.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidx` (`Int64`) – Index of the domain. (input)

Return

`alpha` (`Float64[]`) – The vector α.

Groups

Problem data - domain, *Inspecting the task*

`getpowerdomaininfo`

```
function getpowerdomaininfo(task::MSKtask,
                            domidx::Int64) -> (n :: Int64,nleft :: Int64)
function getpowerdomaininfo(task::MSKtask,
                            domidx::T0)
    where { T0<:Integer }
    -> (n :: Int64,nleft :: Int64)
```

Obtains structural information about a primal or dual power cone domain.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidx` (`Int64`) – Index of the domain. (input)

Return

- `n` (`Int64`) – Dimension of the domain.
- `nleft` (`Int64`) – Number of variables on the left hand side.

Groups

Problem data - domain, *Inspecting the task*

`getprimalobj`

```
function getprimalobj(task::MSKtask,
                      whichsol::Soltype) -> primalobj :: Float64
```

Computes the primal objective value for the desired solution. Note that if the solution is an infeasibility certificate, then the fixed term in the objective is not included.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`primalobj` (`Float64`) – Objective value corresponding to the primal solution.

Groups

Solution information, *Solution - primal*

`getprimalsolutionnorms`

```
function getprimalsolutionnorms(task::MSKtask,
                                whichsol::Soltype) -> (nrmxc :: Float64,nrmxx ::␣
↪Float64,nrmbarx :: Float64)
```

Compute norms of the primal solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

- `nrmxc` (`Float64`) – The norm of the x^c vector.
- `nrmxx` (`Float64`) – The norm of the x vector.
- `nrmbarx` (`Float64`) – The norm of the $\overline{X}$ vector.

Groups
Solution information

`getprobtype`

```
function getprobtype(task::MSKtask) -> probtype :: Problemtype
```

Obtains the problem type.

Parameters
`task` (`MSKtask`) – An optimization task. (input)
Return
`probtype` (*`Problemtype`*) – The problem type.
Groups
Inspecting the task

`getprosta`

```
function getprosta(task::MSKtask,
                   whichsol::Soltype) -> problemsta :: Prosta
```

Obtains the problem status.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return
`problemsta` (*`Prosta`*) – Problem status.
Groups
Solution information

`getpviolacc`

```
function getpviolacc(task::MSKtask,
                     whichsol::Soltype,
                     accidxlist::Vector{Int64}) -> viol :: Vector{Float64}
function getpviolacc(task::MSKtask,
                     whichsol::Soltype,
                     accidxlist::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

Computes the primal solution violation for a set of affine conic constraints. Let x^* be the value of the variable x for the specified solution. For simplicity let us assume that x is a member of a quadratic cone, then the violation is computed as follows

$$\begin{cases} \max(0, \|x_{2:n}\| - x_1)/\sqrt{2}, & x_1 \geq -\|x_{2:n}\|, \\ \|x\|, & \text{otherwise.} \end{cases}$$

Both when the solution is a certificate of dual infeasibility or when it is primal feasible the violation should be small.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `accidxlist` (`Int64[]`) – An array of indexes of conic constraints. (input)

Return
`viol` (`Float64[]`) – `viol[k]` is the violation of the solution associated with the affine conic constraint number `accidxlist[k]`.
Groups
Solution information

getpviolbarvar

```
function getpviolbarvar(task::MSKtask,
                        whichsol::Soltype,
                        sub::Vector{Int32}) -> viol :: Vector{Float64}
function getpviolbarvar(task::MSKtask,
                        whichsol::Soltype,
                        sub::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

Computes the primal solution violation for a set of semidefinite variables. Let $(\overline{X}_j)^*$ be the value of the variable $\overline{X}_j$ for the specified solution. Then the primal violation of the solution associated with variable $\overline{X}_j$ is given by

$$\max(-\lambda_{\min}(\overline{X}_j),\ 0.0).$$

Both when the solution is a certificate of dual infeasibility or when it is primal feasible the violation should be small.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)
- `sub` (`Int32[]`) – An array of indexes of $\overline{X}$ variables. (input)

Return

`viol` (`Float64[]`) – `viol[k]` is how much the solution violates the constraint $\overline{X}_{\mathtt{sub}[k]} \in \mathcal{S}_+$.

Groups

Solution information

getpviolcon

```
function getpviolcon(task::MSKtask,
                     whichsol::Soltype,
                     sub::Vector{Int32}) -> viol :: Vector{Float64}
function getpviolcon(task::MSKtask,
                     whichsol::Soltype,
                     sub::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

Computes the primal solution violation for a set of constraints. The primal violation of the solution associated with the i-th constraint is given by

$$\max(\tau l_i^c - (x_i^c)^*,\ (x_i^c)^* - \tau u_i^c),\ |\sum_{j=1}^{\mathtt{numvar}} a_{ij} x_j^* - x_i^c|)$$

where $\tau = 0$ if the solution is a certificate of dual infeasibility and $\tau = 1$ otherwise. Both when the solution is a certificate of dual infeasibility and when it is primal feasible the violation should be small. The above formula applies for the linear case but is appropriately generalized in other cases.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)
- `sub` (`Int32[]`) – An array of indexes of constraints. (input)

Return

`viol` (`Float64[]`) – `viol[k]` is the violation associated with the solution for the constraint `sub[k]`.

Groups

Solution information

~~getpviolcones~~ *Deprecated*

```
function getpviolcones(task::MSKtask,
                       whichsol::Soltype,
                       sub::Vector{Int32}) -> viol :: Vector{Float64}
function getpviolcones(task::MSKtask,
                       whichsol::Soltype,
                       sub::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Computes the primal solution violation for a set of conic constraints. Let x^* be the value of the variable x for the specified solution. For simplicity let us assume that x is a member of a quadratic cone, then the violation is computed as follows

$$\begin{cases} \max(0, \|x_{2:n}\| - x_1)/\sqrt{2}, & x_1 \geq -\|x_{2:n}\|, \\ \|x\|, & \text{otherwise.} \end{cases}$$

Both when the solution is a certificate of dual infeasibility or when it is primal feasible the violation should be small.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `sub` (`Int32[]`) – An array of indexes of conic constraints. (input)

Return

`viol` (`Float64[]`) – `viol[k]` is the violation of the solution associated with the conic constraint number `sub[k]`.

Groups

Solution information

`getpvioldjc`

```
function getpvioldjc(task::MSKtask,
                     whichsol::Soltype,
                     djcidxlist::Vector{Int64}) -> viol :: Vector{Float64}
function getpvioldjc(task::MSKtask,
                     whichsol::Soltype,
                     djcidxlist::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

Computes the primal solution violation for a set of disjunctive constraints. For a single DJC the violation is defined as

$$\text{viol}\left(\bigvee_{i=1}^{t}\bigwedge_{j=1}^{s_i} T_{i,j}\right) = \min_{i=1,\ldots,t}\left(\max_{j=1,\ldots,s_j}(\text{viol}(T_{i,j}))\right)$$

where the violation of each simple term $T_{i,j}$ is defined as for an ordinary linear constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

- `djcidxlist` (`Int64[]`) – An array of indexes of disjunctive constraints. (input)

Return

`viol` (`Float64[]`) – `viol[k]` is the violation of the solution associated with the disjunctive constraint number `djcidxlist[k]`.

Groups

Solution information

`getpviolvar`

```
function getpviolvar(task::MSKtask,
                     whichsol::Soltype,
                     sub::Vector{Int32}) -> viol :: Vector{Float64}
function getpviolvar(task::MSKtask,
                     whichsol::Soltype,
                     sub::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> viol :: Vector{Float64}
```

Computes the primal solution violation associated to a set of variables. Let x_j^* be the value of x_j for the specified solution. Then the primal violation of the solution associated with variable x_j is given by

$$\max(\tau l_j^x - x_j^*,\ x_j^* - \tau u_j^x,\ 0).$$

where $\tau = 0$ if the solution is a certificate of dual infeasibility and $\tau = 1$ otherwise. Both when the solution is a certificate of dual infeasibility and when it is primal feasible the violation should be small.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `sub` (`Int32[]`) – An array of indexes of x variables. (input)

Return

`viol` (`Float64[]`) – `viol[k]` is the violation associated with the solution for the variable $x_{\texttt{sub[k]}}$.

Groups

Solution information

`getqconk`

```
function getqconk(task::MSKtask,
                  k::Int32) -> (numqcnz :: Int64,qcsubi :: Vector{Int32},qcsubj :
↪: Vector{Int32},qcval :: Vector{Float64})
function getqconk(task::MSKtask,
                  k::T0)
    where { T0<:Integer }
    -> (numqcnz :: Int64,qcsubi :: Vector{Int32},qcsubj :: Vector{Int32},qcval ::
↪ Vector{Float64})
```

Obtains all the quadratic terms in a constraint. The quadratic terms are stored sequentially in `qcsubi`, `qcsubj`, and `qcval`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `k` (`Int32`) – Which constraint. (input)

Return

- `numqcnz` (`Int64`) – Number of quadratic terms.
- `qcsubi` (`Int32[]`) – Row subscripts for quadratic constraint matrix.

- `qcsubj` (`Int32[]`) – Column subscripts for quadratic constraint matrix.
- `qcval` (`Float64[]`) – Quadratic constraint coefficient values.

Groups

Inspecting the task, *Problem data - quadratic part*, *Problem data - constraints*

`getqobj`

```
function getqobj(task::MSKtask) -> (numqonz :: Int64,qosubi :: Vector{Int32},
↪qosubj :: Vector{Int32},qoval :: Vector{Float64})
```

Obtains the quadratic terms in the objective. The required quadratic terms are stored sequentially in `qosubi`, `qosubj`, and `qoval`.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

- `numqonz` (`Int64`) – Number of non-zero elements in the quadratic objective terms.
- `qosubi` (`Int32[]`) – Row subscripts for quadratic objective coefficients.
- `qosubj` (`Int32[]`) – Column subscripts for quadratic objective coefficients.
- `qoval` (`Float64[]`) – Quadratic objective coefficient values.

Groups

Inspecting the task, *Problem data - quadratic part*

`getqobjij`

```
function getqobjij(task::MSKtask,
                   i::Int32,
                   j::Int32) -> qoij :: Float64
function getqobjij(task::MSKtask,
                   i::T0,
                   j::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> qoij :: Float64
```

Obtains one coefficient q^o_{ij} in the quadratic term of the objective.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Row index of the coefficient. (input)
- `j` (`Int32`) – Column index of coefficient. (input)

Return

`qoij` (`Float64`) – The required coefficient.

Groups

Inspecting the task, *Problem data - quadratic part*

`getreducedcosts`

```
function getreducedcosts(task::MSKtask,
                         whichsol::Soltype,
                         first::Int32,
                         last::Int32) -> redcosts :: Vector{Float64}
function getreducedcosts(task::MSKtask,
                         whichsol::Soltype,
                         first::T0,
                         last::T1)
```

(continues on next page)

(continued from previous page)

```
    where { T0<:Integer,
            T1<:Integer }
    -> redcosts :: Vector{Float64}
```

Computes the reduced costs for a slice of variables and returns them in the array `redcosts` i.e.

$$\texttt{redcosts} = [(s_l^x)_j - (s_u^x)_j,\ j = \texttt{first}, \ldots, \texttt{last} - 1] \tag{15.2}$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – The index of the first variable in the sequence. (input)
- `last` (`Int32`) – The index of the last variable in the sequence plus 1. (input)

Return

`redcosts` (`Float64[]`) – The reduced costs for the required slice of variables.

Groups

Solution - dual

`getskc`

```
function getskc(task::MSKtask,
                whichsol::Soltype) -> skc :: Vector{Stakey}
```

Obtains the status keys for the constraints.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`skc` (*`Stakey`* `[]`) – Status keys for the constraints.

Groups

Solution information

`getskcslice`

```
function getskcslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32) -> skc :: Vector{Stakey}
function getskcslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> skc :: Vector{Stakey}
```

Obtains the status keys for a slice of the constraints.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`skc` (*`Stakey`* `[]`) – Status keys for the constraints.

Groups

Solution information

getskn

```
function getskn(task::MSKtask,
                whichsol::Soltype) -> skn :: Vector{Stakey}
```

Obtains the status keys for the conic constraints.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`skn` (*`Stakey`* `[]`) – Status keys for the conic constraints.

Groups

Solution information

getskx

```
function getskx(task::MSKtask,
                whichsol::Soltype) -> skx :: Vector{Stakey}
```

Obtains the status keys for the scalar variables.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`skx` (*`Stakey`* `[]`) – Status keys for the variables.

Groups

Solution information

getskxslice

```
function getskxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32) -> skx :: Vector{Stakey}
function getskxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> skx :: Vector{Stakey}
```

Obtains the status keys for a slice of the scalar variables.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`skx` (*`Stakey`* `[]`) – Status keys for the variables.

Groups

Solution information

getslc

```
function getslc(task::MSKtask,
                whichsol::Soltype) -> slc :: Vector{Float64}
```

Obtains the s_l^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`slc` (`Float64[]`) – Dual variables corresponding to the lower bounds on the constraints.

Groups

Solution - dual

`getslcslice`

```
function getslcslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32) -> slc :: Vector{Float64}
function getslcslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> slc :: Vector{Float64}
```

Obtains a slice of the s_l^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`slc` (`Float64[]`) – Dual variables corresponding to the lower bounds on the constraints.

Groups

Solution - dual

`getslx`

```
function getslx(task::MSKtask,
                whichsol::Soltype) -> slx :: Vector{Float64}
```

Obtains the s_l^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`slx` (`Float64[]`) – Dual variables corresponding to the lower bounds on the variables.

Groups

Solution - dual

`getslxslice`

```
function getslxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32) -> slx :: Vector{Float64}
function getslxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> slx :: Vector{Float64}
```

Obtains a slice of the s_l^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`slx` (`Float64[]`) – Dual variables corresponding to the lower bounds on the variables.

Groups

Solution - dual

`getsnx`

```
function getsnx(task::MSKtask,
                whichsol::Soltype) -> snx :: Vector{Float64}
```

Obtains the s_n^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`snx` (`Float64[]`) – Dual variables corresponding to the conic constraints on the variables.

Groups

Solution - dual

`getsnxslice`

```
function getsnxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32) -> snx :: Vector{Float64}
function getsnxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> snx :: Vector{Float64}
```

Obtains a slice of the s_n^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)

- `whichsol` (*Soltype*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`snx` (`Float64[]`) – Dual variables corresponding to the conic constraints on the variables.

Groups

Solution - dual

`getsolsta`

```
function getsolsta(task::MSKtask,
                   whichsol::Soltype) -> solutionsta :: Solsta
```

Obtains the solution status.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)

Return

`solutionsta` (*Solsta*) – Solution status.

Groups

Solution information

`getsolution`

```
function getsolution(task::MSKtask,
                     whichsol::Soltype) -> (problemsta :: Prosta,solutionsta ::␣
↪Solsta,skc :: Vector{Stakey},skx :: Vector{Stakey},skn :: Vector{Stakey},xc ::␣
↪Vector{Float64},xx :: Vector{Float64},y :: Vector{Float64},slc :: Vector
↪{Float64},suc :: Vector{Float64},slx :: Vector{Float64},sux :: Vector{Float64},
↪snx :: Vector{Float64})
```

Obtains the complete solution.

Consider the case of linear programming. The primal problem is given by

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

and the corresponding dual problem is

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

A conic optimization problem has the same primal variables as in the linear case. Recall that the dual of a conic optimization problem is given by:

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

The mapping between variables and arguments to the function is as follows:

- `xx` : Corresponds to variable x (also denoted x^x).
- `xc` : Corresponds to $x^c := Ax$.
- `y` : Corresponds to variable y.
- `slc`: Corresponds to variable s_l^c.
- `suc`: Corresponds to variable s_u^c.
- `slx`: Corresponds to variable s_l^x.
- `sux`: Corresponds to variable s_u^x.
- `snx`: Corresponds to variable s_n^x.

The meaning of the values returned by this function depend on the *solution status* returned in the argument `solsta`. The most important possible values of `solsta` are:

- *MSK_SOL_STA_OPTIMAL* : An optimal solution satisfying the optimality criteria for continuous problems is returned.
- *MSK_SOL_STA_INTEGER_OPTIMAL* : An optimal solution satisfying the optimality criteria for integer problems is returned.
- *MSK_SOL_STA_PRIM_FEAS* : A solution satisfying the feasibility criteria.
- *MSK_SOL_STA_PRIM_INFEAS_CER* : A primal certificate of infeasibility is returned.
- *MSK_SOL_STA_DUAL_INFEAS_CER* : A dual certificate of infeasibility is returned.

In order to retrieve the primal and dual values of semidefinite variables see *getbarxj* and *getbarsj*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)

Return

- `problemsta` (*Prosta*) – Problem status.
- `solutionsta` (*Solsta*) – Solution status.
- `skc` (*Stakey* `[]`) – Status keys for the constraints.
- `skx` (*Stakey* `[]`) – Status keys for the variables.
- `skn` (*Stakey* `[]`) – Status keys for the conic constraints.
- `xc` (`Float64[]`) – Primal constraint solution.
- `xx` (`Float64[]`) – Primal variable solution.
- `y` (`Float64[]`) – Vector of dual variables corresponding to the constraints.
- `slc` (`Float64[]`) – Dual variables corresponding to the lower bounds on the constraints.
- `suc` (`Float64[]`) – Dual variables corresponding to the upper bounds on the constraints.
- `slx` (`Float64[]`) – Dual variables corresponding to the lower bounds on the variables.
- `sux` (`Float64[]`) – Dual variables corresponding to the upper bounds on the variables.
- `snx` (`Float64[]`) – Dual variables corresponding to the conic constraints on the variables.

Groups

Solution information, *Solution - primal*, *Solution - dual*

`getsolutioninfo`

```
function getsolutioninfo(task::MSKtask,
                         whichsol::Soltype) -> (pobj :: Float64,pviolcon ::␣
↪Float64,pviolvar :: Float64,pviolbarvar :: Float64,pviolcone :: Float64,
↪pviolitg :: Float64,dobj :: Float64,dviolcon :: Float64,dviolvar :: Float64,
↪dviolbarvar :: Float64,dviolcone :: Float64)
```

Obtains information about a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)

Return

- `pobj` (`Float64`) – The primal objective value as computed by *getprimalobj*.
- `pviolcon` (`Float64`) – Maximal primal violation of the solution associated with the x^c variables where the violations are computed by *getpviolcon*.
- `pviolvar` (`Float64`) – Maximal primal violation of the solution for the x variables where the violations are computed by *getpviolvar*.
- `pviolbarvar` (`Float64`) – Maximal primal violation of solution for the $\overline{X}$ variables where the violations are computed by *getpviolbarvar*.
- `pviolcone` (`Float64`) – Maximal primal violation of solution for the conic constraints where the violations are computed by ~~*getpviolcones*~~.
- `pviolitg` (`Float64`) – Maximal violation in the integer constraints. The violation for an integer variable x_j is given by $\min(x_j - \lfloor x_j \rfloor, \lceil x_j \rceil - x_j)$. This number is always zero for the interior-point and basic solutions.
- `dobj` (`Float64`) – Dual objective value as computed by *getdualobj*.
- `dviolcon` (`Float64`) – Maximal violation of the dual solution associated with the x^c variable as computed by *getdviolcon*.
- `dviolvar` (`Float64`) – Maximal violation of the dual solution associated with the x variable as computed by *getdviolvar*.
- `dviolbarvar` (`Float64`) – Maximal violation of the dual solution associated with the $\overline{S}$ variable as computed by *getdviolbarvar*.
- `dviolcone` (`Float64`) – Maximal violation of the dual solution associated with the dual conic constraints as computed by ~~*getdviolcones*~~.

Groups

Solution information

`getsolutioninfonew`

```
function getsolutioninfonew(task::MSKtask,
                            whichsol::Soltype) -> (pobj :: Float64,pviolcon ::␣
↪Float64,pviolvar :: Float64,pviolbarvar :: Float64,pviolcone :: Float64,
↪pviolacc :: Float64,pvioldjc :: Float64,pviolitg :: Float64,dobj :: Float64,
↪dviolcon :: Float64,dviolvar :: Float64,dviolbarvar :: Float64,dviolcone ::␣
↪Float64,dviolacc :: Float64)
```

Obtains information about a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)

Return

- `pobj` (`Float64`) – The primal objective value as computed by *getprimalobj*.
- `pviolcon` (`Float64`) – Maximal primal violation of the solution associated with the x^c variables where the violations are computed by *getpviolcon*.
- `pviolvar` (`Float64`) – Maximal primal violation of the solution for the x variables where the violations are computed by *getpviolvar*.
- `pviolbarvar` (`Float64`) – Maximal primal violation of solution for the $\overline{X}$ variables where the violations are computed by *getpviolbarvar*.
- `pviolcone` (`Float64`) – Maximal primal violation of solution for the conic constraints where the violations are computed by ~~*getpviolcones*~~.
- `pviolacc` (`Float64`) – Maximal primal violation of solution for the affine conic constraints where the violations are computed by *getpviolacc*.

- `pvioldjc` (`Float64`) – Maximal primal violation of solution for the disjunctive constraints where the violations are computed by *getpvioldjc*.
- `pviolitg` (`Float64`) – Maximal violation in the integer constraints. The violation for an integer variable x_j is given by $\min(x_j - \lfloor x_j \rfloor, \lceil x_j \rceil - x_j)$. This number is always zero for the interior-point and basic solutions.
- `dobj` (`Float64`) – Dual objective value as computed by *getdualobj*.
- `dviolcon` (`Float64`) – Maximal violation of the dual solution associated with the x^c variable as computed by *getdviolcon*.
- `dviolvar` (`Float64`) – Maximal violation of the dual solution associated with the x variable as computed by *getdviolvar*.
- `dviolbarvar` (`Float64`) – Maximal violation of the dual solution associated with the $\overline{S}$ variable as computed by *getdviolbarvar*.
- `dviolcone` (`Float64`) – Maximal violation of the dual solution associated with the dual conic constraints as computed by ~~*getdviolcones*~~.
- `dviolacc` (`Float64`) – Maximal violation of the dual solution associated with the affine conic constraints as computed by *getdviolacc*.

Groups

Solution information

`getsolutionnew`

```
function getsolutionnew(task::MSKtask,
                        whichsol::Soltype) -> (problemsta :: Prosta,solutionsta :
↪: Solsta,skc :: Vector{Stakey},skx :: Vector{Stakey},skn :: Vector{Stakey},xc :
↪: Vector{Float64},xx :: Vector{Float64},y :: Vector{Float64},slc :: Vector
↪{Float64},suc :: Vector{Float64},slx :: Vector{Float64},sux :: Vector{Float64},
↪snx :: Vector{Float64},doty :: Vector{Float64})
```

Obtains the complete solution. See *getsolution* for further information.

In order to retrieve the primal and dual values of semidefinite variables see *getbarxj* and *getbarsj*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)

Return

- `problemsta` (*Prosta*) – Problem status.
- `solutionsta` (*Solsta*) – Solution status.
- `skc` (*Stakey* `[]`) – Status keys for the constraints.
- `skx` (*Stakey* `[]`) – Status keys for the variables.
- `skn` (*Stakey* `[]`) – Status keys for the conic constraints.
- `xc` (`Float64[]`) – Primal constraint solution.
- `xx` (`Float64[]`) – Primal variable solution.
- `y` (`Float64[]`) – Vector of dual variables corresponding to the constraints.
- `slc` (`Float64[]`) – Dual variables corresponding to the lower bounds on the constraints.
- `suc` (`Float64[]`) – Dual variables corresponding to the upper bounds on the constraints.
- `slx` (`Float64[]`) – Dual variables corresponding to the lower bounds on the variables.
- `sux` (`Float64[]`) – Dual variables corresponding to the upper bounds on the variables.
- `snx` (`Float64[]`) – Dual variables corresponding to the conic constraints on the variables.
- `doty` (`Float64[]`) – Dual variables corresponding to affine conic constraints.

Groups

Solution information, *Solution - primal*, *Solution - dual*

getsolutionslice

```
function getsolutionslice(task::MSKtask,
                          whichsol::Soltype,
                          solitem::Solitem,
                          first::Int32,
                          last::Int32) -> values :: Vector{Float64}
function getsolutionslice(task::MSKtask,
                          whichsol::Soltype,
                          solitem::Solitem,
                          first::T0,
                          last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> values :: Vector{Float64}
```

Obtains a slice of one item from the solution. The format of the solution is exactly as in *getsolution*. The parameter `solitem` determines which of the solution vectors should be returned.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)
- `solitem` (*Solitem*) – Which part of the solution is required. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`values` (`Float64[]`) – The values in the required sequence are stored sequentially in `values`.

Groups

Solution - primal, *Solution - dual*, *Solution information*

getsparsesymmat

```
function getsparsesymmat(task::MSKtask,
                         idx::Int64) -> (subi :: Vector{Int32},subj :: Vector
↪{Int32},valij :: Vector{Float64})
function getsparsesymmat(task::MSKtask,
                         idx::T0)
    where { T0<:Integer }
    -> (subi :: Vector{Int32},subj :: Vector{Int32},valij :: Vector{Float64})
```

Get a single symmetric matrix from the matrix store.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `idx` (`Int64`) – Index of the matrix to retrieve. (input)

Return

- `subi` (`Int32[]`) – Row subscripts of the matrix non-zero elements.
- `subj` (`Int32[]`) – Column subscripts of the matrix non-zero elements.
- `valij` (`Float64[]`) – Coefficients of the matrix non-zero elements.

Groups

Problem data - semidefinite, *Inspecting the task*

getstrparam

```
function getstrparam(task::MSKtask,
                     param::Sparam) -> (len :: Int32,parvalue :: String)
```

Obtains the value of a string parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*Sparam*) – Which parameter. (input)

Return

- `len` (`Int32`) – The length of the parameter value.
- `parvalue` (`String`) – Parameter value.

Groups

Names, *Parameters*

`getstrparamlen`

```
function getstrparamlen(task::MSKtask,
                        param::Sparam) -> len :: Int32
```

Obtains the length of a string parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*Sparam*) – Which parameter. (input)

Return

`len` (`Int32`) – The length of the parameter value.

Groups

Names, *Parameters*

`getsuc`

```
function getsuc(task::MSKtask,
                whichsol::Soltype) -> suc :: Vector{Float64}
```

Obtains the s_u^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)

Return

`suc` (`Float64[]`) – Dual variables corresponding to the upper bounds on the constraints.

Groups

Solution - dual

`getsucslice`

```
function getsucslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32) -> suc :: Vector{Float64}
function getsucslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> suc :: Vector{Float64}
```

Obtains a slice of the s_u^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`suc` (`Float64[]`) – Dual variables corresponding to the upper bounds on the constraints.

Groups

Solution - dual

`getsux`

```
function getsux(task::MSKtask,
                whichsol::Soltype) -> sux :: Vector{Float64}
```

Obtains the s_u^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`sux` (`Float64[]`) – Dual variables corresponding to the upper bounds on the variables.

Groups

Solution - dual

`getsuxslice`

```
function getsuxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32) -> sux :: Vector{Float64}
function getsuxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> sux :: Vector{Float64}
```

Obtains a slice of the s_u^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`sux` (`Float64[]`) – Dual variables corresponding to the upper bounds on the variables.

Groups

Solution - dual

`getsymmatinfo`

```
function getsymmatinfo(task::MSKtask,
                       idx::Int64) -> (dim :: Int32,nz :: Int64,mattype ::␣
↪Symmattype)
function getsymmatinfo(task::MSKtask,
                       idx::T0)
    where { T0<:Integer }
    -> (dim :: Int32,nz :: Int64,mattype :: Symmattype)
```

MOSEK maintains a vector denoted by E of symmetric data matrices. This function makes it possible to obtain important information about a single matrix in E.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `idx` (`Int64`) – Index of the matrix for which information is requested. (input)

Return

- `dim` (`Int32`) – Returns the dimension of the requested matrix.
- `nz` (`Int64`) – Returns the number of non-zeros in the requested matrix.
- `mattype` (*`Symmattype`*) – Returns the type of the requested matrix.

Groups

Problem data - semidefinite, *Inspecting the task*

`gettaskname`

```
function gettaskname(task::MSKtask) -> taskname :: String
```

Obtains the name assigned to the task.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`taskname` (`String`) – Returns the task name.

Groups

Names, *Inspecting the task*

`gettasknamelen`

```
function gettasknamelen(task::MSKtask) -> len :: Int32
```

Obtains the length the task name.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`len` (`Int32`) – Returns the length of the task name.

Groups

Names, *Inspecting the task*

`getvarbound`

```
function getvarbound(task::MSKtask,
                     i::Int32) -> (bk :: Boundkey,bl :: Float64,bu :: Float64)
function getvarbound(task::MSKtask,
                     i::T0)
    where { T0<:Integer }
    -> (bk :: Boundkey,bl :: Float64,bu :: Float64)
```

Obtains bound information for one variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the variable for which the bound information should be obtained. (input)

Return

- `bk` (*`Boundkey`*) – Bound keys.
- `bl` (`Float64`) – Values for lower bounds.
- `bu` (`Float64`) – Values for upper bounds.

Groups

Problem data - linear part, *Inspecting the task*, *Problem data - bounds*, *Problem data - variables*

`getvarboundslice`

```
function getvarboundslice(task::MSKtask,
                          first::Int32,
                          last::Int32) -> (bk :: Vector{Boundkey},bl :: Vector
↪{Float64},bu :: Vector{Float64})
function getvarboundslice(task::MSKtask,
                          first::T0,
                          last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> (bk :: Vector{Boundkey},bl :: Vector{Float64},bu :: Vector{Float64})
```

Obtains bounds information for a slice of the variables.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

- `bk` (*`Boundkey`*`[]`) – Bound keys.
- `bl` (`Float64[]`) – Values for lower bounds.
- `bu` (`Float64[]`) – Values for upper bounds.

Groups

Problem data - linear part, *Inspecting the task*, *Problem data - bounds*, *Problem data - variables*

`getvarname`

```
function getvarname(task::MSKtask,
                    j::Int32) -> name :: String
function getvarname(task::MSKtask,
                    j::T0)
    where { T0<:Integer }
    -> name :: String
```

Obtains the name of a variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of a variable. (input)

Return

`name` (`String`) – Returns the required name.

Groups

Names, *Problem data - linear part*, *Problem data - variables*, *Inspecting the task*

getvarnameindex

```
function getvarnameindex(task::MSKtask,
                         somename::AbstractString) -> (asgn :: Int32,index ::␣
↪Int32)
```

Checks whether the name `somename` has been assigned to any variable. If so, the index of the variable is reported.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `somename` (`AbstractString`) – The name which should be checked. (input)

Return

- `asgn` (`Int32`) – Is non-zero if the name `somename` is assigned to a variable.
- `index` (`Int32`) – If the name `somename` is assigned to a variable, then `index` is the index of the variable.

Groups

Names, *Problem data - linear part*, *Problem data - variables*, *Inspecting the task*

getvarnamelen

```
function getvarnamelen(task::MSKtask,
                       i::Int32) -> len :: Int32
function getvarnamelen(task::MSKtask,
                       i::T0)
    where { T0<:Integer }
    -> len :: Int32
```

Obtains the length of the name of a variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of a variable. (input)

Return

`len` (`Int32`) – Returns the length of the indicated name.

Groups

Names, *Problem data - linear part*, *Problem data - variables*, *Inspecting the task*

getvartype

```
function getvartype(task::MSKtask,
                    j::Int32) -> vartype :: Variabletype
function getvartype(task::MSKtask,
                    j::T0)
    where { T0<:Integer }
    -> vartype :: Variabletype
```

Gets the variable type of one variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the variable. (input)

Return

`vartype` (*`Variabletype`*) – Variable type of the j-th variable.

Groups

Inspecting the task, *Problem data - variables*

getvartypelist

```
function getvartypelist(task::MSKtask,
                        subj::Vector{Int32}) -> vartype :: Vector{Variabletype}
function getvartypelist(task::MSKtask,
                        subj::T0)
    where { T0<:AbstractVector{<:Integer} }
    -> vartype :: Vector{Variabletype}
```

Obtains the variable type of one or more variables. Upon return `vartype[k]` is the variable type of variable `subj[k]`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subj` (`Int32[]`) – A list of variable indexes. (input)

Return

`vartype` (*`Variabletype`*`[]`) – The variables types corresponding to the variables specified by `subj`.

Groups

Inspecting the task, *Problem data - variables*

`getversion`

```
function getversion() -> (major :: Int32,minor :: Int32,revision :: Int32)
```

Obtains **MOSEK** version information.

Return

- `major` (`Int32`) – Major version number.
- `minor` (`Int32`) – Minor version number.
- `revision` (`Int32`) – Revision number.

Groups

Versions

`getxc`

```
function getxc(task::MSKtask,
               whichsol::Soltype) -> xc :: Vector{Float64}
```

Obtains the x^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`xc` (`Float64[]`) – Primal constraint solution.

Groups

Solution - primal

`getxcslice`

```
function getxcslice(task::MSKtask,
                    whichsol::Soltype,
                    first::Int32,
                    last::Int32) -> xc :: Vector{Float64}
function getxcslice(task::MSKtask,
                    whichsol::Soltype,
                    first::T0,
                    last::T1)
```

(continues on next page)

(continued from previous page)

```
    where { T0<:Integer,
            T1<:Integer }
    -> xc :: Vector{Float64}
```

Obtains a slice of the x^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`xc` (`Float64[]`) – Primal constraint solution.

Groups

Solution - primal

`getxx`

```
function getxx(task::MSKtask,
               whichsol::Soltype) -> xx :: Vector{Float64}
```

Obtains the x^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`xx` (`Float64[]`) – Primal variable solution.

Groups

Solution - primal

`getxxslice`

```
function getxxslice(task::MSKtask,
                    whichsol::Soltype,
                    first::Int32,
                    last::Int32) -> xx :: Vector{Float64}
function getxxslice(task::MSKtask,
                    whichsol::Soltype,
                    first::T0,
                    last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> xx :: Vector{Float64}
```

Obtains a slice of the x^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`xx` (`Float64[]`) – Primal variable solution.

Groups

Solution - primal

gety

```
function gety(task::MSKtask,
              whichsol::Soltype) -> y :: Vector{Float64}
```

Obtains the y vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`y` (`Float64[]`) – Vector of dual variables corresponding to the constraints.

Groups

Solution - dual

getyslice

```
function getyslice(task::MSKtask,
                   whichsol::Soltype,
                   first::Int32,
                   last::Int32) -> y :: Vector{Float64}
function getyslice(task::MSKtask,
                   whichsol::Soltype,
                   first::T0,
                   last::T1)
    where { T0<:Integer,
            T1<:Integer }
    -> y :: Vector{Float64}
```

Obtains a slice of the y vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

Return

`y` (`Float64[]`) – Vector of dual variables corresponding to the constraints.

Groups

Solution - dual

iinfitemtostr

```
function iinfitemtostr(item::Iinfitem) -> str :: String
```

Obtains an identifier string corresponding to a information item.

Parameters

`item` (*`Iinfitem`*) – Information item. (input)

Return

`str` (`String`) – String corresponding to the bound information item `item`.

Groups

Names

infeasibilityreport

```
function infeasibilityreport(task::MSKtask,
                             whichstream::Streamtype,
                             whichsol::Soltype)
```

Prints the infeasibility report to an output stream.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*Streamtype*) – Index of the stream. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)

Groups

Infeasibility diagnostic

`initbasissolve`

```
function initbasissolve(task::MSKtask) -> basis :: Vector{Int32}
```

Prepare a task for use with the *solvewithbasis* function.

This function should be called

- immediately before the first call to *solvewithbasis*, and
- immediately before any subsequent call to *solvewithbasis* if the task has been modified.

If the basis is singular i.e. not invertible, then the error *MSK_RES_ERR_BASIS_SINGULAR* is reported.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`basis` (`Int32[]`) – The array of basis indexes to use. The array is interpreted as follows: If $\texttt{basis}[i] \leq numcon$, then $x^c_{\texttt{basis}[i]}$ is in the basis at position i, otherwise $x_{\texttt{basis}[i]-\texttt{numcon}}$ is in the basis at position i.

Groups

Solving systems with basis matrix

`inputdata`

```
function inputdata(task::MSKtask,
                   maxnumcon::Int32,
                   maxnumvar::Int32,
                   c::Union{Nothing,Vector{Float64}},
                   cfix::Float64,
                   aptrb::Vector{Int64},
                   aptre::Vector{Int64},
                   asub::Vector{Int32},
                   aval::Vector{Float64},
                   bkc::Vector{Boundkey},
                   blc::Vector{Float64},
                   buc::Vector{Float64},
                   bkx::Vector{Boundkey},
                   blx::Vector{Float64},
                   bux::Vector{Float64})
function inputdata(task::MSKtask,
                   maxnumcon::T0,
                   maxnumvar::T1,
                   c::T2,
                   cfix::T3,
                   aptrb::T4,
                   aptre::T5,
                   asub::T6,
                   aval::T7,
                   bkc::Vector{Boundkey},
                   blc::T8,
```

(continues on next page)

(continued from previous page)

```
                    buc::T9,
                    bkx::Vector{Boundkey},
                    blx::T10,
                    bux::T11)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number},
            T3<:Number,
            T4<:AbstractVector{<:Integer},
            T5<:AbstractVector{<:Integer},
            T6<:AbstractVector{<:Integer},
            T7<:AbstractVector{<:Number},
            T8<:AbstractVector{<:Number},
            T9<:AbstractVector{<:Number},
            T10<:AbstractVector{<:Number},
            T11<:AbstractVector{<:Number} }
function inputdata(task::MSKtask,
                    maxnumcon::T0,
                    maxnumvar::T1,
                    c::T2,
                    cfix::T3,
                    A:: SparseMatrixCSC{Float64},
                    bkc::Vector{Boundkey},
                    blc::T8,
                    buc::T9,
                    bkx::Vector{Boundkey},
                    blx::T10,
                    bux::T11)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number},
            T3<:Number,
            T8<:AbstractVector{<:Number},
            T9<:AbstractVector{<:Number},
            T10<:AbstractVector{<:Number},
            T11<:AbstractVector{<:Number} }
```

Input the linear part of an optimization problem.

The non-zeros of A are inputted column-wise in the format described in Section *Column or Row Ordered Sparse Matrix*.

For an explained code example see Section *Linear Optimization* and Section *Matrix Formats*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumcon` (`Int32`) – Number of preallocated constraints in the optimization task. (input)
- `maxnumvar` (`Int32`) – Number of preallocated variables in the optimization task. (input)
- `c` (`Float64[]`) – Linear terms of the objective as a dense vector. The length is the number of variables. (input)
- `cfix` (`Float64`) – Fixed term in the objective. (input)
- `aptrb` (`Int64[]`) – Row or column start pointers. (input)
- `aptre` (`Int64[]`) – Row or column end pointers. (input)
- `asub` (`Int32[]`) – Coefficient subscripts. (input)
- `aval` (`Float64[]`) – Coefficient values. (input)

- `bkc` (*Boundkey* `[]`) – Bound keys for the constraints. (input)
- `blc` (`Float64[]`) – Lower bounds for the constraints. (input)
- `buc` (`Float64[]`) – Upper bounds for the constraints. (input)
- `bkx` (*Boundkey* `[]`) – Bound keys for the variables. (input)
- `blx` (`Float64[]`) – Lower bounds for the variables. (input)
- `bux` (`Float64[]`) – Upper bounds for the variables. (input)
- `A` (`SparseMatrixCSC{Float64}`) – Sparse matrix defining the column values (input)

Groups

Problem data - linear part, *Problem data - bounds*, *Problem data - constraints*

`isdouparname`

```
function isdouparname(task::MSKtask,
                      parname::AbstractString) -> param :: Dparam
```

Checks whether `parname` is a valid double parameter name.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `parname` (`AbstractString`) – Parameter name. (input)

Return

`param` (*Dparam*) – Returns the parameter corresponding to the name, if one exists.

Groups

Parameters, *Names*

`isintparname`

```
function isintparname(task::MSKtask,
                      parname::AbstractString) -> param :: Iparam
```

Checks whether `parname` is a valid integer parameter name.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `parname` (`AbstractString`) – Parameter name. (input)

Return

`param` (*Iparam*) – Returns the parameter corresponding to the name, if one exists.

Groups

Parameters, *Names*

`isstrparname`

```
function isstrparname(task::MSKtask,
                      parname::AbstractString) -> param :: Sparam
```

Checks whether `parname` is a valid string parameter name.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `parname` (`AbstractString`) – Parameter name. (input)

Return

`param` (*Sparam*) – Returns the parameter corresponding to the name, if one exists.

Groups

Parameters, *Names*

`licensecleanup`

```
function licensecleanup()
```

Stops all threads and deletes all handles used by the license system. If this function is called, it must be called as the last **MOSEK** API call. No other **MOSEK** API calls are valid after this.

Groups
License system

`liinfitemtostr`

```
function liinfitemtostr(item::Liinfitem) -> str :: String
```

Obtains an identifier string corresponding to a information item.

Parameters
`item` (*`Liinfitem`*) – Information item. (input)
Return
`str` (`String`) – String corresponding to the bound information item `item`.
Groups
Names

`linkfiletostream`

```
function linkfiletostream(task::MSKtask,
                          whichstream::Streamtype,
                          filename::AbstractString,
                          append::Int32)
function linkfiletostream(task::MSKtask,
                          whichstream::Streamtype,
                          filename::Union{Nothing,AbstractString},
                          append::T0)
    where { T0<:Integer }
```

Directs all output from a task stream `whichstream` to a file `filename`.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*`Streamtype`*) – Index of the stream. (input)
- `filename` (`AbstractString`) – A valid file name. (input)
- `append` (`Int32`) – If this argument is 0 the output file will be overwritten, otherwise it will be appended to. (input)

Groups
Logging

`linkfiletostream`

```
function linkfiletostream(env::MSKenv,
                          whichstream::Streamtype,
                          filename::AbstractString,
                          append::Int32)
function linkfiletostream(env::MSKenv,
                          whichstream::Streamtype,
                          filename::Union{Nothing,AbstractString},
                          append::T0)
    where { T0<:Integer }
function linkfiletostream(whichstream::Streamtype,
                          filename::AbstractString,
                          append::Int32)
```

(continues on next page)

(continued from previous page)

```
function linkfiletostream(whichstream::Streamtype,
                          filename::Union{Nothing,AbstractString},
                          append::T0)
    where { T0<:Integer }
```

Sends all output from the stream defined by `whichstream` to the file given by `filename`.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)
- `whichstream` (*Streamtype*) – Index of the stream. (input)
- `filename` (`AbstractString`) – A valid file name. (input)
- `append` (`Int32`) – If this argument is 0 the file will be overwritten, otherwise it will be appended to. (input)

Groups

Logging

`makeenv`

```
function makeenv() -> newenv :: Env
```

Creates a new environment.

Return

`newenv` (`MSKenv`) – A new environment.

Groups

Environment and task management

`maketask`

```
function maketask() -> newtask :: Task
function maketask(task::Task) -> newtask :: Task
function maketask(env::Env) -> newtask :: Task
```

Creates a new task.

Parameters

- `task` (`MSKtask`) – A task that will be cloned. (input)
- `env` (`MSKenv`) – Parent environment. (input)

Return

`newtask` (`MSKtask`) – A new task.

Groups

Environment and task management

`onesolutionsummary`

```
function onesolutionsummary(task::MSKtask,
                            whichstream::Streamtype,
                            whichsol::Soltype)
```

Prints a short summary of a specified solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*Streamtype*) – Index of the stream. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)

Groups

Logging, *Solution information*

optimize

```
function optimize(task::MSKtask) -> trmcode :: Rescode
```

Calls the optimizer. Depending on the problem type and the selected optimizer this will call one of the optimizers in **MOSEK**. By default the interior point optimizer will be selected for continuous problems. The optimizer may be selected manually by setting the parameter *MSK_IPAR_OPTIMIZER*.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Return

`trmcode` (*Rescode*) – Is either *MSK_RES_OK* or a termination response code.

Groups

Optimization

optimizebatch

```
function optimizebatch(env::MSKenv,
                       israce::Bool,
                       maxtime::Float64,
                       numthreads::Int32,
                       task::Vector{MSKtask}) -> (trmcode :: Vector{Rescode},
↪rcode :: Vector{Rescode})
function optimizebatch(env::MSKenv,
                       israce::Bool,
                       maxtime::T0,
                       numthreads::T1,
                       task::Vector{MSKtask})
    where { T0<:Number,
            T1<:Integer }
    -> (trmcode :: Vector{Rescode},rcode :: Vector{Rescode})
function optimizebatch(israce::Bool,
                       maxtime::Float64,
                       numthreads::Int32,
                       task::Vector{MSKtask}) -> (trmcode :: Vector{Rescode},
↪rcode :: Vector{Rescode})
function optimizebatch(israce::Bool,
                       maxtime::T0,
                       numthreads::T1,
                       task::Vector{MSKtask})
    where { T0<:Number,
            T1<:Integer }
    -> (trmcode :: Vector{Rescode},rcode :: Vector{Rescode})
```

Optimize a number of tasks in parallel using a specified number of threads. All callbacks and log output streams are disabled.

Assuming that each task takes about same time and there many more tasks than number of threads then a linear speedup can be achieved, also known as strong scaling. A typical application of this method is to solve many small tasks of similar type; in this case it is recommended that each of them is allocated a single thread by setting *MSK_IPAR_NUM_THREADS* to 1.

If the parameters `israce` or `maxtime` are used, then the result may not be deterministic, in the sense that the tasks which complete first may vary between runs.

The remaining behavior, including termination and response codes returned for each task, are the same as if each task was optimized separately.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)

- `israce` (`Bool`) – If nonzero, then the function is terminated after the first task has been completed. (input)
- `maxtime` (`Float64`) – Time limit for the function: if nonnegative, then the function is terminated after maxtime (seconds) has expired. (input)
- `numthreads` (`Int32`) – Number of threads to be employed. (input)
- `task` (`MSKtask[]`) – An array of tasks to optimize in parallel. (input)

Return

- `trmcode` (*`Rescode`*`[]`) – The termination code for each task.
- `rcode` (*`Rescode`*`[]`) – The response code for each task.

Groups

Optimization

`optimizermt`

```
function optimizermt(task::MSKtask,
                     address::AbstractString,
                     accesstoken::AbstractString) -> trmcode :: Rescode
```

Offload the optimization task to an instance of OptServer specified by `addr`, which should be a valid URL, for example `http://server:port` or `https://server:port`. The call will block until a result is available or the connection closes.

If the server requires authentication, the authentication token can be passed in the `accesstoken` argument.

If the server requires encryption, the keys can be passed using one of the solver parameters *`MSK_SPAR_REMOTE_TLS_CERT`* or *`MSK_SPAR_REMOTE_TLS_CERT_PATH`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `address` (`AbstractString`) – Address of the OptServer. (input)
- `accesstoken` (`AbstractString`) – Access token. (input)

Return

`trmcode` (*`Rescode`*) – Is either *`MSK_RES_OK`* or a termination response code.

Groups

Remote optimization

`optimizersummary`

```
function optimizersummary(task::MSKtask,
                          whichstream::Streamtype)
```

Prints a short summary with optimizer statistics from last optimization.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*`Streamtype`*) – Index of the stream. (input)

Groups

Logging

`primalrepair`

```
function primalrepair(task::MSKtask,
                      wlc::Union{Nothing,Vector{Float64}},
                      wuc::Union{Nothing,Vector{Float64}},
                      wlx::Union{Nothing,Vector{Float64}},
                      wux::Union{Nothing,Vector{Float64}})
function primalrepair(task::MSKtask,
```

(continues on next page)

(continued from previous page)

```
                      wlc::T0,
                      wuc::T1,
                      wlx::T2,
                      wux::T3)
    where { T0<:AbstractVector{<:Number},
            T1<:AbstractVector{<:Number},
            T2<:AbstractVector{<:Number},
            T3<:AbstractVector{<:Number} }
```

The function repairs a primal infeasible optimization problem by adjusting the bounds on the constraints and variables where the adjustment is computed as the minimal weighted sum of relaxations to the bounds on the constraints and variables. Observe the function only repairs the problem but does not solve it. If an optimal solution is required the problem should be optimized after the repair.

The function is applicable to linear and conic problems possibly with integer variables.

Observe that when computing the minimal weighted relaxation the termination tolerance specified by the parameters of the task is employed. For instance the parameter *MSK_IPAR_MIO_MODE* can be used to make **MOSEK** ignore the integer constraints during the repair which usually leads to a much faster repair. However, the drawback is of course that the repaired problem may not have an integer feasible solution.

Note the function modifies the task in place. If this is not desired, then apply the function to a cloned task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `wlc` (`Float64[]`) – $(w_l^c)_i$ is the weight associated with relaxing the lower bound on constraint i. If the weight is negative, then the lower bound is not relaxed. Moreover, if the argument is `nothing`, then all the weights are assumed to be 1. (input)
- `wuc` (`Float64[]`) – $(w_u^c)_i$ is the weight associated with relaxing the upper bound on constraint i. If the weight is negative, then the upper bound is not relaxed. Moreover, if the argument is `nothing`, then all the weights are assumed to be 1. (input)
- `wlx` (`Float64[]`) – $(w_l^x)_j$ is the weight associated with relaxing the lower bound on variable j. If the weight is negative, then the lower bound is not relaxed. Moreover, if the argument is `nothing`, then all the weights are assumed to be 1. (input)
- `wux` (`Float64[]`) – $(w_l^x)_i$ is the weight associated with relaxing the upper bound on variable j. If the weight is negative, then the upper bound is not relaxed. Moreover, if the argument is `nothing`, then all the weights are assumed to be 1. (input)

Groups

Infeasibility diagnostic

`primalsensitivity`

```
function primalsensitivity(task::MSKtask,
                           subi::Vector{Int32},
                           marki::Vector{Mark},
                           subj::Vector{Int32},
                           markj::Vector{Mark}) -> (leftpricei :: Vector{Float64}
↪,rightpricei :: Vector{Float64},leftrangei :: Vector{Float64},rightrangei ::␣
↪Vector{Float64},leftpricej :: Vector{Float64},rightpricej :: Vector{Float64},
↪leftrangej :: Vector{Float64},rightrangej :: Vector{Float64})
function primalsensitivity(task::MSKtask,
```

(continues on next page)

(continued from previous page)

```
                          subi::T0,
                          marki::Vector{Mark},
                          subj::T1,
                          markj::Vector{Mark})
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer} }
    -> (leftpricei :: Vector{Float64},rightpricei :: Vector{Float64},leftrangei :
↪: Vector{Float64},rightrangei :: Vector{Float64},leftpricej :: Vector{Float64},
↪rightpricej :: Vector{Float64},leftrangej :: Vector{Float64},rightrangej ::␣
↪Vector{Float64})
```

Calculates sensitivity information for bounds on variables and constraints. For details on sensitivity analysis, the definitions of *shadow price* and *linearity interval* and an example see Section *Sensitivity Analysis*.

The type of sensitivity analysis to be performed (basis or optimal partition) is controlled by the parameter *MSK_IPAR_SENSITIVITY_TYPE*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subi` (`Int32[]`) – Indexes of constraints to analyze. (input)
- `marki` (*Mark* `[]`) – The value of `marki[i]` indicates for which bound of constraint `subi[i]` sensitivity analysis is performed. If `marki[i]` = *MSK_MARK_UP* the upper bound of constraint `subi[i]` is analyzed, and if `marki[i]` = *MSK_MARK_LO* the lower bound is analyzed. If `subi[i]` is an equality constraint, either *MSK_MARK_LO* or *MSK_MARK_UP* can be used to select the constraint for sensitivity analysis. (input)
- `subj` (`Int32[]`) – Indexes of variables to analyze. (input)
- `markj` (*Mark* `[]`) – The value of `markj[j]` indicates for which bound of variable `subj[j]` sensitivity analysis is performed. If `markj[j]` = *MSK_MARK_UP* the upper bound of variable `subj[j]` is analyzed, and if `markj[j]` = *MSK_MARK_LO* the lower bound is analyzed. If `subj[j]` is a fixed variable, either *MSK_MARK_LO* or *MSK_MARK_UP* can be used to select the bound for sensitivity analysis. (input)

Return

- `leftpricei` (`Float64[]`) – `leftpricei[i]` is the left shadow price for the bound `marki[i]` of constraint `subi[i]`.
- `rightpricei` (`Float64[]`) – `rightpricei[i]` is the right shadow price for the bound `marki[i]` of constraint `subi[i]`.
- `leftrangei` (`Float64[]`) – `leftrangei[i]` is the left range β_1 for the bound `marki[i]` of constraint `subi[i]`.
- `rightrangei` (`Float64[]`) – `rightrangei[i]` is the right range β_2 for the bound `marki[i]` of constraint `subi[i]`.
- `leftpricej` (`Float64[]`) – `leftpricej[j]` is the left shadow price for the bound `markj[j]` of variable `subj[j]`.
- `rightpricej` (`Float64[]`) – `rightpricej[j]` is the right shadow price for the bound `markj[j]` of variable `subj[j]`.
- `leftrangej` (`Float64[]`) – `leftrangej[j]` is the left range β_1 for the bound `markj[j]` of variable `subj[j]`.
- `rightrangej` (`Float64[]`) – `rightrangej[j]` is the right range β_2 for the bound `markj[j]` of variable `subj[j]`.

Groups

Sensitivity analysis

`printparam`

```
function printparam(task::MSKtask)
```

Prints the current parameter settings to the message stream.

Parameters
`task` (`MSKtask`) – An optimization task. (input)

Groups
Inspecting the task, *Logging*

`probtypetostr`

```
function probtypetostr(task::MSKtask,
                       probtype::Problemtype) -> str :: String
```

Obtains a string containing the name of a given problem type.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `probtype` (*`Problemtype`*) – Problem type. (input)

Return
`str` (`String`) – String corresponding to the problem type key `probtype`.

Groups
Inspecting the task, *Names*

`prostatostr`

```
function prostatostr(task::MSKtask,
                     problemsta::Prosta) -> str :: String
```

Obtains a string containing the name of a given problem status.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `problemsta` (*`Prosta`*) – Problem status. (input)

Return
`str` (`String`) – String corresponding to the status key `prosta`.

Groups
Solution information, *Names*

`putacc`

```
function putacc(task::MSKtask,
                accidx::Int64,
                domidx::Int64,
                afeidxlist::Vector{Int64},
                b::Union{Nothing,Vector{Float64}})
function putacc(task::MSKtask,
                accidx::T0,
                domidx::T1,
                afeidxlist::T2,
                b::T3)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number} }
```

Puts an affine conic constraint. This method overwrites an existing affine conic constraint number `accidx` with new data specified in the same format as in *`appendacc`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `accidx` (`Int64`) – Affine conic constraint index. (input)
- `domidx` (`Int64`) – Domain index. (input)
- `afeidxlist` (`Int64[]`) – List of affine expression indexes. (input)
- `b` (`Float64[]`) – The vector of constant terms modifying affine expressions. Optional, can be `nothing` if not required. (input)

Groups

Problem data - affine conic constraints

`putaccb`

```
function putaccb(task::MSKtask,
                 accidx::Int64,
                 b::Union{Nothing,Vector{Float64}})
function putaccb(task::MSKtask,
                 accidx::T0,
                 b::T1)
    where { T0<:Integer,
            T1<:AbstractVector{<:Number} }
```

Updates an existing affine conic constraint number `accidx` by putting a new vector b.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `accidx` (`Int64`) – Affine conic constraint index. (input)
- `b` (`Float64[]`) – The vector of constant terms modifying affine expressions. Optional, can be `nothing` if not required. (input)

Groups

Problem data - affine conic constraints

`putaccbj`

```
function putaccbj(task::MSKtask,
                  accidx::Int64,
                  j::Int64,
                  bj::Float64)
function putaccbj(task::MSKtask,
                  accidx::T0,
                  j::T1,
                  bj::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Number }
```

Sets one value $b[j]$ in the b vector for the affine conic constraint number `accidx`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `accidx` (`Int64`) – Affine conic constraint index. (input)
- `j` (`Int64`) – The index of an element in b to change. (input)
- `bj` (`Float64`) – The new value of $b[j]$. (input)

Groups

Problem data - affine conic constraints

`putaccdoty`

```
function putaccdoty(task::MSKtask,
                    whichsol::Soltype,
                    accidx::Int64) -> doty :: Vector{Float64}
function putaccdoty(task::MSKtask,
                    whichsol::Soltype,
                    accidx::T0)
    where { T0<:Integer }
    -> doty :: Vector{Float64}
```

Puts the $\dot{y}$ vector for a solution (the dual values of an affine conic constraint).

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `accidx` (`Int64`) – The index of the affine conic constraint. (input)

Return

`doty` (`Float64[]`) – The dual values for this affine conic constraint. The array should have length equal to the dimension of the constraint.

Groups

Solution - dual, *Problem data - affine conic constraints*

`putacclist`

```
function putacclist(task::MSKtask,
                    accidxs::Vector{Int64},
                    domidxs::Vector{Int64},
                    afeidxlist::Vector{Int64},
                    b::Union{Nothing,Vector{Float64}})
function putacclist(task::MSKtask,
                    accidxs::T0,
                    domidxs::T1,
                    afeidxlist::T2,
                    b::T3)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number} }
```

Puts affine conic constraints. This method overwrites existing affine conic constraints whose numbers are provided in the list `accidxs` with new data which is a concatenation of individual constraint descriptions in the same format as in *`appendacc`* (see also *`appendaccs`*).

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `accidxs` (`Int64[]`) – Affine conic constraint indices. (input)
- `domidxs` (`Int64[]`) – Domain indices. (input)
- `afeidxlist` (`Int64[]`) – List of affine expression indexes. (input)
- `b` (`Float64[]`) – The vector of constant terms modifying affine expressions. Optional, can be `nothing` if not required. (input)

Groups

Problem data - affine conic constraints

`putaccname`

```
function putaccname(task::MSKtask,
                    accidx::Int64,
                    name::Union{Nothing,AbstractString})
```

(continues on next page)

(continued from previous page)

```
function putaccname(task::MSKtask,
                    accidx::T0,
                    name::Union{Nothing,AbstractString})
    where { T0<:Integer }
```

Sets the name of an affine conic constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `accidx` (`Int64`) – Index of the affine conic constraint. (input)
- `name` (`AbstractString`) – The name of the affine conic constraint. (input)

Groups

Names, *Problem data - affine conic constraints*

`putacol`

```
function putacol(task::MSKtask,
                 j::Int32,
                 subj::Vector{Int32},
                 valj::Vector{Float64})
function putacol(task::MSKtask,
                 j::T0,
                 subj::T1,
                 valj::T2)
    where { T0<:Integer,
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
```

Change one column of the linear constraint matrix A. Resets all the elements in column j to zero and then sets

$$a_{\texttt{subj}[k],\texttt{j}} = \texttt{valj}[k], \quad k = 1, \ldots, \texttt{nzj}.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of a column in A. (input)
- `subj` (`Int32[]`) – Row indexes of non-zero values in column j of A. (input)
- `valj` (`Float64[]`) – New non-zero values of column j in A. (input)

Groups

Problem data - linear part

`putacollist`

```
function putacollist(task::MSKtask,
                     sub::Vector{Int32},
                     ptrb::Vector{Int64},
                     ptre::Vector{Int64},
                     asub::Vector{Int32},
                     aval::Vector{Float64})
function putacollist(task::MSKtask,
                     sub::T0,
                     ptrb::T1,
                     ptre::T2,
                     asub::T3,
                     aval::T4)
    where { T0<:AbstractVector{<:Integer},
```

(continues on next page)

(continued from previous page)

```
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Number} }
function putacollist(task::MSKtask,
                     sub::T0,
                     A:: SparseMatrixCSC{Float64})
    where { T0<:AbstractVector{<:Integer} }
```

Change a set of columns in the linear constraint matrix A with data in sparse triplet format. The requested columns are set to zero and then updated with:

$$\begin{array}{ll} \texttt{for} & i = 1, \ldots, \texttt{num} \\ & a_{\texttt{asub}[k],\texttt{sub}[i]} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i], \ldots, \texttt{ptre}[i] - 1. \end{array}$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `sub` (`Int32[]`) – Indexes of columns that should be replaced, no duplicates. (input)
- `ptrb` (`Int64[]`) – Array of pointers to the first element in each column. (input)
- `ptre` (`Int64[]`) – Array of pointers to the last element plus one in each column. (input)
- `asub` (`Int32[]`) – Row indexes of new elements. (input)
- `aval` (`Float64[]`) – Coefficient values. (input)
- `A` (`SparseMatrixCSC{Float64}`) – Sparse matrix defining the column values (input)

Groups

Problem data - linear part

`putacolslice`

```
function putacolslice(task::MSKtask,
                      first::Int32,
                      last::Int32,
                      ptrb::Vector{Int64},
                      ptre::Vector{Int64},
                      asub::Vector{Int32},
                      aval::Vector{Float64})
function putacolslice(task::MSKtask,
                      first::T0,
                      last::T1,
                      ptrb::T2,
                      ptre::T3,
                      asub::T4,
                      aval::T5)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Integer},
            T5<:AbstractVector{<:Number} }
function putacolslice(task::MSKtask,
                      first::T0,
                      last::T1,
                      A:: SparseMatrixCSC{Float64})
    where { T0<:Integer,
            T1<:Integer }
```

Change a slice of columns in the linear constraint matrix A with data in sparse triplet format. The requested columns are set to zero and then updated with:

$$\begin{array}{ll}\texttt{for} & i = \texttt{first}, \ldots, \texttt{last} - 1 \\ & a_{\texttt{asub}[k],i} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i - \texttt{first} + 1], \ldots, \texttt{ptre}[i - \texttt{first} + 1] - 1.\end{array}$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – First column in the slice. (input)
- `last` (`Int32`) – Last column plus one in the slice. (input)
- `ptrb` (`Int64[]`) – Array of pointers to the first element in each column. (input)
- `ptre` (`Int64[]`) – Array of pointers to the last element plus one in each column. (input)
- `asub` (`Int32[]`) – Row indexes of new elements. (input)
- `aval` (`Float64[]`) – Coefficient values. (input)
- `A` (`SparseMatrixCSC{Float64}`) – Sparse matrix defining the column values (input)

Groups

Problem data - linear part

`putafebarfblocktriplet`

```
function putafebarfblocktriplet(task::MSKtask,
                                afeidx::Vector{Int64},
                                barvaridx::Vector{Int32},
                                subk::Vector{Int32},
                                subl::Vector{Int32},
                                valkl::Vector{Float64})
function putafebarfblocktriplet(task::MSKtask,
                                afeidx::T0,
                                barvaridx::T1,
                                subk::T2,
                                subl::T3,
                                valkl::T4)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Number} }
```

Inputs the $\overline{F}$ matrix data in block triplet form.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64[]`) – Constraint index. (input)
- `barvaridx` (`Int32[]`) – Symmetric matrix variable index. (input)
- `subk` (`Int32[]`) – Block row index. (input)
- `subl` (`Int32[]`) – Block column index. (input)
- `valkl` (`Float64[]`) – The numerical value associated with each block triplet. (input)

Groups

Problem data - affine expressions, *Problem data - semidefinite*

`putafebarfentry`

```
function putafebarfentry(task::MSKtask,
                         afeidx::Int64,
                         barvaridx::Int32,
                         termidx::Vector{Int64},
                         termweight::Vector{Float64})
function putafebarfentry(task::MSKtask,
                         afeidx::T0,
                         barvaridx::T1,
                         termidx::T2,
                         termweight::T3)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number} }
```

This function sets one entry $\overline{F}_{ij}$ where $i =$ afeidx is the row index in the store of affine expressions and $j =$ barvaridx is the index of a symmetric variable. That is, the expression

$$\langle \overline{F}_{ij}, \overline{X}_j \rangle$$

will be added to the i-th affine expression.

The matrix $\overline{F}_{ij}$ is specified as a weighted sum of symmetric matrices from the symmetric matrix storage E, so $\overline{F}_{ij}$ is a symmetric matrix, precisely:

$$\overline{F}_{\text{afeidx,barvaridx}} = \sum_k \text{termweight}[k] \cdot E_{\text{termidx}[k]}.$$

By default all elements in $\overline{F}$ are 0, so only non-zero elements need be added. Setting the same entry again will overwrite the earlier entry.

The symmetric matrices from E are defined separately using the function *appendsparsesymmat*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Row index of $\overline{F}$. (input)
- `barvaridx` (`Int32`) – Semidefinite variable index. (input)
- `termidx` (`Int64[]`) – Indices in E of the matrices appearing in the weighted sum for the $\overline{F}$ entry being specified. (input)
- `termweight` (`Float64[]`) – `termweight[k]` is the coefficient of the `termidx[k]`-th element of E in the weighted sum the $\overline{F}$ entry being specified. (input)

Groups

Problem data - affine expressions, *Problem data - semidefinite*

`putafebarfentrylist`

```
function putafebarfentrylist(task::MSKtask,
                             afeidx::Vector{Int64},
                             barvaridx::Vector{Int32},
                             numterm::Vector{Int64},
                             ptrterm::Vector{Int64},
                             termidx::Vector{Int64},
                             termweight::Vector{Float64})
function putafebarfentrylist(task::MSKtask,
                             afeidx::T0,
                             barvaridx::T1,
                             numterm::T2,
                             ptrterm::T3,
                             termidx::T4,
```

(continues on next page)

(continued from previous page)

```
                              termweight::T5)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Integer},
            T5<:AbstractVector{<:Number} }
```

This function sets a list of entries in $\overline{F}$. Each entry should be described as in *putafebarfentry* and all those descriptions should be combined (for example concatenated) in the input to this method. That means the k-th entry set will have row index `afeidx[k]`, symmetric variable index `barvaridx[k]` and the description of this term consists of indices in E and weights appearing in positions

$$\text{ptrterm}[k], \ldots, \text{ptrterm}[k] + (\text{lenterm}[k] - 1)$$

in the corresponding arrays `termidx` and `termweight`. See *putafebarfentry* for details.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64[]`) – Row indexes of $\overline{F}$. (input)
- `barvaridx` (`Int32[]`) – Semidefinite variable indexes. (input)
- `numterm` (`Int64[]`) – The number of terms in the weighted sums that form each entry. (input)
- `ptrterm` (`Int64[]`) – The pointer to the beginning of the description of each entry. (input)
- `termidx` (`Int64[]`) – Concatenated lists of indices in E of the matrices appearing in the weighted sums for the $\overline{F}$ being specified. (input)
- `termweight` (`Float64[]`) – Concatenated lists of weights appearing in the weighted sums forming the $\overline{F}$ elements being specified. (input)

Groups

Problem data - affine expressions, *Problem data - semidefinite*

`putafebarfrow`

```
function putafebarfrow(task::MSKtask,
                       afeidx::Int64,
                       barvaridx::Vector{Int32},
                       numterm::Vector{Int64},
                       ptrterm::Vector{Int64},
                       termidx::Vector{Int64},
                       termweight::Vector{Float64})
function putafebarfrow(task::MSKtask,
                       afeidx::T0,
                       barvaridx::T1,
                       numterm::T2,
                       ptrterm::T3,
                       termidx::T4,
                       termweight::T5)
    where { T0<:Integer,
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Integer},
            T5<:AbstractVector{<:Number} }
```

This function inputs one row in $\overline{F}$. It first clears the row, i.e. sets $\overline{F}_{\mathtt{afeidx},*} = 0$ and then sets the new entries. Each entry should be described as in *putafebarfentry* and all those descriptions should be combined (for example concatenated) in the input to this method. That means the k-th entry set will have row index `afeidx`, symmetric variable index `barvaridx[k]` and the description of this term consists of indices in E and weights appearing in positions

$$\mathtt{ptrterm}[k], \ldots, \mathtt{ptrterm}[k] + (\mathtt{numterm}[k] - 1)$$

in the corresponding arrays `termidx` and `termweight`. See *putafebarfentry* for details.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Row index of $\overline{F}$. (input)
- `barvaridx` (`Int32[]`) – Semidefinite variable indexes. (input)
- `numterm` (`Int64[]`) – The number of terms in the weighted sums that form each entry. (input)
- `ptrterm` (`Int64[]`) – The pointer to the beginning of the description of each entry. (input)
- `termidx` (`Int64[]`) – Concatenated lists of indices in E of the matrices appearing in the weighted sums for the $\overline{F}$ entries in the row. (input)
- `termweight` (`Float64[]`) – Concatenated lists of weights appearing in the weighted sums forming the $\overline{F}$ entries in the row. (input)

Groups

Problem data - affine expressions, *Problem data - semidefinite*

`putafefcol`

```
function putafefcol(task::MSKtask,
                    varidx::Int32,
                    afeidx::Vector{Int64},
                    val::Vector{Float64})
function putafefcol(task::MSKtask,
                    varidx::T0,
                    afeidx::T1,
                    val::T2)
    where { T0<:Integer,
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
```

Change one column of the matrix F of affine expressions. Resets all the elements in column `varidx` to zero and then sets

$$F_{\mathtt{afeidx}[k],\mathtt{varidx}} = \mathtt{val}[k], \quad k = 1, \ldots, \mathtt{numnz}.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `varidx` (`Int32`) – Index of a column in F. (input)
- `afeidx` (`Int64[]`) – Row indexes of non-zero values in the column of F. (input)
- `val` (`Float64[]`) – New non-zero values in the column of F. (input)

Groups

Problem data - affine expressions

`putafefentry`

```
function putafefentry(task::MSKtask,
                      afeidx::Int64,
                      varidx::Int32,
```

(continues on next page)

(continued from previous page)

```
                        value::Float64)
function putafefentry(task::MSKtask,
                        afeidx::T0,
                        varidx::T1,
                        value::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Number }
```

Replaces one entry in the affine expression store F, that is it sets:

$$F_{\text{afeidx,varidx}} = \text{value}.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Row index in F. (input)
- `varidx` (`Int32`) – Column index in F. (input)
- `value` (`Float64`) – Value of $F_{\text{afeidx,varidx}}$. (input)

Groups

Problem data - affine expressions

`putafefentrylist`

```
function putafefentrylist(task::MSKtask,
                          afeidx::Vector{Int64},
                          varidx::Vector{Int32},
                          val::Vector{Float64})
function putafefentrylist(task::MSKtask,
                          afeidx::T0,
                          varidx::T1,
                          val::T2)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
```

Replaces a number of entries in the affine expression store F, that is it sets:

$$F_{\text{afeidxs}[k],\text{varidx}[k]} = \text{val}[k]$$

for all k.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64[]`) – Row indices in F. (input)
- `varidx` (`Int32[]`) – Column indices in F. (input)
- `val` (`Float64[]`) – Values of the entries in F. (input)

Groups

Problem data - affine expressions

`putafefrow`

```
function putafefrow(task::MSKtask,
                    afeidx::Int64,
                    varidx::Vector{Int32},
                    val::Vector{Float64})
function putafefrow(task::MSKtask,
```

(continues on next page)

(continued from previous page)

```
                    afeidx::T0,
                    varidx::T1,
                    val::T2)
    where { T0<:Integer,
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
```

Change one row of the matrix F of affine expressions. Resets all the elements in row `afeidx` to zero and then sets

$$F_{\texttt{afeidx},\texttt{varidx}[k]} = \texttt{val}[k], \quad k = 1, \ldots, \texttt{numnz}.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Index of a row in F. (input)
- `varidx` (`Int32[]`) – Column indexes of non-zero values in the row of F. (input)
- `val` (`Float64[]`) – New non-zero values in the row of F. (input)

Groups

Problem data - affine expressions

`putafefrowlist`

```
function putafefrowlist(task::MSKtask,
                        afeidx::Vector{Int64},
                        numnzrow::Vector{Int32},
                        ptrrow::Vector{Int64},
                        varidx::Vector{Int32},
                        val::Vector{Float64})
function putafefrowlist(task::MSKtask,
                        afeidx::T0,
                        numnzrow::T1,
                        ptrrow::T2,
                        varidx::T3,
                        val::T4)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Number} }
```

Clears and then changes a number of rows of the matrix F of affine expressions. The k-th of the rows to be changed has index $i = \text{afeidx}[k]$, contains $\text{numnzrow}[k]$ nonzeros and its description as in *putafefrow* starts in position $\text{ptrrow}[k]$ of the arrays `varidx` and `val`. Formally, the row with index i is cleared and then set as:

$$F_{i,\text{varidx}[\text{ptrrow}[k]+j]} = \text{val}[\text{ptrrow}[k] + j], \quad j = 0, \ldots, \text{numnzrow}[k] - 1.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64[]`) – Indices of rows in F. (input)
- `numnzrow` (`Int32[]`) – Number of non-zeros in each of the modified rows of F. (input)
- `ptrrow` (`Int64[]`) – Pointer to the first nonzero in each row of F. (input)
- `varidx` (`Int32[]`) – Column indexes of non-zero values. (input)
- `val` (`Float64[]`) – New non-zero values in the rows of F. (input)

Groups

Problem data - affine expressions

putafeg

```
function putafeg(task::MSKtask,
                 afeidx::Int64,
                 g::Float64)
function putafeg(task::MSKtask,
                 afeidx::T0,
                 g::T1)
    where { T0<:Integer,
            T1<:Number }
```

Change one element of the vector g in affine expressions i.e.

$$g_{\texttt{afeidx}} = \texttt{gi}.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64`) – Index of an entry in g. (input)
- `g` (`Float64`) – New value for $g_{\texttt{afeidx}}$. (input)

Groups

Problem data - affine expressions

putafeglist

```
function putafeglist(task::MSKtask,
                     afeidx::Vector{Int64},
                     g::Vector{Float64})
function putafeglist(task::MSKtask,
                     afeidx::T0,
                     g::T1)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Number} }
```

Changes a list of elements of the vector g in affine expressions i.e. for all k it sets

$$g_{\texttt{afeidx}[k]} = \texttt{glist}[k].$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `afeidx` (`Int64[]`) – Indices of entries in g. (input)
- `g` (`Float64[]`) – New values for g. (input)

Groups

Problem data - affine expressions

putafegslice

```
function putafegslice(task::MSKtask,
                      first::Int64,
                      last::Int64,
                      slice::Vector{Float64})
function putafegslice(task::MSKtask,
                      first::T0,
                      last::T1,
                      slice::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Modifies a slice in the vector g of constant terms in affine expressions using the principle

$$g_{\mathtt{j}} = \mathtt{slice}[\mathtt{j} - \mathtt{first} + 1], \quad j = \text{first}, .., \text{last} - 1$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int64`) – First index in the sequence. (input)
- `last` (`Int64`) – Last index plus 1 in the sequence. (input)
- `slice` (`Float64[]`) – The slice of g as a dense vector. The length is `last-first`. (input)

Groups

Problem data - affine expressions

`putaij`

```
function putaij(task::MSKtask,
                i::Int32,
                j::Int32,
                aij::Float64)
function putaij(task::MSKtask,
                i::T0,
                j::T1,
                aij::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Number }
```

Changes a coefficient in the linear coefficient matrix A using the method

$$a_{i,j} = \mathtt{aij}.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Constraint (row) index. (input)
- `j` (`Int32`) – Variable (column) index. (input)
- `aij` (`Float64`) – New coefficient for $a_{i,j}$. (input)

Groups

Problem data - linear part

`putaijlist`

```
function putaijlist(task::MSKtask,
                    subi::Vector{Int32},
                    subj::Vector{Int32},
                    valij::Vector{Float64})
function putaijlist(task::MSKtask,
                    subi::T0,
                    subj::T1,
                    valij::T2)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
```

Changes one or more coefficients in A using the method

$$a_{\mathtt{subi[k]},\mathtt{subj[k]}} = \mathtt{valij}[\mathtt{k}], \quad k = 1, \ldots, \mathtt{num}.$$

Duplicates are not allowed.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subi` (`Int32[]`) – Constraint (row) indices. (input)
- `subj` (`Int32[]`) – Variable (column) indices. (input)
- `valij` (`Float64[]`) – New coefficient values for $a_{i,j}$. (input)

Groups

Problem data - linear part

`putarow`

```
function putarow(task::MSKtask,
                 i::Int32,
                 subi::Vector{Int32},
                 vali::Vector{Float64})
function putarow(task::MSKtask,
                 i::T0,
                 subi::T1,
                 vali::T2)
    where { T0<:Integer,
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
```

Change one row of the linear constraint matrix A. Resets all the elements in row i to zero and then sets

$$a_{\texttt{i},\texttt{subi}[k]} = \texttt{vali}[k], \quad k = 1, \ldots, \texttt{nzi}.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of a row in A. (input)
- `subi` (`Int32[]`) – Column indexes of non-zero values in row i of A. (input)
- `vali` (`Float64[]`) – New non-zero values of row i in A. (input)

Groups

Problem data - linear part

`putarowlist`

```
function putarowlist(task::MSKtask,
                     sub::Vector{Int32},
                     ptrb::Vector{Int64},
                     ptre::Vector{Int64},
                     asub::Vector{Int32},
                     aval::Vector{Float64})
function putarowlist(task::MSKtask,
                     sub::T0,
                     ptrb::T1,
                     ptre::T2,
                     asub::T3,
                     aval::T4)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Number} }
function putarowlist(task::MSKtask,
                     sub::T0,
                     At:: SparseMatrixCSC{Float64})
    where { T0<:AbstractVector{<:Integer} }
```

Change a set of rows in the linear constraint matrix A with data in sparse triplet format. The requested rows are set to zero and then updated with:

$$\begin{array}{ll} \texttt{for} & i = 1, \ldots, \texttt{num} \\ & a_{\texttt{sub}[i],\texttt{asub}[k]} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i], \ldots, \texttt{ptre}[i] - 1. \end{array}$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `sub` (`Int32[]`) – Indexes of rows that should be replaced, no duplicates. (input)
- `ptrb` (`Int64[]`) – Array of pointers to the first element in each row. (input)
- `ptre` (`Int64[]`) – Array of pointers to the last element plus one in each row. (input)
- `asub` (`Int32[]`) – Column indexes of new elements. (input)
- `aval` (`Float64[]`) – Coefficient values. (input)
- `At` (`SparseMatrixCSC{Float64}`) – Transposed matrix defining the row values. Note that for efficiency reasons the *columns* of this matrix defines the *rows* to be replaced (input)

Groups

Problem data - linear part

`putarowslice`

```
function putarowslice(task::MSKtask,
                      first::Int32,
                      last::Int32,
                      ptrb::Vector{Int64},
                      ptre::Vector{Int64},
                      asub::Vector{Int32},
                      aval::Vector{Float64})
function putarowslice(task::MSKtask,
                      first::T0,
                      last::T1,
                      ptrb::T2,
                      ptre::T3,
                      asub::T4,
                      aval::T5)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Integer},
            T5<:AbstractVector{<:Number} }
function putarowslice(task::MSKtask,
                      first::T0,
                      last::T1,
                      At:: SparseMatrixCSC{Float64})
    where { T0<:Integer,
            T1<:Integer }
```

Change a slice of rows in the linear constraint matrix A with data in sparse triplet format. The requested rows are set to zero and then updated with:

$$\begin{array}{ll} \texttt{for} & i = \texttt{first}, \ldots, \texttt{last} - 1 \\ & a_{i,\texttt{asub}[k]} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i - \texttt{first} + 1], \ldots, \texttt{ptre}[i - \texttt{first} + 1] - 1. \end{array}$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – First row in the slice. (input)

- `last` (`Int32`) – Last row plus one in the slice. (input)
- `ptrb` (`Int64[]`) – Array of pointers to the first element in each row. (input)
- `ptre` (`Int64[]`) – Array of pointers to the last element plus one in each row. (input)
- `asub` (`Int32[]`) – Column indexes of new elements. (input)
- `aval` (`Float64[]`) – Coefficient values. (input)
- `At` (`SparseMatrixCSC{Float64}`) – Transposed matrix defining the row values. Note that for efficiency reasons the *columns* of this matrix defines the *rows* to be replaced (input)

Groups

Problem data - linear part

`putatruncatetol`

```
function putatruncatetol(task::MSKtask,
                         tolzero::Float64)
function putatruncatetol(task::MSKtask,
                         tolzero::T0)
    where { T0<:Number }
```

Truncates (sets to zero) all elements in A that satisfy

$$|a_{i,j}| \leq \texttt{tolzero}.$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `tolzero` (`Float64`) – Truncation tolerance. (input)

Groups

Problem data - linear part

`putbarablocktriplet`

```
function putbarablocktriplet(task::MSKtask,
                             subi::Vector{Int32},
                             subj::Vector{Int32},
                             subk::Vector{Int32},
                             subl::Vector{Int32},
                             valijkl::Vector{Float64})
function putbarablocktriplet(task::MSKtask,
                             subi::T0,
                             subj::T1,
                             subk::T2,
                             subl::T3,
                             valijkl::T4)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Number} }
```

Inputs the $\overline{A}$ matrix in block triplet form.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subi` (`Int32[]`) – Constraint index. (input)
- `subj` (`Int32[]`) – Symmetric matrix variable index. (input)
- `subk` (`Int32[]`) – Block row index. (input)
- `subl` (`Int32[]`) – Block column index. (input)

- `valijkl` (`Float64[]`) – The numerical value associated with each block triplet. (input)

Groups

Problem data - semidefinite

`putbaraij`

```
function putbaraij(task::MSKtask,
                   i::Int32,
                   j::Int32,
                   sub::Vector{Int64},
                   weights::Vector{Float64})
function putbaraij(task::MSKtask,
                   i::T0,
                   j::T1,
                   sub::T2,
                   weights::T3)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number} }
```

This function sets one element in the $\overline{A}$ matrix.

Each element in the $\overline{A}$ matrix is a weighted sum of symmetric matrices from the symmetric matrix storage E, so $\overline{A}_{ij}$ is a symmetric matrix. By default all elements in $\overline{A}$ are 0, so only non-zero elements need be added. Setting the same element again will overwrite the earlier entry.

The symmetric matrices from E are defined separately using the function *appendsparsesymmat*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Row index of $\overline{A}$. (input)
- `j` (`Int32`) – Column index of $\overline{A}$. (input)
- `sub` (`Int64[]`) – Indices in E of the matrices appearing in the weighted sum for $\overline{A}_{ij}$. (input)
- `weights` (`Float64[]`) – `weights[k]` is the coefficient of the `sub[k]`-th element of E in the weighted sum forming $\overline{A}_{ij}$. (input)

Groups

Problem data - semidefinite

`putbaraijlist`

```
function putbaraijlist(task::MSKtask,
                       subi::Vector{Int32},
                       subj::Vector{Int32},
                       alphaptrb::Vector{Int64},
                       alphaptre::Vector{Int64},
                       matidx::Vector{Int64},
                       weights::Vector{Float64})
function putbaraijlist(task::MSKtask,
                       subi::T0,
                       subj::T1,
                       alphaptrb::T2,
                       alphaptre::T3,
                       matidx::T4,
                       weights::T5)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
```

(continues on next page)

(continued from previous page)

```
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Integer},
            T5<:AbstractVector{<:Number} }
function putbaraijlist(task::MSKtask,
                       subi::T0,
                       subj::T1,
                       A:: SparseMatrixCSC{Float64})
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer} }
```

This function sets a list of elements in the $\overline{A}$ matrix.

Each element in the $\overline{A}$ matrix is a weighted sum of symmetric matrices from the symmetric matrix storage E, so $\overline{A}_{ij}$ is a symmetric matrix. By default all elements in $\overline{A}$ are 0, so only non-zero elements need be added. Setting the same element again will overwrite the earlier entry.

The symmetric matrices from E are defined separately using the function *appendsparsesymmat*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subi` (`Int32[]`) – Row index of $\overline{A}$. (input)
- `subj` (`Int32[]`) – Column index of $\overline{A}$. (input)
- `alphaptrb` (`Int64[]`) – Start entries for terms in the weighted sum that forms $\overline{A}_{ij}$. (input)
- `alphaptre` (`Int64[]`) – End entries for terms in the weighted sum that forms $\overline{A}_{ij}$. (input)
- `matidx` (`Int64[]`) – Indices in E of the matrices appearing in the weighted sum for $\overline{A}_{ij}$. (input)
- `weights` (`Float64[]`) – `weights[k]` is the coefficient of the `sub[k]`-th element of E in the weighted sum forming $\overline{A}_{ij}$. (input)
- `A` (`SparseMatrixCSC{Float64}`) – Sparse matrix defining the column values (input)

Groups

Problem data - semidefinite

`putbararowlist`

```
function putbararowlist(task::MSKtask,
                        subi::Vector{Int32},
                        ptrb::Vector{Int64},
                        ptre::Vector{Int64},
                        subj::Vector{Int32},
                        nummat::Vector{Int64},
                        matidx::Vector{Int64},
                        weights::Vector{Float64})
function putbararowlist(task::MSKtask,
                        subi::T0,
                        ptrb::T1,
                        ptre::T2,
                        subj::T3,
                        nummat::T4,
                        matidx::T5,
                        weights::T6)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
```

(continues on next page)

(continued from previous page)

```
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Integer},
            T5<:AbstractVector{<:Integer},
            T6<:AbstractVector{<:Number} }
function putbararowlist(task::MSKtask,
                        subi::T0,
                        A:: SparseMatrixCSC{Float64},
                        matidx::T5,
                        weights::T6)
    where { T0<:AbstractVector{<:Integer},
            T5<:AbstractVector{<:Integer},
            T6<:AbstractVector{<:Number} }
```

This function replaces a list of rows in the $\overline{A}$ matrix.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subi` (`Int32[]`) – Row indexes of $\overline{A}$. (input)
- `ptrb` (`Int64[]`) – Start of rows in $\overline{A}$. (input)
- `ptre` (`Int64[]`) – End of rows in $\overline{A}$. (input)
- `subj` (`Int32[]`) – Column index of $\overline{A}$. (input)
- `nummat` (`Int64[]`) – Number of entries in weighted sum of matrixes. (input)
- `matidx` (`Int64[]`) – Matrix indexes for weighted sum of matrixes. (input)
- `weights` (`Float64[]`) – Weights for weighted sum of matrixes. (input)
- `A` (`SparseMatrixCSC{Float64}`) – Sparse matrix defining the column values (input)

Groups

Problem data - semidefinite

`putbarcblocktriplet`

```
function putbarcblocktriplet(task::MSKtask,
                             subj::Vector{Int32},
                             subk::Vector{Int32},
                             subl::Vector{Int32},
                             valjkl::Vector{Float64})
function putbarcblocktriplet(task::MSKtask,
                             subj::T0,
                             subk::T1,
                             subl::T2,
                             valjkl::T3)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number} }
```

Inputs the $\overline{C}$ matrix in block triplet form.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subj` (`Int32[]`) – Symmetric matrix variable index. (input)
- `subk` (`Int32[]`) – Block row index. (input)
- `subl` (`Int32[]`) – Block column index. (input)
- `valjkl` (`Float64[]`) – The numerical value associated with each block triplet. (input)

Groups

Problem data - semidefinite

`putbarcj`

```
function putbarcj(task::MSKtask,
                  j::Int32,
                  sub::Vector{Int64},
                  weights::Vector{Float64})
function putbarcj(task::MSKtask,
                  j::T0,
                  sub::T1,
                  weights::T2)
    where { T0<:Integer,
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
```

This function sets one entry in the $\overline{C}$ vector.

Each element in the $\overline{C}$ vector is a weighted sum of symmetric matrices from the symmetric matrix storage E, so $\overline{C}_j$ is a symmetric matrix. By default all elements in $\overline{C}$ are 0, so only non-zero elements need be added. Setting the same element again will overwrite the earlier entry.

The symmetric matrices from E are defined separately using the function *appendsparsesymmat*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the element in $\overline{C}$ that should be changed. (input)
- `sub` (`Int64[]`) – Indices in E of matrices appearing in the weighted sum for $\overline{C}_j$ (input)
- `weights` (`Float64[]`) – `weights[k]` is the coefficient of the `sub[k]`-th element of E in the weighted sum forming $\overline{C}_j$. (input)

Groups

Problem data - semidefinite, *Problem data - objective*

`putbarsj`

```
function putbarsj(task::MSKtask,
                  whichsol::Soltype,
                  j::Int32,
                  barsj::Vector{Float64})
function putbarsj(task::MSKtask,
                  whichsol::Soltype,
                  j::T0,
                  barsj::T1)
    where { T0<:Integer,
            T1<:AbstractVector{<:Number} }
```

Sets the dual solution for a semidefinite variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)
- `j` (`Int32`) – Index of the semidefinite variable. (input)
- `barsj` (`Float64[]`) – Value of $\overline{S}_j$. Format as in *getbarsj*. (input)

Groups

Solution - semidefinite

`putbarvarname`

```
function putbarvarname(task::MSKtask,
                       j::Int32,
                       name::Union{Nothing,AbstractString})
function putbarvarname(task::MSKtask,
                       j::T0,
                       name::Union{Nothing,AbstractString})
    where { T0<:Integer }
```

Sets the name of a semidefinite variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the variable. (input)
- `name` (`AbstractString`) – The variable name. (input)

Groups

Names, *Problem data - semidefinite*

`putbarxj`

```
function putbarxj(task::MSKtask,
                  whichsol::Soltype,
                  j::Int32,
                  barxj::Vector{Float64})
function putbarxj(task::MSKtask,
                  whichsol::Soltype,
                  j::T0,
                  barxj::T1)
    where { T0<:Integer,
            T1<:AbstractVector{<:Number} }
```

Sets the primal solution for a semidefinite variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `j` (`Int32`) – Index of the semidefinite variable. (input)
- `barxj` (`Float64[]`) – Value of $\overline{X}_j$. Format as in *`getbarxj`*. (input)

Groups

Solution - semidefinite

`putcallbackfunc`

```
function putcallbackfunc(task::MSKtask,
                         f::Function)
```

Receive callbacks with solver status and information during optimization.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `f` (`Function`) – The callback function. (input)

Groups

Callback

`putcfix`

```
function putcfix(task::MSKtask,
                 cfix::Float64)
function putcfix(task::MSKtask,
                 cfix::T0)
    where { T0<:Number }
```

Replaces the fixed term in the objective by a new one.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `cfix` (`Float64`) – Fixed term in the objective. (input)

Groups

Problem data - linear part, *Problem data - objective*

`putcj`

```
function putcj(task::MSKtask,
               j::Int32,
               cj::Float64)
function putcj(task::MSKtask,
               j::T0,
               cj::T1)
    where { T0<:Integer,
            T1<:Number }
```

Modifies one coefficient in the linear objective vector c, i.e.

$$c_{\mathtt{j}} = \mathtt{cj}.$$

If the absolute value exceeds *MSK_DPAR_DATA_TOL_C_HUGE* an error is generated. If the absolute value exceeds *MSK_DPAR_DATA_TOL_CJ_LARGE*, a warning is generated, but the coefficient is inputted as specified.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the variable for which c should be changed. (input)
- `cj` (`Float64`) – New value of c_j. (input)

Groups

Problem data - linear part, *Problem data - objective*

`putclist`

```
function putclist(task::MSKtask,
                  subj::Vector{Int32},
                  val::Vector{Float64})
function putclist(task::MSKtask,
                  subj::T0,
                  val::T1)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Number} }
```

Modifies the coefficients in the linear term c in the objective using the principle

$$c_{\mathtt{subj[t]}} = \mathtt{val[t]}, \quad t = 1, \ldots, \mathtt{num}.$$

If a variable index is specified multiple times in `subj` only the last entry is used. Data checks are performed as in *putcj*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subj` (`Int32[]`) – Indices of variables for which the coefficient in c should be changed. (input)
- `val` (`Float64[]`) – New numerical values for coefficients in c that should be modified. (input)

Groups

Problem data - linear part, *Problem data - variables*, *Problem data - objective*

`putconbound`

```
function putconbound(task::MSKtask,
                     i::Int32,
                     bkc::Boundkey,
                     blc::Float64,
                     buc::Float64)
function putconbound(task::MSKtask,
                     i::T0,
                     bkc::Boundkey,
                     blc::T1,
                     buc::T2)
    where { T0<:Integer,
            T1<:Number,
            T2<:Number }
```

Changes the bounds for one constraint.

If the bound value specified is numerically larger than *MSK_DPAR_DATA_TOL_BOUND_INF* it is considered infinite and the bound key is changed accordingly. If a bound value is numerically larger than *MSK_DPAR_DATA_TOL_BOUND_WRN*, a warning will be displayed, but the bound is inputted as specified.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the constraint. (input)
- `bkc` (*Boundkey*) – New bound key. (input)
- `blc` (`Float64`) – New lower bound. (input)
- `buc` (`Float64`) – New upper bound. (input)

Groups

Problem data - linear part, *Problem data - constraints*, *Problem data - bounds*

`putconboundlist`

```
function putconboundlist(task::MSKtask,
                         sub::Vector{Int32},
                         bkc::Vector{Boundkey},
                         blc::Vector{Float64},
                         buc::Vector{Float64})
function putconboundlist(task::MSKtask,
                         sub::T0,
                         bkc::Vector{Boundkey},
                         blc::T1,
                         buc::T2)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Number},
            T2<:AbstractVector{<:Number} }
```

Changes the bounds for a list of constraints. If multiple bound changes are specified for a constraint, then only the last change takes effect. Data checks are performed as in *putconbound*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `sub` (`Int32[]`) – List of constraint indexes. (input)
- `bkc` (*`Boundkey`*`[]`) – Bound keys for the constraints. (input)
- `blc` (`Float64[]`) – Lower bounds for the constraints. (input)
- `buc` (`Float64[]`) – Upper bounds for the constraints. (input)

Groups

Problem data - linear part, *Problem data - constraints*, *Problem data - bounds*

`putconboundlistconst`

```
function putconboundlistconst(task::MSKtask,
                              sub::Vector{Int32},
                              bkc::Boundkey,
                              blc::Float64,
                              buc::Float64)
function putconboundlistconst(task::MSKtask,
                              sub::T0,
                              bkc::Boundkey,
                              blc::T1,
                              buc::T2)
    where { T0<:AbstractVector{<:Integer},
            T1<:Number,
            T2<:Number }
```

Changes the bounds for one or more constraints. Data checks are performed as in *`putconbound`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `sub` (`Int32[]`) – List of constraint indexes. (input)
- `bkc` (*`Boundkey`*) – New bound key for all constraints in the list. (input)
- `blc` (`Float64`) – New lower bound for all constraints in the list. (input)
- `buc` (`Float64`) – New upper bound for all constraints in the list. (input)

Groups

Problem data - linear part, *Problem data - constraints*, *Problem data - bounds*

`putconboundslice`

```
function putconboundslice(task::MSKtask,
                          first::Int32,
                          last::Int32,
                          bkc::Vector{Boundkey},
                          blc::Vector{Float64},
                          buc::Vector{Float64})
function putconboundslice(task::MSKtask,
                          first::T0,
                          last::T1,
                          bkc::Vector{Boundkey},
                          blc::T2,
                          buc::T3)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number},
            T3<:AbstractVector{<:Number} }
```

Changes the bounds for a slice of the constraints. Data checks are performed as in *`putconbound`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `bkc` (*`Boundkey`*`[]`) – Bound keys for the constraints. (input)
- `blc` (`Float64[]`) – Lower bounds for the constraints. (input)
- `buc` (`Float64[]`) – Upper bounds for the constraints. (input)

Groups

Problem data - linear part, *Problem data - constraints*, *Problem data - bounds*

`putconboundsliceconst`

```
function putconboundsliceconst(task::MSKtask,
                               first::Int32,
                               last::Int32,
                               bkc::Boundkey,
                               blc::Float64,
                               buc::Float64)
function putconboundsliceconst(task::MSKtask,
                               first::T0,
                               last::T1,
                               bkc::Boundkey,
                               blc::T2,
                               buc::T3)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Number,
            T3<:Number }
```

Changes the bounds for a slice of the constraints. Data checks are performed as in *`putconbound`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `bkc` (*`Boundkey`*) – New bound key for all constraints in the slice. (input)
- `blc` (`Float64`) – New lower bound for all constraints in the slice. (input)
- `buc` (`Float64`) – New upper bound for all constraints in the slice. (input)

Groups

Problem data - linear part, *Problem data - constraints*, *Problem data - bounds*

~~`putcone`~~ *Deprecated*

```
function putcone(task::MSKtask,
                 k::Int32,
                 ct::Conetype,
                 conepar::Float64,
                 submem::Vector{Int32})
function putcone(task::MSKtask,
                 k::T0,
                 ct::Conetype,
                 conepar::T1,
                 submem::T2)
    where { T0<:Integer,
            T1<:Number,
            T2<:AbstractVector{<:Integer} }
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `k` (`Int32`) – Index of the cone. (input)
- `ct` (*`Conetype`*) – Specifies the type of the cone. (input)
- `conepar` (`Float64`) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0. (input)
- `submem` (`Int32[]`) – Variable subscripts of the members in the cone. (input)

Groups

Problem data - cones (deprecated)

~~`putconename`~~ *Deprecated*

```
function putconename(task::MSKtask,
                     j::Int32,
                     name::Union{Nothing,AbstractString})
function putconename(task::MSKtask,
                     j::T0,
                     name::Union{Nothing,AbstractString})
    where { T0<:Integer }
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the cone. (input)
- `name` (`AbstractString`) – The name of the cone. (input)

Groups

Names, *Problem data - cones (deprecated)*

`putconname`

```
function putconname(task::MSKtask,
                    i::Int32,
                    name::Union{Nothing,AbstractString})
function putconname(task::MSKtask,
                    i::T0,
                    name::Union{Nothing,AbstractString})
    where { T0<:Integer }
```

Sets the name of a constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the constraint. (input)
- `name` (`AbstractString`) – The name of the constraint. (input)

Groups

Names, *Problem data - constraints*, *Problem data - linear part*

`putconsolutioni`

```
function putconsolutioni(task::MSKtask,
                         i::Int32,
                         whichsol::Soltype,
                         sk::Stakey,
                         x::Float64,
```

(continues on next page)

(continued from previous page)

```
                          sl::Float64,
                          su::Float64)
function putconsolutioni(task::MSKtask,
                          i::T0,
                          whichsol::Soltype,
                          sk::Stakey,
                          x::T1,
                          sl::T2,
                          su::T3)
    where { T0<:Integer,
            T1<:Number,
            T2<:Number,
            T3<:Number }
```

Sets the primal and dual solution information for a single constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the constraint. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `sk` (*`Stakey`*) – Status key of the constraint. (input)
- `x` (`Float64`) – Primal solution value of the constraint. (input)
- `sl` (`Float64`) – Solution value of the dual variable associated with the lower bound. (input)
- `su` (`Float64`) – Solution value of the dual variable associated with the upper bound. (input)

Groups

Solution information, *Solution - primal*, *Solution - dual*

`putcslice`

```
function putcslice(task::MSKtask,
                   first::Int32,
                   last::Int32,
                   slice::Vector{Float64})
function putcslice(task::MSKtask,
                   first::T0,
                   last::T1,
                   slice::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Modifies a slice in the linear term c in the objective using the principle

$$c_{\mathtt{j}} = \mathtt{slice}[\mathtt{j} - \mathtt{first} + 1], \quad j = \mathtt{first}, .., \mathtt{last} - 1$$

Data checks are performed as in *`putcj`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – First element in the slice of c. (input)
- `last` (`Int32`) – Last element plus 1 of the slice in c to be changed. (input)
- `slice` (`Float64[]`) – New numerical values for coefficients in c that should be modified. (input)

Groups

Problem data - linear part, *Problem data - objective*

putdjc

```
function putdjc(task::MSKtask,
                djcidx::Int64,
                domidxlist::Vector{Int64},
                afeidxlist::Vector{Int64},
                b::Union{Nothing,Vector{Float64}},
                termsizelist::Vector{Int64})
function putdjc(task::MSKtask,
                djcidx::T0,
                domidxlist::T1,
                afeidxlist::T2,
                b::T3,
                termsizelist::T4)
    where { T0<:Integer,
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number},
            T4<:AbstractVector{<:Integer} }
```

Inputs a disjunctive constraint. The constraint has the form

$$T_1 \text{ or } T_2 \text{ or } \cdots \text{ or } T_{\text{numterms}}$$

For each $i = 1, \ldots$ numterms the i-th clause (term) T_i has the form *a sequence of affine expressions belongs to a product of domains*, where the number of domains is termsizelist$[i]$ and the number of affine expressions is equal to the sum of dimensions of all domains appearing in T_i.

All the domains and all the affine expressions appearing in the above description are arranged sequentially in the lists `domidxlist` and `afeidxlist`, respectively. In particular, the length of `domidxlist` must be equal to the sum of elements of `termsizelist`, and the length of `afeidxlist` must be equal to the sum of dimensions of all the domains appearing in `domidxlist`.

The elements of `domidxlist` are indexes of domains previously defined with one of the `append..domain` functions.

The elements of `afeidxlist` are indexes to the store of affine expressions, i.e. the k-th affine expression appearing in the disjunctive constraint is going to be

$$F_{\text{afeidxlist}[k],:}x + g_{\text{afeidxlist}[k]}$$

If an optional vector `b` of the same length as `afeidxlist` is specified then the k-th affine expression appearing in the disjunctive constraint will be taken as

$$F_{\text{afeidxlist}[k],:}x + g_{\text{afeidxlist}[k]} - b_k$$

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of the disjunctive constraint. (input)
- `domidxlist` (`Int64[]`) – List of domain indexes. (input)
- `afeidxlist` (`Int64[]`) – List of affine expression indexes. (input)
- `b` (`Float64[]`) – The vector of constant terms modifying affine expressions. (input)
- `termsizelist` (`Int64[]`) – List of term sizes. (input)

Groups

Problem data - disjunctive constraints

putdjcname

```
function putdjcname(task::MSKtask,
                    djcidx::Int64,
                    name::Union{Nothing,AbstractString})
function putdjcname(task::MSKtask,
                    djcidx::T0,
                    name::Union{Nothing,AbstractString})
    where { T0<:Integer }
```

Sets the name of a disjunctive constraint.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `djcidx` (`Int64`) – Index of the disjunctive constraint. (input)
- `name` (`AbstractString`) – The name of the disjunctive constraint. (input)

Groups

Names, *Problem data - disjunctive constraints*

`putdjcslice`

```
function putdjcslice(task::MSKtask,
                     idxfirst::Int64,
                     idxlast::Int64,
                     domidxlist::Vector{Int64},
                     afeidxlist::Vector{Int64},
                     b::Union{Nothing,Vector{Float64}},
                     termsizelist::Vector{Int64},
                     termsindjc::Vector{Int64})
function putdjcslice(task::MSKtask,
                     idxfirst::T0,
                     idxlast::T1,
                     domidxlist::T2,
                     afeidxlist::T3,
                     b::T4,
                     termsizelist::T5,
                     termsindjc::T6)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Integer},
            T4<:AbstractVector{<:Number},
            T5<:AbstractVector{<:Integer},
            T6<:AbstractVector{<:Integer} }
```

Inputs a slice of disjunctive constraints.

The array `termsindjc` should have length idxlast − idxfirst and contain the number of terms in consecutive constraints forming the slice.

The rest of the input consists of concatenated descriptions of individual constraints, where each constraint is described as in *putdjc*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `idxfirst` (`Int64`) – Index of the first disjunctive constraint in the slice. (input)
- `idxlast` (`Int64`) – Index of the last disjunctive constraint in the slice plus 1. (input)
- `domidxlist` (`Int64[]`) – List of domain indexes. (input)
- `afeidxlist` (`Int64[]`) – List of affine expression indexes. (input)

- `b` (`Float64[]`) – The vector of constant terms modifying affine expressions. Optional, can be `nothing` if not required. (input)
- `termsizelist` (`Int64[]`) – List of term sizes. (input)
- `termsindjc` (`Int64[]`) – Number of terms in each of the disjunctive constraints in the slice. (input)

Groups

Problem data - disjunctive constraints

`putdomainname`

```
function putdomainname(task::MSKtask,
                       domidx::Int64,
                       name::Union{Nothing,AbstractString})
function putdomainname(task::MSKtask,
                       domidx::T0,
                       name::Union{Nothing,AbstractString})
    where { T0<:Integer }
```

Sets the name of a domain.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `domidx` (`Int64`) – Index of the domain. (input)
- `name` (`AbstractString`) – The name of the domain. (input)

Groups

Names, *Problem data - domain*

`putdouparam`

```
function putdouparam(task::MSKtask,
                     param::Dparam,
                     parvalue::Float64)
function putdouparam(task::MSKtask,
                     param::Dparam,
                     parvalue::T0)
    where { T0<:Number }
```

Sets the value of a double parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*`Dparam`*) – Which parameter. (input)
- `parvalue` (`Float64`) – Parameter value. (input)

Groups

Parameters

`putintparam`

```
function putintparam(task::MSKtask,
                     param::Iparam,
                     parvalue::Int32)
function putintparam(task::MSKtask,
                     param::Iparam,
                     parvalue::T0)
    where { T0<:Integer }
```

Sets the value of an integer parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*`Iparam`*) – Which parameter. (input)
- `parvalue` (`Int32`) – Parameter value. (input)

Groups

Parameters

`putlicensecode`

```
function putlicensecode(env::MSKenv,
                        code::Union{Nothing,Vector{Int32}})
function putlicensecode(env::MSKenv,
                        code::T0)
    where { T0<:AbstractVector{<:Integer} }
function putlicensecode(code::Union{Nothing,Vector{Int32}})
function putlicensecode(code::T0)
    where { T0<:AbstractVector{<:Integer} }
```

Input a runtime license code. This function has an effect only before the first optimization.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)
- `code` (`Int32[]`) – A runtime license code. (input)

Groups

License system

`putlicensedebug`

```
function putlicensedebug(env::MSKenv,
                         licdebug::Int32)
function putlicensedebug(env::MSKenv,
                         licdebug::T0)
    where { T0<:Integer }
function putlicensedebug(licdebug::Int32)
function putlicensedebug(licdebug::T0)
    where { T0<:Integer }
```

Enables debug information for the license system. If `licdebug` is non-zero, then **MOSEK** will print debug info regarding the license checkout.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)
- `licdebug` (`Int32`) – Whether license checkout debug info should be printed. (input)

Groups

License system

`putlicensepath`

```
function putlicensepath(env::MSKenv,
                        licensepath::Union{Nothing,AbstractString})
function putlicensepath(licensepath::Union{Nothing,AbstractString})
```

Set the path to the license file. This function has an effect only before the first optimization.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)
- `licensepath` (`AbstractString`) – A path specifying where to search for the license. (input)

Groups

License system

`putlicensewait`

```
function putlicensewait(env::MSKenv,
                        licwait::Int32)
function putlicensewait(env::MSKenv,
                        licwait::T0)
    where { T0<:Integer }
function putlicensewait(licwait::Int32)
function putlicensewait(licwait::T0)
    where { T0<:Integer }
```

Control whether **MOSEK** should wait for an available license if no license is available. If `licwait` is non-zero, then **MOSEK** will wait for `licwait-1` milliseconds between each check for an available license.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)
- `licwait` (`Int32`) – Whether **MOSEK** should wait for a license if no license is available. (input)

Groups

License system

`putlintparam`

```
function putlintparam(task::MSKtask,
                      param::Iparam,
                      parvalue::Int64)
function putlintparam(task::MSKtask,
                      param::Iparam,
                      parvalue::T0)
    where { T0<:Integer }
```

Sets the value of an integer parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*Iparam*) – Which parameter. (input)
- `parvalue` (`Int64`) – Parameter value. (input)

Groups

Parameters

`putmaxnumacc`

```
function putmaxnumacc(task::MSKtask,
                      maxnumacc::Int64)
function putmaxnumacc(task::MSKtask,
                      maxnumacc::T0)
    where { T0<:Integer }
```

Sets the number of preallocated affine conic constraints in the optimization task. When this number is reached **MOSEK** will automatically allocate more space. It is never mandatory to call this function, since **MOSEK** will reallocate any internal structures whenever it is required.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumacc` (`Int64`) – Number of preallocated affine conic constraints. (input)

Groups

Environment and task management, *Problem data - affine conic constraints*

`putmaxnumafe`

```
function putmaxnumafe(task::MSKtask,
                      maxnumafe::Int64)
function putmaxnumafe(task::MSKtask,
                      maxnumafe::T0)
    where { T0<:Integer }
```

Sets the number of preallocated affine expressions in the optimization task. When this number is reached **MOSEK** will automatically allocate more space for affine expressions. It is never mandatory to call this function, since **MOSEK** will reallocate any internal structures whenever it is required.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumafe` (`Int64`) – Number of preallocated affine expressions. (input)

Groups

Environment and task management, *Problem data - affine expressions*

`putmaxnumanz`

```
function putmaxnumanz(task::MSKtask,
                      maxnumanz::Int64)
function putmaxnumanz(task::MSKtask,
                      maxnumanz::T0)
    where { T0<:Integer }
```

Sets the number of preallocated non-zero entries in A.

MOSEK stores only the non-zero elements in the linear coefficient matrix A and it cannot predict how much storage is required to store A. Using this function it is possible to specify the number of non-zeros to preallocate for storing A.

If the number of non-zeros in the problem is known, it is a good idea to set `maxnumanz` slightly larger than this number, otherwise a rough estimate can be used. In general, if A is inputted in many small chunks, setting this value may speed up the data input phase.

It is not mandatory to call this function, since **MOSEK** will reallocate internal structures whenever it is necessary.

The function call has no effect if both `maxnumcon` and `maxnumvar` are zero.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumanz` (`Int64`) – Number of preallocated non-zeros in A. (input)

Groups

Environment and task management, *Problem data - linear part*

`putmaxnumbarvar`

```
function putmaxnumbarvar(task::MSKtask,
                         maxnumbarvar::Int32)
function putmaxnumbarvar(task::MSKtask,
                         maxnumbarvar::T0)
    where { T0<:Integer }
```

Sets the number of preallocated symmetric matrix variables in the optimization task. When this number of variables is reached **MOSEK** will automatically allocate more space for variables.

It is not mandatory to call this function. It only gives a hint about the amount of data to preallocate for efficiency reasons.

Please note that `maxnumbarvar` must be larger than the current number of symmetric matrix variables in the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumbarvar` (`Int32`) – Number of preallocated symmetric matrix variables. (input)

Groups

Environment and task management, *Problem data - semidefinite*

`putmaxnumcon`

```
function putmaxnumcon(task::MSKtask,
                      maxnumcon::Int32)
function putmaxnumcon(task::MSKtask,
                      maxnumcon::T0)
    where { T0<:Integer }
```

Sets the number of preallocated constraints in the optimization task. When this number of constraints is reached **MOSEK** will automatically allocate more space for constraints.

It is never mandatory to call this function, since **MOSEK** will reallocate any internal structures whenever it is required.

Please note that `maxnumcon` must be larger than the current number of constraints in the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumcon` (`Int32`) – Number of preallocated constraints in the optimization task. (input)

Groups

Environment and task management, *Problem data - constraints*

~~`putmaxnumcone`~~ *Deprecated*

```
function putmaxnumcone(task::MSKtask,
                       maxnumcone::Int32)
function putmaxnumcone(task::MSKtask,
                       maxnumcone::T0)
    where { T0<:Integer }
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Sets the number of preallocated conic constraints in the optimization task. When this number of conic constraints is reached **MOSEK** will automatically allocate more space for conic constraints.

It is not mandatory to call this function, since **MOSEK** will reallocate any internal structures whenever it is required.

Please note that `maxnumcon` must be larger than the current number of conic constraints in the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumcone` (`Int32`) – Number of preallocated conic constraints in the optimization task. (input)

Groups

Environment and task management, *Problem data - cones (deprecated)*

`putmaxnumdjc`

```
function putmaxnumdjc(task::MSKtask,
                      maxnumdjc::Int64)
function putmaxnumdjc(task::MSKtask,
                      maxnumdjc::T0)
    where { T0<:Integer }
```

Sets the number of preallocated disjunctive constraints in the optimization task. When this number is reached **MOSEK** will automatically allocate more space. It is never mandatory to call this function, since **MOSEK** will reallocate any internal structures whenever it is required.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumdjc` (`Int64`) – Number of preallocated disjunctive constraints in the task. (input)

Groups

Environment and task management, *Problem data - disjunctive constraints*

`putmaxnumdomain`

```
function putmaxnumdomain(task::MSKtask,
                         maxnumdomain::Int64)
function putmaxnumdomain(task::MSKtask,
                         maxnumdomain::T0)
    where { T0<:Integer }
```

Sets the number of preallocated domains in the optimization task. When this number is reached **MOSEK** will automatically allocate more space. It is never mandatory to call this function, since **MOSEK** will reallocate any internal structures whenever it is required.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumdomain` (`Int64`) – Number of preallocated domains. (input)

Groups

Environment and task management, *Problem data - domain*

`putmaxnumqnz`

```
function putmaxnumqnz(task::MSKtask,
                      maxnumqnz::Int64)
function putmaxnumqnz(task::MSKtask,
                      maxnumqnz::T0)
    where { T0<:Integer }
```

Sets the number of preallocated non-zero entries in quadratic terms.

MOSEK stores only the non-zero elements in Q. Therefore, **MOSEK** cannot predict how much storage is required to store Q. Using this function it is possible to specify the number non-zeros to preallocate for storing Q (both objective and constraints).

It may be advantageous to reserve more non-zeros for Q than actually needed since it may improve the internal efficiency of **MOSEK**, however, it is never worthwhile to specify more than the double of the anticipated number of non-zeros in Q.

It is not mandatory to call this function, since **MOSEK** will reallocate internal structures whenever it is necessary.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumqnz` (`Int64`) – Number of non-zero elements preallocated in quadratic coefficient matrices. (input)

Groups

Environment and task management, *Problem data - quadratic part*

`putmaxnumvar`

```
function putmaxnumvar(task::MSKtask,
                      maxnumvar::Int32)
function putmaxnumvar(task::MSKtask,
                      maxnumvar::T0)
    where { T0<:Integer }
```

Sets the number of preallocated variables in the optimization task. When this number of variables is reached **MOSEK** will automatically allocate more space for variables.

It is not mandatory to call this function. It only gives a hint about the amount of data to preallocate for efficiency reasons.

Please note that `maxnumvar` must be larger than the current number of variables in the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumvar` (`Int32`) – Number of preallocated variables in the optimization task. (input)

Groups

Environment and task management, *Problem data - variables*

`putnadouparam`

```
function putnadouparam(task::MSKtask,
                       paramname::AbstractString,
                       parvalue::Float64)
function putnadouparam(task::MSKtask,
                       paramname::Union{Nothing,AbstractString},
                       parvalue::T0)
    where { T0<:Number }
```

Sets the value of a named double parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `paramname` (`AbstractString`) – Name of a parameter. (input)
- `parvalue` (`Float64`) – Parameter value. (input)

Groups

Parameters

`putnaintparam`

```
function putnaintparam(task::MSKtask,
                       paramname::AbstractString,
                       parvalue::Int32)
function putnaintparam(task::MSKtask,
                       paramname::Union{Nothing,AbstractString},
                       parvalue::T0)
    where { T0<:Integer }
```

Sets the value of a named integer parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `paramname` (`AbstractString`) – Name of a parameter. (input)

- `parvalue` (`Int32`) – Parameter value. (input)

Groups
Parameters

`putnastrparam`

```
function putnastrparam(task::MSKtask,
                       paramname::AbstractString,
                       parvalue::AbstractString)
```

Sets the value of a named string parameter.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `paramname` (`AbstractString`) – Name of a parameter. (input)
- `parvalue` (`AbstractString`) – Parameter value. (input)

Groups
Parameters

`putobjname`

```
function putobjname(task::MSKtask,
                    objname::AbstractString)
```

Assigns a new name to the objective.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `objname` (`AbstractString`) – Name of the objective. (input)

Groups
Problem data - linear part, *Names*, *Problem data - objective*

`putobjsense`

```
function putobjsense(task::MSKtask,
                     sense::Objsense)
```

Sets the objective sense of the task.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `sense` (*`Objsense`*) – The objective sense of the task. The values *`MSK_OBJECTIVE_SENSE_MAXIMIZE`* and *`MSK_OBJECTIVE_SENSE_MINIMIZE`* mean that the problem is maximized or minimized respectively. (input)

Groups
Problem data - linear part, *Problem data - objective*

`putoptserverhost`

```
function putoptserverhost(task::MSKtask,
                          host::Union{Nothing,AbstractString})
```

Specify an OptServer URL for remote calls. The URL should contain protocol, host and port in the form `http://server:port` or `https://server:port`. If the URL is set using this function, all subsequent calls to any **MOSEK** function that involves synchronous optimization will be sent to the specified OptServer instead of being executed locally. Passing `nothing` or empty string deactivates this redirection.

Has the same effect as setting the parameter *`MSK_SPAR_REMOTE_OPTSERVER_HOST`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `host` (`AbstractString`) – A URL specifying the optimization server to be used. (input)

Groups

Remote optimization

`putparam`

```
function putparam(task::MSKtask,
                  parname::AbstractString,
                  parvalue::AbstractString)
```

Checks if `parname` is valid parameter name. If it is, the parameter is assigned the value specified by `parvalue`.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `parname` (`AbstractString`) – Parameter name. (input)
- `parvalue` (`AbstractString`) – Parameter value. (input)

Groups

Parameters

`putqcon`

```
function putqcon(task::MSKtask,
                 qcsubk::Vector{Int32},
                 qcsubi::Vector{Int32},
                 qcsubj::Vector{Int32},
                 qcval::Vector{Float64})
function putqcon(task::MSKtask,
                 qcsubk::T0,
                 qcsubi::T1,
                 qcsubj::T2,
                 qcval::T3)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number} }
```

Replace all quadratic entries in the constraints. The list of constraints has the form

$$l_k^c \leq \frac{1}{2} \sum_{i=1}^{\texttt{numvar}} \sum_{j=1}^{\texttt{numvar}} q_{ij}^k x_i x_j + \sum_{j=1}^{\texttt{numvar}} a_{kj} x_j \leq u_k^c, \quad k = 1, \ldots, m.$$

This function sets all the quadratic terms to zero and then performs the update:

$$q^{\texttt{qcsubk[t]}}_{\texttt{qcsubi[t]},\texttt{qcsubj[t]}} = q^{\texttt{qcsubk[t]}}_{\texttt{qcsubj[t]},\texttt{qcsubi[t]}} = q^{\texttt{qcsubk[t]}}_{\texttt{qcsubj[t]},\texttt{qcsubi[t]}} + \texttt{qcval[t]},$$

for $t = 1, \ldots,$ `numqcnz`.

Please note that:

- For large problems it is essential for the efficiency that the function *putmaxnumqnz* is employed to pre-allocate space.
- Only the lower triangular parts should be specified because the Q matrices are symmetric. Specifying entries where $i < j$ will result in an error.
- Only non-zero elements should be specified.
- The order in which the non-zero elements are specified is insignificant.

- Duplicate elements are added together as shown above. Hence, it is usually not recommended to specify the same entry multiple times.

For a code example see Section *Quadratic Optimization*

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `qcsubk` (`Int32[]`) – Constraint subscripts for quadratic coefficients. (input)
- `qcsubi` (`Int32[]`) – Row subscripts for quadratic constraint matrix. (input)
- `qcsubj` (`Int32[]`) – Column subscripts for quadratic constraint matrix. (input)
- `qcval` (`Float64[]`) – Quadratic constraint coefficient values. (input)

Groups

Problem data - quadratic part

`putqconk`

```
function putqconk(task::MSKtask,
                  k::Int32,
                  qcsubi::Vector{Int32},
                  qcsubj::Vector{Int32},
                  qcval::Vector{Float64})
function putqconk(task::MSKtask,
                  k::T0,
                  qcsubi::T1,
                  qcsubj::T2,
                  qcval::T3)
    where { T0<:Integer,
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number} }
function putqconk(task::MSKtask,
                  k::T0,
                  Qk:: SparseMatrixCSC{Float64})
    where { T0<:Integer }
```

Replaces all the quadratic entries in one constraint. This function performs the same operations as *putqcon* but only with respect to constraint number `k` and it does not modify the other constraints. See the description of *putqcon* for definitions and important remarks.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `k` (`Int32`) – The constraint in which the new Q elements are inserted. (input)
- `qcsubi` (`Int32[]`) – Row subscripts for quadratic constraint matrix. (input)
- `qcsubj` (`Int32[]`) – Column subscripts for quadratic constraint matrix. (input)
- `qcval` (`Float64[]`) – Quadratic constraint coefficient values. (input)
- `Qk` (`SparseMatrixCSC{Float64}`) – Sparse matrix defining the column values (input)

Groups

Problem data - quadratic part

`putqobj`

```
function putqobj(task::MSKtask,
                 qosubi::Vector{Int32},
                 qosubj::Vector{Int32},
                 qoval::Vector{Float64})
function putqobj(task::MSKtask,
```

(continues on next page)

(continued from previous page)

```
                  qosubi::T0,
                  qosubj::T1,
                  qoval::T2)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Number} }
function putqobj(task::MSKtask,
                  Qk:: SparseMatrixCSC{Float64})
```

Replace all quadratic terms in the objective. If the objective has the form

$$\frac{1}{2}\sum_{i=1}^{\texttt{numvar}}\sum_{j=1}^{\texttt{numvar}} q^o_{ij}x_ix_j + \sum_{j=1}^{\texttt{numvar}} c_jx_j + c^f$$

then this function sets all the quadratic terms to zero and then performs the update:

$$q^o_{\texttt{qosubi[t]},\texttt{qosubj[t]}} = q^o_{\texttt{qosubj[t]},\texttt{qosubi[t]}} = q^o_{\texttt{qosubj[t]},\texttt{qosubi[t]}} + \texttt{qoval[t]},$$

for $t = 1, \ldots, \texttt{numqonz}$.

See the description of *putqcon* for important remarks and example.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `qosubi` (`Int32[]`) – Row subscripts for quadratic objective coefficients. (input)
- `qosubj` (`Int32[]`) – Column subscripts for quadratic objective coefficients. (input)
- `qoval` (`Float64[]`) – Quadratic objective coefficient values. (input)
- `Qk` (`SparseMatrixCSC{Float64}`) – Sparse matrix defining the column values (input)

Groups

Problem data - quadratic part, *Problem data - objective*

`putqobjij`

```
function putqobjij(task::MSKtask,
                   i::Int32,
                   j::Int32,
                   qoij::Float64)
function putqobjij(task::MSKtask,
                   i::T0,
                   j::T1,
                   qoij::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Number }
```

Replaces one coefficient in the quadratic term in the objective. The function performs the assignment

$$q^o_{ij} = q^o_{ji} = \texttt{qoij}.$$

Only the elements in the lower triangular part are accepted. Setting q_{ij} with $j > i$ will cause an error.

Please note that replacing all quadratic elements one by one is more computationally expensive than replacing them all at once. Use *putqobj* instead whenever possible.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Row index for the coefficient to be replaced. (input)
- `j` (`Int32`) – Column index for the coefficient to be replaced. (input)
- `qoij` (`Float64`) – The new value for q_{ij}^o. (input)

Groups

Problem data - quadratic part, *Problem data - objective*

`putskc`

```
function putskc(task::MSKtask,
                whichsol::Soltype,
                skc::Vector{Stakey})
```

Sets the status keys for the constraints.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `skc` (*`Stakey`*`[]`) – Status keys for the constraints. (input)

Groups

Solution information

`putskcslice`

```
function putskcslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32,
                     skc::Vector{Stakey})
function putskcslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1,
                     skc::Vector{Stakey})
    where { T0<:Integer,
            T1<:Integer }
```

Sets the status keys for a slice of the constraints.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `skc` (*`Stakey`*`[]`) – Status keys for the constraints. (input)

Groups

Solution information

`putskx`

```
function putskx(task::MSKtask,
                whichsol::Soltype,
                skx::Vector{Stakey})
```

Sets the status keys for the scalar variables.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)

- `whichsol` (*Soltype*) – Selects a solution. (input)
- `skx` (*Stakey* `[]`) – Status keys for the variables. (input)

Groups

Solution information

`putskxslice`

```
function putskxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32,
                     skx::Vector{Stakey})
function putskxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1,
                     skx::Vector{Stakey})
    where { T0<:Integer,
            T1<:Integer }
```

Sets the status keys for a slice of the variables.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `skx` (*Stakey* `[]`) – Status keys for the variables. (input)

Groups

Solution information

`putslc`

```
function putslc(task::MSKtask,
                whichsol::Soltype,
                slc::Vector{Float64})
function putslc(task::MSKtask,
                whichsol::Soltype,
                slc::T0)
    where { T0<:AbstractVector{<:Number} }
```

Sets the s_l^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)
- `slc` (`Float64[]`) – Dual variables corresponding to the lower bounds on the constraints. (input)

Groups

Solution - dual

`putslcslice`

```
function putslcslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32,
```

(continues on next page)

(continued from previous page)

```
                    slc::Vector{Float64})
function putslcslice(task::MSKtask,
                    whichsol::Soltype,
                    first::T0,
                    last::T1,
                    slc::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Sets a slice of the s_l^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `slc` (`Float64[]`) – Dual variables corresponding to the lower bounds on the constraints. (input)

Groups

Solution - dual

`putslx`

```
function putslx(task::MSKtask,
                whichsol::Soltype,
                slx::Vector{Float64})
function putslx(task::MSKtask,
                whichsol::Soltype,
                slx::T0)
    where { T0<:AbstractVector{<:Number} }
```

Sets the s_l^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `slx` (`Float64[]`) – Dual variables corresponding to the lower bounds on the variables. (input)

Groups

Solution - dual

`putslxslice`

```
function putslxslice(task::MSKtask,
                    whichsol::Soltype,
                    first::Int32,
                    last::Int32,
                    slx::Vector{Float64})
function putslxslice(task::MSKtask,
                    whichsol::Soltype,
                    first::T0,
                    last::T1,
                    slx::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Sets a slice of the s_l^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `slx` (`Float64[]`) – Dual variables corresponding to the lower bounds on the variables. (input)

Groups

Solution - dual

`putsnx`

```
function putsnx(task::MSKtask,
                whichsol::Soltype,
                sux::Vector{Float64})
function putsnx(task::MSKtask,
                whichsol::Soltype,
                sux::T0)
    where { T0<:AbstractVector{<:Number} }
```

Sets the s_n^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `sux` (`Float64[]`) – Dual variables corresponding to the upper bounds on the variables. (input)

Groups

Solution - dual

`putsnxslice`

```
function putsnxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32,
                     snx::Vector{Float64})
function putsnxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1,
                     snx::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Sets a slice of the s_n^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `snx` (`Float64[]`) – Dual variables corresponding to the conic constraints on the variables. (input)

Groups

Solution - dual

putsolution

```
function putsolution(task::MSKtask,
                     whichsol::Soltype,
                     skc::Vector{Stakey},
                     skx::Vector{Stakey},
                     skn::Vector{Stakey},
                     xc::Union{Nothing,Vector{Float64}},
                     xx::Union{Nothing,Vector{Float64}},
                     y::Union{Nothing,Vector{Float64}},
                     slc::Union{Nothing,Vector{Float64}},
                     suc::Union{Nothing,Vector{Float64}},
                     slx::Union{Nothing,Vector{Float64}},
                     sux::Union{Nothing,Vector{Float64}},
                     snx::Union{Nothing,Vector{Float64}})
function putsolution(task::MSKtask,
                     whichsol::Soltype,
                     skc::Vector{Stakey},
                     skx::Vector{Stakey},
                     skn::Vector{Stakey},
                     xc::T0,
                     xx::T1,
                     y::T2,
                     slc::T3,
                     suc::T4,
                     slx::T5,
                     sux::T6,
                     snx::T7)
    where { T0<:AbstractVector{<:Number},
            T1<:AbstractVector{<:Number},
            T2<:AbstractVector{<:Number},
            T3<:AbstractVector{<:Number},
            T4<:AbstractVector{<:Number},
            T5<:AbstractVector{<:Number},
            T6<:AbstractVector{<:Number},
            T7<:AbstractVector{<:Number} }
```

Inserts a solution into the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `skc` (*`Stakey`*`[]`) – Status keys for the constraints. (input)
- `skx` (*`Stakey`*`[]`) – Status keys for the variables. (input)
- `skn` (*`Stakey`*`[]`) – Status keys for the conic constraints. (input)
- `xc` (`Float64[]`) – Primal constraint solution. (input)
- `xx` (`Float64[]`) – Primal variable solution. (input)
- `y` (`Float64[]`) – Vector of dual variables corresponding to the constraints. (input)
- `slc` (`Float64[]`) – Dual variables corresponding to the lower bounds on the constraints. (input)
- `suc` (`Float64[]`) – Dual variables corresponding to the upper bounds on the constraints. (input)
- `slx` (`Float64[]`) – Dual variables corresponding to the lower bounds on the variables. (input)
- `sux` (`Float64[]`) – Dual variables corresponding to the upper bounds on the variables. (input)

- `snx` (`Float64[]`) – Dual variables corresponding to the conic constraints on the variables. (input)

Groups

Solution information, *Solution - primal*, *Solution - dual*

`putsolutionnew`

```
function putsolutionnew(task::MSKtask,
                        whichsol::Soltype,
                        skc::Vector{Stakey},
                        skx::Vector{Stakey},
                        skn::Vector{Stakey},
                        xc::Union{Nothing,Vector{Float64}},
                        xx::Union{Nothing,Vector{Float64}},
                        y::Union{Nothing,Vector{Float64}},
                        slc::Union{Nothing,Vector{Float64}},
                        suc::Union{Nothing,Vector{Float64}},
                        slx::Union{Nothing,Vector{Float64}},
                        sux::Union{Nothing,Vector{Float64}},
                        snx::Union{Nothing,Vector{Float64}},
                        doty::Union{Nothing,Vector{Float64}})
function putsolutionnew(task::MSKtask,
                        whichsol::Soltype,
                        skc::Vector{Stakey},
                        skx::Vector{Stakey},
                        skn::Vector{Stakey},
                        xc::T0,
                        xx::T1,
                        y::T2,
                        slc::T3,
                        suc::T4,
                        slx::T5,
                        sux::T6,
                        snx::T7,
                        doty::T8)
    where { T0<:AbstractVector{<:Number},
            T1<:AbstractVector{<:Number},
            T2<:AbstractVector{<:Number},
            T3<:AbstractVector{<:Number},
            T4<:AbstractVector{<:Number},
            T5<:AbstractVector{<:Number},
            T6<:AbstractVector{<:Number},
            T7<:AbstractVector{<:Number},
            T8<:AbstractVector{<:Number} }
```

Inserts a solution into the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `skc` (*`Stakey`*`[]`) – Status keys for the constraints. (input)
- `skx` (*`Stakey`*`[]`) – Status keys for the variables. (input)
- `skn` (*`Stakey`*`[]`) – Status keys for the conic constraints. (input)
- `xc` (`Float64[]`) – Primal constraint solution. (input)
- `xx` (`Float64[]`) – Primal variable solution. (input)
- `y` (`Float64[]`) – Vector of dual variables corresponding to the constraints. (input)

- `slc` (`Float64[]`) – Dual variables corresponding to the lower bounds on the constraints. (input)
- `suc` (`Float64[]`) – Dual variables corresponding to the upper bounds on the constraints. (input)
- `slx` (`Float64[]`) – Dual variables corresponding to the lower bounds on the variables. (input)
- `sux` (`Float64[]`) – Dual variables corresponding to the upper bounds on the variables. (input)
- `snx` (`Float64[]`) – Dual variables corresponding to the conic constraints on the variables. (input)
- `doty` (`Float64[]`) – Dual variables corresponding to affine conic constraints. (input)

Groups

Solution information, *Solution - primal*, *Solution - dual*

`putsolutionyi`

```
function putsolutionyi(task::MSKtask,
                       i::Int32,
                       whichsol::Soltype,
                       y::Float64)
function putsolutionyi(task::MSKtask,
                       i::T0,
                       whichsol::Soltype,
                       y::T1)
    where { T0<:Integer,
            T1<:Number }
```

Inputs the dual variable of a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `i` (`Int32`) – Index of the dual variable. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `y` (`Float64`) – Solution value of the dual variable. (input)

Groups

Solution information, *Solution - dual*

`putstreamfunc`

```
function putstreamfunc(task::MSKtask,
                       whichstream::Streamtype,
                       f::Function)
```

Directs all output from a task stream to a stream callback function. The function should accept a string.

Can for example be called as:

```
putstreamfunc(task, MSK_STREAM_LOG, msg -> print(msg))
```

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*`streamtype`*) – Index of the stream. (input)
- `f` (`Function`) – The stream handler function. (input)

Groups

Logging, *Callback*

`putstrparam`

```
function putstrparam(task::MSKtask,
                     param::Sparam,
                     parvalue::AbstractString)
```

Sets the value of a string parameter.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*`Sparam`*) – Which parameter. (input)
- `parvalue` (`AbstractString`) – Parameter value. (input)

Groups

Parameters

`putsuc`

```
function putsuc(task::MSKtask,
                whichsol::Soltype,
                suc::Vector{Float64})
function putsuc(task::MSKtask,
                whichsol::Soltype,
                suc::T0)
    where { T0<:AbstractVector{<:Number} }
```

Sets the s_u^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `suc` (`Float64[]`) – Dual variables corresponding to the upper bounds on the constraints. (input)

Groups

Solution - dual

`putsucslice`

```
function putsucslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32,
                     suc::Vector{Float64})
function putsucslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1,
                     suc::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Sets a slice of the s_u^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)

- `suc` (`Float64[]`) – Dual variables corresponding to the upper bounds on the constraints. (input)

Groups

Solution - dual

`putsux`

```
function putsux(task::MSKtask,
                whichsol::Soltype,
                sux::Vector{Float64})
function putsux(task::MSKtask,
                whichsol::Soltype,
                sux::T0)
    where { T0<:AbstractVector{<:Number} }
```

Sets the s_u^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `sux` (`Float64[]`) – Dual variables corresponding to the upper bounds on the variables. (input)

Groups

Solution - dual

`putsuxslice`

```
function putsuxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::Int32,
                     last::Int32,
                     sux::Vector{Float64})
function putsuxslice(task::MSKtask,
                     whichsol::Soltype,
                     first::T0,
                     last::T1,
                     sux::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Sets a slice of the s_u^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `sux` (`Float64[]`) – Dual variables corresponding to the upper bounds on the variables. (input)

Groups

Solution - dual

`puttaskname`

```
function puttaskname(task::MSKtask,
                     taskname::AbstractString)
```

Assigns a new name to the task.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `taskname` (`AbstractString`) – Name assigned to the task. (input)

Groups

Names, *Environment and task management*

`putvarbound`

```
function putvarbound(task::MSKtask,
                     j::Int32,
                     bkx::Boundkey,
                     blx::Float64,
                     bux::Float64)
function putvarbound(task::MSKtask,
                     j::T0,
                     bkx::Boundkey,
                     blx::T1,
                     bux::T2)
    where { T0<:Integer,
            T1<:Number,
            T2<:Number }
```

Changes the bounds for one variable.

If the bound value specified is numerically larger than *MSK_DPAR_DATA_TOL_BOUND_INF* it is considered infinite and the bound key is changed accordingly. If a bound value is numerically larger than *MSK_DPAR_DATA_TOL_BOUND_WRN*, a warning will be displayed, but the bound is inputted as specified.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the variable. (input)
- `bkx` (*Boundkey*) – New bound key. (input)
- `blx` (`Float64`) – New lower bound. (input)
- `bux` (`Float64`) – New upper bound. (input)

Groups

Problem data - linear part, *Problem data - variables*, *Problem data - bounds*

`putvarboundlist`

```
function putvarboundlist(task::MSKtask,
                         sub::Vector{Int32},
                         bkx::Vector{Boundkey},
                         blx::Vector{Float64},
                         bux::Vector{Float64})
function putvarboundlist(task::MSKtask,
                         sub::T0,
                         bkx::Vector{Boundkey},
                         blx::T1,
                         bux::T2)
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Number},
            T2<:AbstractVector{<:Number} }
```

Changes the bounds for one or more variables. If multiple bound changes are specified for a variable, then only the last change takes effect. Data checks are performed as in *putvarbound*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)

- `sub` (`Int32[]`) – List of variable indexes. (input)
- `bkx` (*`Boundkey`* `[]`) – Bound keys for the variables. (input)
- `blx` (`Float64[]`) – Lower bounds for the variables. (input)
- `bux` (`Float64[]`) – Upper bounds for the variables. (input)

Groups

Problem data - linear part, *Problem data - variables*, *Problem data - bounds*

`putvarboundlistconst`

```
function putvarboundlistconst(task::MSKtask,
                              sub::Vector{Int32},
                              bkx::Boundkey,
                              blx::Float64,
                              bux::Float64)
function putvarboundlistconst(task::MSKtask,
                              sub::T0,
                              bkx::Boundkey,
                              blx::T1,
                              bux::T2)
    where { T0<:AbstractVector{<:Integer},
            T1<:Number,
            T2<:Number }
```

Changes the bounds for one or more variables. Data checks are performed as in *`putvarbound`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `sub` (`Int32[]`) – List of variable indexes. (input)
- `bkx` (*`Boundkey`*) – New bound key for all variables in the list. (input)
- `blx` (`Float64`) – New lower bound for all variables in the list. (input)
- `bux` (`Float64`) – New upper bound for all variables in the list. (input)

Groups

Problem data - linear part, *Problem data - variables*, *Problem data - bounds*

`putvarboundslice`

```
function putvarboundslice(task::MSKtask,
                          first::Int32,
                          last::Int32,
                          bkx::Vector{Boundkey},
                          blx::Vector{Float64},
                          bux::Vector{Float64})
function putvarboundslice(task::MSKtask,
                          first::T0,
                          last::T1,
                          bkx::Vector{Boundkey},
                          blx::T2,
                          bux::T3)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number},
            T3<:AbstractVector{<:Number} }
```

Changes the bounds for a slice of the variables. Data checks are performed as in *`putvarbound`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – First index in the sequence. (input)

- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `bkx` (*`Boundkey`*`[]`) – Bound keys for the variables. (input)
- `blx` (`Float64[]`) – Lower bounds for the variables. (input)
- `bux` (`Float64[]`) – Upper bounds for the variables. (input)

Groups

Problem data - linear part, *Problem data - variables*, *Problem data - bounds*

`putvarboundsliceconst`

```
function putvarboundsliceconst(task::MSKtask,
                               first::Int32,
                               last::Int32,
                               bkx::Boundkey,
                               blx::Float64,
                               bux::Float64)
function putvarboundsliceconst(task::MSKtask,
                               first::T0,
                               last::T1,
                               bkx::Boundkey,
                               blx::T2,
                               bux::T3)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Number,
            T3<:Number }
```

Changes the bounds for a slice of the variables. Data checks are performed as in *`putvarbound`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `bkx` (*`Boundkey`*) – New bound key for all variables in the slice. (input)
- `blx` (`Float64`) – New lower bound for all variables in the slice. (input)
- `bux` (`Float64`) – New upper bound for all variables in the slice. (input)

Groups

Problem data - linear part, *Problem data - variables*, *Problem data - bounds*

`putvarname`

```
function putvarname(task::MSKtask,
                    j::Int32,
                    name::Union{Nothing,AbstractString})
function putvarname(task::MSKtask,
                    j::T0,
                    name::Union{Nothing,AbstractString})
    where { T0<:Integer }
```

Sets the name of a variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the variable. (input)
- `name` (`AbstractString`) – The variable name. (input)

Groups

Names, *Problem data - variables*, *Problem data - linear part*

putvarsolutionj

```
function putvarsolutionj(task::MSKtask,
                         j::Int32,
                         whichsol::Soltype,
                         sk::Stakey,
                         x::Float64,
                         sl::Float64,
                         su::Float64,
                         sn::Float64)
function putvarsolutionj(task::MSKtask,
                         j::T0,
                         whichsol::Soltype,
                         sk::Stakey,
                         x::T1,
                         sl::T2,
                         su::T3,
                         sn::T4)
    where { T0<:Integer,
            T1<:Number,
            T2<:Number,
            T3<:Number,
            T4<:Number }
```

Sets the primal and dual solution information for a single variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the variable. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `sk` (*`Stakey`*) – Status key of the variable. (input)
- `x` (`Float64`) – Primal solution value of the variable. (input)
- `sl` (`Float64`) – Solution value of the dual variable associated with the lower bound. (input)
- `su` (`Float64`) – Solution value of the dual variable associated with the upper bound. (input)
- `sn` (`Float64`) – Solution value of the dual variable associated with the conic constraint. (input)

Groups

Solution information, *Solution - primal*, *Solution - dual*

putvartype

```
function putvartype(task::MSKtask,
                    j::Int32,
                    vartype::Variabletype)
function putvartype(task::MSKtask,
                    j::T0,
                    vartype::Variabletype)
    where { T0<:Integer }
```

Sets the variable type of one variable.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `j` (`Int32`) – Index of the variable. (input)
- `vartype` (*`Variabletype`*) – The new variable type. (input)

Groups

Problem data - variables

`putvartypelist`

```
function putvartypelist(task::MSKtask,
                        subj::Vector{Int32},
                        vartype::Vector{Variabletype})
function putvartypelist(task::MSKtask,
                        subj::T0,
                        vartype::Vector{Variabletype})
    where { T0<:AbstractVector{<:Integer} }
```

Sets the variable type for one or more variables. If the same index is specified multiple times in `subj` only the last entry takes effect.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subj` (`Int32[]`) – A list of variable indexes for which the variable type should be changed. (input)
- `vartype` (*`Variabletype`*`[]`) – A list of variable types that should be assigned to the variables specified by `subj`. (input)

Groups

Problem data - variables

`putxc`

```
function putxc(task::MSKtask,
               whichsol::Soltype) -> xc :: Vector{Float64}
```

Sets the x^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Return

`xc` (`Float64[]`) – Primal constraint solution.

Groups

Solution - primal

`putxcslice`

```
function putxcslice(task::MSKtask,
                    whichsol::Soltype,
                    first::Int32,
                    last::Int32,
                    xc::Vector{Float64})
function putxcslice(task::MSKtask,
                    whichsol::Soltype,
                    first::T0,
                    last::T1,
                    xc::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Sets a slice of the x^c vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `xc` (`Float64[]`) – Primal constraint solution. (input)

Groups

Solution - primal

`putxx`

```
function putxx(task::MSKtask,
               whichsol::Soltype,
               xx::Vector{Float64})
function putxx(task::MSKtask,
               whichsol::Soltype,
               xx::T0)
    where { T0<:AbstractVector{<:Number} }
```

Sets the x^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `xx` (`Float64[]`) – Primal variable solution. (input)

Groups

Solution - primal

`putxxslice`

```
function putxxslice(task::MSKtask,
                    whichsol::Soltype,
                    first::Int32,
                    last::Int32,
                    xx::Vector{Float64})
function putxxslice(task::MSKtask,
                    whichsol::Soltype,
                    first::T0,
                    last::T1,
                    xx::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Sets a slice of the x^x vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `xx` (`Float64[]`) – Primal variable solution. (input)

Groups

Solution - primal

`puty`

```
function puty(task::MSKtask,
              whichsol::Soltype,
              y::Vector{Float64})
function puty(task::MSKtask,
              whichsol::Soltype,
              y::T0)
    where { T0<:AbstractVector{<:Number} }
```

Sets the y vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `y` (`Float64[]`) – Vector of dual variables corresponding to the constraints. (input)

Groups

Solution - primal

`putyslice`

```
function putyslice(task::MSKtask,
                   whichsol::Soltype,
                   first::Int32,
                   last::Int32,
                   y::Vector{Float64})
function putyslice(task::MSKtask,
                   whichsol::Soltype,
                   first::T0,
                   last::T1,
                   y::T2)
    where { T0<:Integer,
            T1<:Integer,
            T2<:AbstractVector{<:Number} }
```

Sets a slice of the y vector for a solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `first` (`Int32`) – First index in the sequence. (input)
- `last` (`Int32`) – Last index plus 1 in the sequence. (input)
- `y` (`Float64[]`) – Vector of dual variables corresponding to the constraints. (input)

Groups

Solution - dual

`readbsolution`

```
function readbsolution(task::MSKtask,
                       filename::AbstractString,
                       compress::Compresstype)
```

Read a binary dump of the task solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)
- `compress` (*`Compresstype`*) – Data compression type. (input)

Groups
Input/Output

`readdata`

```
function readdata(task::MSKtask,
                  filename::AbstractString)
```

Reads an optimization problem and associated data from a file.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups
Input/Output

`readdataformat`

```
function readdataformat(task::MSKtask,
                        filename::AbstractString,
                        format::Dataformat,
                        compress::Compresstype)
```

Reads an optimization problem and associated data from a file.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)
- `format` (*`Dataformat`*) – File data format. (input)
- `compress` (*`Compresstype`*) – File compression type. (input)

Groups
Input/Output

`readjsonsol`

```
function readjsonsol(task::MSKtask,
                     filename::AbstractString)
```

Reads a solution file in JSON format (JSOL file) and inserts it in the task. Only the section `Task/solutions` is taken into consideration.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups
Input/Output

`readjsonstring`

```
function readjsonstring(task::MSKtask,
                        data::AbstractString)
```

Load task data from a JSON string, replacing any data that already exists in the task object. All problem data, parameters and other settings are resorted, but if the string contains solutions, the solution status after loading a file is set to unknown, even if it is optimal or otherwise well-defined.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `data` (`AbstractString`) – Problem data in text format. (input)

Groups
Input/Output

`readlpstring`

```
function readlpstring(task::MSKtask,
                      data::AbstractString)
```

Load task data from a string in LP format, replacing any data that already exists in the task object.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `data` (`AbstractString`) – Problem data in text format. (input)

Groups

Input/Output

`readopfstring`

```
function readopfstring(task::MSKtask,
                       data::AbstractString)
```

Load task data from a string in OPF format, replacing any data that already exists in the task object.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `data` (`AbstractString`) – Problem data in text format. (input)

Groups

Input/Output

`readparamfile`

```
function readparamfile(task::MSKtask,
                       filename::AbstractString)
```

Reads **MOSEK** parameters from a file. Data is read from the file `filename` if it is a nonempty string. Otherwise data is read from the file specified by *MSK_SPAR_PARAM_READ_FILE_NAME*.

The parameter file must begin with `BEGIN MOSEK` and end with `END MOSEK`; empty lines and lines starting from a `%` sign are ignored; each remaining line contains a valid **MOSEK** parameter name followed by its value (using generic names as in the command-line tool syntax). Example:

```
BEGIN MOSEK
% This is a comment.
MSK_IPAR_PRESOLVE_USE         MSK_OFF
MSK_DPAR_INTPNT_TOL_PFEAS     1.0e-9
END MOSEK
```

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups

Input/Output, *Parameters*

`readptfstring`

```
function readptfstring(task::MSKtask,
                       data::AbstractString)
```

Load task data from a PTF string, replacing any data that already exists in the task object. All problem data, parameters and other settings are resorted, but if the string contains solutions, the solution status after loading a file is set to unknown, even if it is optimal or otherwise well-defined.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `data` (`AbstractString`) – Problem data in text format. (input)

Groups

Input/Output

`readsolution`

```
function readsolution(task::MSKtask,
                      whichsol::Soltype,
                      filename::AbstractString)
```

Reads a solution file and inserts it as a specified solution in the task. Data is read from the file `filename` if it is a nonempty string. Otherwise data is read from one of the files specified by *MSK_SPAR_BAS_SOL_FILE_NAME*, *MSK_SPAR_ITR_SOL_FILE_NAME* or *MSK_SPAR_INT_SOL_FILE_NAME* depending on which solution is chosen.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups

Input/Output

`readsolutionfile`

```
function readsolutionfile(task::MSKtask,
                          filename::AbstractString)
```

Read solution file in format determined by the filename

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups

Input/Output

`readsummary`

```
function readsummary(task::MSKtask,
                     whichstream::Streamtype)
```

Prints a short summary of last file that was read.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*Streamtype*) – Index of the stream. (input)

Groups

Input/Output, *Inspecting the task*

`readtask`

```
function readtask(task::MSKtask,
                  filename::AbstractString)
```

Load task data from a file, replacing any data that already exists in the task object. All problem data, parameters and other settings are resorted, but if the file contains solutions, the solution status after loading a file is set to unknown, even if it was optimal or otherwise well-defined when the file was dumped.

See section *The Task Format* for a description of the Task format.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups

Input/Output

`removebarvars`

```
function removebarvars(task::MSKtask,
                       subset::Vector{Int32})
function removebarvars(task::MSKtask,
                       subset::T0)
    where { T0<:AbstractVector{<:Integer} }
```

The function removes a subset of the symmetric matrices from the optimization task. This implies that the remaining symmetric matrices are renumbered.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subset` (`Int32[]`) – Indexes of symmetric matrices which should be removed. (input)

Groups

Problem data - semidefinite

~~`removecones`~~ *Deprecated*

```
function removecones(task::MSKtask,
                     subset::Vector{Int32})
function removecones(task::MSKtask,
                     subset::T0)
    where { T0<:AbstractVector{<:Integer} }
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Removes a number of conic constraints from the problem. This implies that the remaining conic constraints are renumbered. In general, it is much more efficient to remove a cone with a high index than a low index.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subset` (`Int32[]`) – Indexes of cones which should be removed. (input)

Groups

Problem data - cones (deprecated)

`removecons`

```
function removecons(task::MSKtask,
                    subset::Vector{Int32})
function removecons(task::MSKtask,
                    subset::T0)
    where { T0<:AbstractVector{<:Integer} }
```

The function removes a subset of the constraints from the optimization task. This implies that the remaining constraints are renumbered.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `subset` (`Int32[]`) – Indexes of constraints which should be removed. (input)

Groups
Problem data - constraints, *Problem data - linear part*

`removevars`

```
function removevars(task::MSKtask,
                    subset::Vector{Int32})
function removevars(task::MSKtask,
                    subset::T0)
    where { T0<:AbstractVector{<:Integer} }
```

The function removes a subset of the variables from the optimization task. This implies that the remaining variables are renumbered.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `subset` (`Int32[]`) – Indexes of variables which should be removed. (input)

Groups
Problem data - variables, *Problem data - linear part*

`rescodetostr`

```
function rescodetostr(res::Rescode) -> str :: String
```

Obtains an identifier string corresponding to a response code.

Parameters
`res` (*Rescode*) – Response code. (input)

Return
`str` (`String`) – String corresponding to the bound response code `res`.

Groups
Names

`resetdouparam`

```
function resetdouparam(task::MSKtask,
                       param::Dparam)
```

Resets a double parameter to its default value.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*Dparam*) – Which parameter. (input)

Groups
Parameters

`resetexpirylicenses`

```
function resetexpirylicenses(env::MSKenv)
function resetexpirylicenses()
```

Reset the license expiry reporting startpoint.

Parameters
`env` (`MSKenv`) – The MOSEK environment. (input)

Groups
License system

`resetintparam`

```
function resetintparam(task::MSKtask,
                       param::Iparam)
```

Resets an integer parameter to its default value.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*Iparam*) – Which parameter. (input)

Groups

Parameters

`resetparameters`

```
function resetparameters(task::MSKtask)
```

Resets all the parameters to their default values.

Parameters

`task` (`MSKtask`) – An optimization task. (input)

Groups

Parameters

`resetstrparam`

```
function resetstrparam(task::MSKtask,
                       param::Sparam)
```

Resets a string parameter to its defalt value.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `param` (*Sparam*) – Which parameter. (input)

Groups

Parameters

`resizetask`

```
function resizetask(task::MSKtask,
                    maxnumcon::Int32,
                    maxnumvar::Int32,
                    maxnumcone::Int32,
                    maxnumanz::Int64,
                    maxnumqnz::Int64)
function resizetask(task::MSKtask,
                    maxnumcon::T0,
                    maxnumvar::T1,
                    maxnumcone::T2,
                    maxnumanz::T3,
                    maxnumqnz::T4)
    where { T0<:Integer,
            T1<:Integer,
            T2<:Integer,
            T3<:Integer,
            T4<:Integer }
```

Sets the amount of preallocated space assigned for each type of data in an optimization task.

It is never mandatory to call this function, since it only gives a hint about the amount of data to preallocate for efficiency reasons.

Please note that the procedure is **destructive** in the sense that all existing data stored in the task is destroyed.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `maxnumcon` (`Int32`) – New maximum number of constraints. (input)
- `maxnumvar` (`Int32`) – New maximum number of variables. (input)
- `maxnumcone` (`Int32`) – New maximum number of cones. (input)
- `maxnumanz` (`Int64`) – New maximum number of non-zeros in A. (input)
- `maxnumqnz` (`Int64`) – New maximum number of non-zeros in all Q matrices. (input)

Groups

Environment and task management

`sensitivityreport`

```
function sensitivityreport(task::MSKtask,
                           whichstream::Streamtype)
```

Reads a sensitivity format file from a location given by *MSK_SPAR_SENSITIVITY_FILE_NAME* and writes the result to the stream `whichstream`. If *MSK_SPAR_SENSITIVITY_RES_FILE_NAME* is set to a non-empty string, then the sensitivity report is also written to a file of this name.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*Streamtype*) – Index of the stream. (input)

Groups

Sensitivity analysis

`solutiondef`

```
function solutiondef(task::MSKtask,
                     whichsol::Soltype) -> isdef :: Bool
```

Checks whether a solution is defined.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*Soltype*) – Selects a solution. (input)

Return

`isdef` (`Bool`) – Is non-zero if the requested solution is defined.

Groups

Solution information

`solutionsummary`

```
function solutionsummary(task::MSKtask,
                         whichstream::Streamtype)
```

Prints a short summary of the current solutions.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichstream` (*Streamtype*) – Index of the stream. (input)

Groups

Logging, *Solution information*

`solvewithbasis`

```
function solvewithbasis(task::MSKtask,
                        transp::Bool,
                        numnz::Int32,
                        sub::Vector{Int32},
                        val::Vector{Float64}) -> numnzout :: Int32
function solvewithbasis(task::MSKtask,
                        transp::Bool,
                        numnz::T0,
                        sub::Vector{Int32},
                        val::Vector{Float64})
    where { T0<:Integer }
    -> numnzout :: Int32
```

If a basic solution is available, then exactly *numcon* basis variables are defined. These *numcon* basis variables are denoted the basis. Associated with the basis is a basis matrix denoted B. This function solves either the linear equation system

$$B\overline{X} = b \tag{15.3}$$

or the system

$$B^T\overline{X} = b \tag{15.4}$$

for the unknowns $\overline{X}$, with b being a user-defined vector. In order to make sense of the solution $\overline{X}$ it is important to know the ordering of the variables in the basis because the ordering specifies how B is constructed. When calling *initbasissolve* an ordering of the basis variables is obtained, which can be used to deduce how **MOSEK** has constructed B. Indeed if the k-th basis variable is variable x_j it implies that

$$B_{i,k} = A_{i,j},\ i = 1, \ldots, \texttt{numcon}.$$

Otherwise if the k-th basis variable is variable x_j^c it implies that

$$B_{i,k} = \begin{cases} -1, & i = j, \\ 0, & i \neq j. \end{cases}$$

The function *initbasissolve* must be called before a call to this function. Please note that this function exploits the sparsity in the vector b to speed up the computations.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `transp` (`Bool`) – If this argument is zero, then (15.3) is solved, if non-zero then (15.4) is solved. (input)
- `numnz` (`Int32`) – The number of non-zeros in b. (input)
- `sub` (`Int32[]`) – As input it contains the positions of non-zeros in b. As output it contains the positions of the non-zeros in $\overline{X}$. It must have room for *numcon* elements. (input/output)
- `val` (`Float64[]`) – As input it is the vector b as a dense vector (although the positions of non-zeros are specified in `sub` it is required that `val`$[i] = 0$ when $b[i] = 0$). As output `val` is the vector $\overline{X}$ as a dense vector. It must have length *numcon*. (input/output)

Return

`numnzout` (`Int32`) – The number of non-zeros in $\overline{X}$.

Groups

Solving systems with basis matrix

`sparsetriangularsolvedense`

```
function sparsetriangularsolvedense(env::MSKenv,
                                    transposed::Transpose,
                                    lnzc::Vector{Int32},
                                    lptrc::Vector{Int64},
                                    lsubc::Vector{Int32},
                                    lvalc::Vector{Float64},
                                    b::Vector{Float64})
function sparsetriangularsolvedense(env::MSKenv,
                                    transposed::Transpose,
                                    lnzc::T0,
                                    lptrc::T1,
                                    lsubc::T2,
                                    lvalc::T3,
                                    b::Vector{Float64})
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number} }
function sparsetriangularsolvedense(transposed::Transpose,
                                    lnzc::Vector{Int32},
                                    lptrc::Vector{Int64},
                                    lsubc::Vector{Int32},
                                    lvalc::Vector{Float64},
                                    b::Vector{Float64})
function sparsetriangularsolvedense(transposed::Transpose,
                                    lnzc::T0,
                                    lptrc::T1,
                                    lsubc::T2,
                                    lvalc::T3,
                                    b::Vector{Float64})
    where { T0<:AbstractVector{<:Integer},
            T1<:AbstractVector{<:Integer},
            T2<:AbstractVector{<:Integer},
            T3<:AbstractVector{<:Number} }
```

The function solves a triangular system of the form

$$Lx = b$$

or

$$L^T x = b$$

where L is a sparse lower triangular nonsingular matrix. This implies in particular that diagonals in L are nonzero.

Parameters

- `env` (`MSKenv`) – The MOSEK environment. (input)
- `transposed` (*`Transpose`*) – Controls whether to use with L or L^T. (input)
- `lnzc` (`Int32[]`) – `lnzc[j]` is the number of nonzeros in column `j`. (input)
- `lptrc` (`Int64[]`) – `lptrc[j]` is a pointer to the first row index and value in column `j`. (input)
- `lsubc` (`Int32[]`) – Row indexes for each column stored sequentially. Must be stored in increasing order for each column. (input)
- `lvalc` (`Float64[]`) – The value corresponding to the row index stored in `lsubc`. (input)
- `b` (`Float64[]`) – The right-hand side of linear equation system to be solved as a dense vector. (input/output)

Groups
Linear algebra

~~strtoconetype~~ *Deprecated*

```
function strtoconetype(task::MSKtask,
                       str::AbstractString) -> conetype :: Conetype
```

NOTE: This interface to conic optimization is deprecated and will be removed in a future major release. Conic problems should be specified using the affine conic constraints interface (ACC), see Sec. 6.2 for details.

Obtains cone type code corresponding to a cone type string.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `str` (`AbstractString`) – String corresponding to the cone type code `conetype`. (input)

Return
`conetype` (*`Conetype`*) – The cone type corresponding to the string `str`.

Groups
Names

`strtosk`

```
function strtosk(task::MSKtask,
                 str::AbstractString) -> sk :: Stakey
```

Obtains the status key corresponding to an abbreviation string.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `str` (`AbstractString`) – A status key abbreviation string. (input)

Return
`sk` (*`Stakey`*) – Status key corresponding to the string.

Groups
Names

`updatesolutioninfo`

```
function updatesolutioninfo(task::MSKtask,
                            whichsol::Soltype)
```

Update the information items related to the solution.

Parameters
- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)

Groups
Information items and statistics

`writebsolution`

```
function writebsolution(task::MSKtask,
                        filename::AbstractString,
                        compress::Compresstype)
```

Write a binary dump of the task solution.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)
- `compress` (*`Compresstype`*) – Data compression type. (input)

Groups

Input/Output

`writedata`

```
function writedata(task::MSKtask,
                   filename::AbstractString)
```

Writes problem data associated with the optimization task to a file in one of the supported formats. See Section *Supported File Formats* for the complete list.

The data file format is determined by the file name extension. To write in compressed format append the extension `.gz`. E.g to write a gzip compressed MPS file use the extension `mps.gz`.

Please note that MPS, LP and OPF files require all variables to have unique names. If a task contains no names, it is possible to write the file with automatically generated anonymous names by setting the *`MSK_IPAR_WRITE_GENERIC_NAMES`* parameter to *`MSK_ON`*.

Data is written to the file `filename` if it is a nonempty string. Otherwise data is written to the file specified by *`MSK_SPAR_DATA_FILE_NAME`*.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups

Input/Output

`writedatastream`

```
function writedatastream(task::Task,
                         format::Dataformat,
                         compress::Compresstype,
                         stream::OutputStream)
```

Writes problem data associated with the optimization task to a stream in one of the supported formats.

Example:

```
        writedatastream(task,MSK_DATA_FORMAT_TASK, MSK_COMPRESS_NONE, open(
↪"outfile.task", "w"))
        writedatastream(task,MSK_DATA_FORMAT_PTF,  MSK_COMPRESS_NONE, stdout)
```

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `format` (*`mosek.dataformat`*) – Data format. (input)
- `compress` (*`mosek.compresstype`*) – Selects compression type. (input)
- `stream` (`OutputStream`) – The output stream. (input)

Groups

Input/Output

`writejsonsol`

```
function writejsonsol(task::MSKtask,
                      filename::AbstractString)
```

Saves the current solutions and solver information items in a JSON file. If the file name has the extensions .gz or .zst, then the file is gzip or Zstd compressed respectively.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups

Input/Output

`writeparamfile`

```
function writeparamfile(task::MSKtask,
                        filename::AbstractString)
```

Writes all the parameters to a parameter file.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups

Input/Output, *Parameters*

`writesolution`

```
function writesolution(task::MSKtask,
                       whichsol::Soltype,
                       filename::AbstractString)
```

Saves the current basic, interior-point, or integer solution to a file.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `whichsol` (*`Soltype`*) – Selects a solution. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups

Input/Output

`writesolutionfile`

```
function writesolutionfile(task::MSKtask,
                           filename::AbstractString)
```

Write solution file in format determined by the filename

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups

Input/Output

`writetask`

```
function writetask(task::MSKtask,
                   filename::AbstractString)
```

Write a binary dump of the task data. This format saves all problem data, coefficients and parameter settings. See section *The Task Format* for a description of the Task format.

Parameters

- `task` (`MSKtask`) – An optimization task. (input)
- `filename` (`AbstractString`) – A valid file name. (input)

Groups

Input/Output

15.4 Parameters grouped by topic

Analysis

- *MSK_DPAR_ANA_SOL_INFEAS_TOL*
- *MSK_IPAR_ANA_SOL_BASIS*
- *MSK_IPAR_ANA_SOL_PRINT_VIOLATED*
- *MSK_IPAR_LOG_ANA_PRO*

Basis identification

- *MSK_DPAR_SIM_LU_TOL_REL_PIV*
- *MSK_IPAR_BI_CLEAN_OPTIMIZER*
- *MSK_IPAR_BI_IGNORE_MAX_ITER*
- *MSK_IPAR_BI_IGNORE_NUM_ERROR*
- *MSK_IPAR_BI_MAX_ITERATIONS*
- *MSK_IPAR_INTPNT_BASIS*
- *MSK_IPAR_LOG_BI*
- *MSK_IPAR_LOG_BI_FREQ*

Conic interior-point method

- *MSK_DPAR_INTPNT_CO_TOL_DFEAS*
- *MSK_DPAR_INTPNT_CO_TOL_INFEAS*
- *MSK_DPAR_INTPNT_CO_TOL_MU_RED*
- *MSK_DPAR_INTPNT_CO_TOL_NEAR_REL*
- *MSK_DPAR_INTPNT_CO_TOL_PFEAS*
- *MSK_DPAR_INTPNT_CO_TOL_REL_GAP*

Data check

- *MSK_DPAR_DATA_SYM_MAT_TOL*
- *MSK_DPAR_DATA_SYM_MAT_TOL_HUGE*
- *MSK_DPAR_DATA_SYM_MAT_TOL_LARGE*
- *MSK_DPAR_DATA_TOL_AIJ_HUGE*
- *MSK_DPAR_DATA_TOL_AIJ_LARGE*
- *MSK_DPAR_DATA_TOL_BOUND_INF*
- *MSK_DPAR_DATA_TOL_BOUND_WRN*
- *MSK_DPAR_DATA_TOL_C_HUGE*
- *MSK_DPAR_DATA_TOL_CJ_LARGE*
- *MSK_DPAR_DATA_TOL_QIJ*
- *MSK_DPAR_DATA_TOL_X*
- *MSK_DPAR_SEMIDEFINITE_TOL_APPROX*

Data input/output

- *MSK_IPAR_INFEAS_REPORT_AUTO*
- *MSK_IPAR_LOG_FILE*
- *MSK_IPAR_OPF_WRITE_HEADER*
- *MSK_IPAR_OPF_WRITE_HINTS*
- *MSK_IPAR_OPF_WRITE_LINE_LENGTH*
- *MSK_IPAR_OPF_WRITE_PARAMETERS*
- *MSK_IPAR_OPF_WRITE_PROBLEM*
- *MSK_IPAR_OPF_WRITE_SOL_BAS*
- *MSK_IPAR_OPF_WRITE_SOL_ITG*
- *MSK_IPAR_OPF_WRITE_SOL_ITR*
- *MSK_IPAR_OPF_WRITE_SOLUTIONS*
- *MSK_IPAR_PARAM_READ_CASE_NAME*
- *MSK_IPAR_PARAM_READ_IGN_ERROR*
- *MSK_IPAR_PTF_WRITE_PARAMETERS*
- *MSK_IPAR_PTF_WRITE_SINGLE_PSD_TERMS*
- *MSK_IPAR_PTF_WRITE_SOLUTIONS*
- *MSK_IPAR_PTF_WRITE_TRANSFORM*
- *MSK_IPAR_READ_ASYNC*
- *MSK_IPAR_READ_DEBUG*
- *MSK_IPAR_READ_KEEP_FREE_CON*
- *MSK_IPAR_READ_MPS_FORMAT*
- *MSK_IPAR_READ_MPS_WIDTH*
- *MSK_IPAR_READ_TASK_IGNORE_PARAM*
- *MSK_IPAR_SOL_READ_NAME_WIDTH*
- *MSK_IPAR_SOL_READ_WIDTH*
- *MSK_IPAR_WRITE_ASYNC*
- *MSK_IPAR_WRITE_BAS_CONSTRAINTS*
- *MSK_IPAR_WRITE_BAS_HEAD*
- *MSK_IPAR_WRITE_BAS_VARIABLES*
- *MSK_IPAR_WRITE_COMPRESSION*
- *MSK_IPAR_WRITE_FREE_CON*
- *MSK_IPAR_WRITE_GENERIC_NAMES*
- *MSK_IPAR_WRITE_IGNORE_INCOMPATIBLE_ITEMS*
- *MSK_IPAR_WRITE_INT_CONSTRAINTS*
- *MSK_IPAR_WRITE_INT_HEAD*

- *MSK_IPAR_WRITE_INT_VARIABLES*
- *MSK_IPAR_WRITE_JSON_INDENTATION*
- *MSK_IPAR_WRITE_LP_FULL_OBJ*
- *MSK_IPAR_WRITE_LP_LINE_WIDTH*
- *MSK_IPAR_WRITE_MPS_FORMAT*
- *MSK_IPAR_WRITE_MPS_INT*
- *MSK_IPAR_WRITE_SOL_BARVARIABLES*
- *MSK_IPAR_WRITE_SOL_CONSTRAINTS*
- *MSK_IPAR_WRITE_SOL_HEAD*
- *MSK_IPAR_WRITE_SOL_IGNORE_INVALID_NAMES*
- *MSK_IPAR_WRITE_SOL_VARIABLES*
- *MSK_SPAR_BAS_SOL_FILE_NAME*
- *MSK_SPAR_DATA_FILE_NAME*
- *MSK_SPAR_DEBUG_FILE_NAME*
- *MSK_SPAR_INT_SOL_FILE_NAME*
- *MSK_SPAR_ITR_SOL_FILE_NAME*
- *MSK_SPAR_MIO_DEBUG_STRING*
- *MSK_SPAR_PARAM_COMMENT_SIGN*
- *MSK_SPAR_PARAM_READ_FILE_NAME*
- *MSK_SPAR_PARAM_WRITE_FILE_NAME*
- *MSK_SPAR_READ_MPS_BOU_NAME*
- *MSK_SPAR_READ_MPS_OBJ_NAME*
- *MSK_SPAR_READ_MPS_RAN_NAME*
- *MSK_SPAR_READ_MPS_RHS_NAME*
- *MSK_SPAR_SENSITIVITY_FILE_NAME*
- *MSK_SPAR_SENSITIVITY_RES_FILE_NAME*
- *MSK_SPAR_SOL_FILTER_XC_LOW*
- *MSK_SPAR_SOL_FILTER_XC_UPR*
- *MSK_SPAR_SOL_FILTER_XX_LOW*
- *MSK_SPAR_SOL_FILTER_XX_UPR*
- *MSK_SPAR_STAT_KEY*
- *MSK_SPAR_STAT_NAME*

Debugging

- *MSK_IPAR_AUTO_SORT_A_BEFORE_OPT*

Dual simplex

- *MSK_IPAR_SIM_DUAL_CRASH*
- *MSK_IPAR_SIM_DUAL_RESTRICT_SELECTION*
- *MSK_IPAR_SIM_DUAL_SELECTION*

Infeasibility report

- *MSK_IPAR_INFEAS_GENERIC_NAMES*
- *MSK_IPAR_INFEAS_REPORT_LEVEL*
- *MSK_IPAR_LOG_INFEAS_ANA*

Interior-point method

- *MSK_DPAR_INTPNT_CO_TOL_DFEAS*
- *MSK_DPAR_INTPNT_CO_TOL_INFEAS*
- *MSK_DPAR_INTPNT_CO_TOL_MU_RED*
- *MSK_DPAR_INTPNT_CO_TOL_NEAR_REL*
- *MSK_DPAR_INTPNT_CO_TOL_PFEAS*
- *MSK_DPAR_INTPNT_CO_TOL_REL_GAP*
- *MSK_DPAR_INTPNT_QO_TOL_DFEAS*
- *MSK_DPAR_INTPNT_QO_TOL_INFEAS*
- *MSK_DPAR_INTPNT_QO_TOL_MU_RED*
- *MSK_DPAR_INTPNT_QO_TOL_NEAR_REL*
- *MSK_DPAR_INTPNT_QO_TOL_PFEAS*
- *MSK_DPAR_INTPNT_QO_TOL_REL_GAP*
- *MSK_DPAR_INTPNT_TOL_DFEAS*
- *MSK_DPAR_INTPNT_TOL_DSAFE*
- *MSK_DPAR_INTPNT_TOL_INFEAS*
- *MSK_DPAR_INTPNT_TOL_MU_RED*
- *MSK_DPAR_INTPNT_TOL_PATH*
- *MSK_DPAR_INTPNT_TOL_PFEAS*
- *MSK_DPAR_INTPNT_TOL_PSAFE*
- *MSK_DPAR_INTPNT_TOL_REL_GAP*
- *MSK_DPAR_INTPNT_TOL_REL_STEP*
- *MSK_DPAR_INTPNT_TOL_STEP_SIZE*
- *MSK_DPAR_QCQO_REFORMULATE_REL_DROP_TOL*

- *MSK_IPAR_BI_IGNORE_MAX_ITER*
- *MSK_IPAR_BI_IGNORE_NUM_ERROR*
- *MSK_IPAR_INTPNT_BASIS*
- *MSK_IPAR_INTPNT_DIFF_STEP*
- *MSK_IPAR_INTPNT_HOTSTART*
- *MSK_IPAR_INTPNT_MAX_ITERATIONS*
- *MSK_IPAR_INTPNT_MAX_NUM_COR*
- *MSK_IPAR_INTPNT_OFF_COL_TRH*
- *MSK_IPAR_INTPNT_ORDER_GP_NUM_SEEDS*
- *MSK_IPAR_INTPNT_ORDER_METHOD*
- *MSK_IPAR_INTPNT_REGULARIZATION_USE*
- *MSK_IPAR_INTPNT_SCALING*
- *MSK_IPAR_INTPNT_SOLVE_FORM*
- *MSK_IPAR_INTPNT_STARTING_POINT*
- *MSK_IPAR_LOG_INTPNT*

License manager

- *MSK_IPAR_CACHE_LICENSE*
- *MSK_IPAR_LICENSE_DEBUG*
- *MSK_IPAR_LICENSE_PAUSE_TIME*
- *MSK_IPAR_LICENSE_SUPPRESS_EXPIRE_WRNS*
- *MSK_IPAR_LICENSE_TRH_EXPIRY_WRN*
- *MSK_IPAR_LICENSE_WAIT*

Logging

- *MSK_IPAR_HEARTBEAT_SIM_FREQ_TICKS*
- *MSK_IPAR_LOG*
- *MSK_IPAR_LOG_ANA_PRO*
- *MSK_IPAR_LOG_BI*
- *MSK_IPAR_LOG_BI_FREQ*
- *MSK_IPAR_LOG_CUT_SECOND_OPT*
- *MSK_IPAR_LOG_EXPAND*
- *MSK_IPAR_LOG_FEAS_REPAIR*
- *MSK_IPAR_LOG_FILE*
- *MSK_IPAR_LOG_INCLUDE_SUMMARY*
- *MSK_IPAR_LOG_INFEAS_ANA*

- *MSK_IPAR_LOG_INTPNT*
- *MSK_IPAR_LOG_LOCAL_INFO*
- *MSK_IPAR_LOG_MIO*
- *MSK_IPAR_LOG_MIO_FREQ*
- *MSK_IPAR_LOG_ORDER*
- *MSK_IPAR_LOG_PRESOLVE*
- *MSK_IPAR_LOG_SENSITIVITY*
- *MSK_IPAR_LOG_SENSITIVITY_OPT*
- *MSK_IPAR_LOG_SIM*
- *MSK_IPAR_LOG_SIM_FREQ*
- *MSK_IPAR_LOG_SIM_FREQ_GIGA_TICKS*
- *MSK_IPAR_LOG_STORAGE*

Mixed-integer optimization

- *MSK_DPAR_MIO_CLIQUE_TABLE_SIZE_FACTOR*
- *MSK_DPAR_MIO_DJC_MAX_BIGM*
- *MSK_DPAR_MIO_MAX_TIME*
- *MSK_DPAR_MIO_REL_GAP_CONST*
- *MSK_DPAR_MIO_TOL_ABS_GAP*
- *MSK_DPAR_MIO_TOL_ABS_RELAX_INT*
- *MSK_DPAR_MIO_TOL_FEAS*
- *MSK_DPAR_MIO_TOL_REL_DUAL_BOUND_IMPROVEMENT*
- *MSK_DPAR_MIO_TOL_REL_GAP*
- *MSK_IPAR_LOG_MIO*
- *MSK_IPAR_LOG_MIO_FREQ*
- *MSK_IPAR_MIO_BRANCH_DIR*
- *MSK_IPAR_MIO_CONFLICT_ANALYSIS_LEVEL*
- *MSK_IPAR_MIO_CONIC_OUTER_APPROXIMATION*
- *MSK_IPAR_MIO_CONSTRUCT_SOL*
- *MSK_IPAR_MIO_CROSSOVER_MAX_NODES*
- *MSK_IPAR_MIO_CUT_CLIQUE*
- *MSK_IPAR_MIO_CUT_CMIR*
- *MSK_IPAR_MIO_CUT_GMI*
- *MSK_IPAR_MIO_CUT_IMPLIED_BOUND*
- *MSK_IPAR_MIO_CUT_KNAPSACK_COVER*
- *MSK_IPAR_MIO_CUT_LIPRO*

- *MSK_IPAR_MIO_CUT_SELECTION_LEVEL*
- *MSK_IPAR_MIO_DATA_PERMUTATION_METHOD*
- *MSK_IPAR_MIO_DUAL_RAY_ANALYSIS_LEVEL*
- *MSK_IPAR_MIO_FEASPUMP_LEVEL*
- *MSK_IPAR_MIO_HEURISTIC_LEVEL*
- *MSK_IPAR_MIO_INDEPENDENT_BLOCK_LEVEL*
- *MSK_IPAR_MIO_MAX_NUM_BRANCHES*
- *MSK_IPAR_MIO_MAX_NUM_RELAXS*
- *MSK_IPAR_MIO_MAX_NUM_RESTARTS*
- *MSK_IPAR_MIO_MAX_NUM_ROOT_CUT_ROUNDS*
- *MSK_IPAR_MIO_MAX_NUM_SOLUTIONS*
- *MSK_IPAR_MIO_MEMORY_EMPHASIS_LEVEL*
- *MSK_IPAR_MIO_MIN_REL*
- *MSK_IPAR_MIO_NODE_OPTIMIZER*
- *MSK_IPAR_MIO_NODE_SELECTION*
- *MSK_IPAR_MIO_NUMERICAL_EMPHASIS_LEVEL*
- *MSK_IPAR_MIO_OPT_FACE_MAX_NODES*
- *MSK_IPAR_MIO_PERSPECTIVE_REFORMULATE*
- *MSK_IPAR_MIO_PROBING_LEVEL*
- *MSK_IPAR_MIO_PROPAGATE_OBJECTIVE_CONSTRAINT*
- *MSK_IPAR_MIO_QCQO_REFORMULATION_METHOD*
- *MSK_IPAR_MIO_RENS_MAX_NODES*
- *MSK_IPAR_MIO_RINS_MAX_NODES*
- *MSK_IPAR_MIO_ROOT_OPTIMIZER*
- *MSK_IPAR_MIO_SEED*
- *MSK_IPAR_MIO_SYMMETRY_LEVEL*
- *MSK_IPAR_MIO_VAR_SELECTION*
- *MSK_IPAR_MIO_VB_DETECTION_LEVEL*

Output information

- *MSK_IPAR_HEARTBEAT_SIM_FREQ_TICKS*
- *MSK_IPAR_INFEAS_REPORT_LEVEL*
- *MSK_IPAR_LICENSE_SUPPRESS_EXPIRE_WRNS*
- *MSK_IPAR_LICENSE_TRH_EXPIRY_WRN*
- *MSK_IPAR_LOG*
- *MSK_IPAR_LOG_BI*
- *MSK_IPAR_LOG_BI_FREQ*
- *MSK_IPAR_LOG_CUT_SECOND_OPT*
- *MSK_IPAR_LOG_EXPAND*
- *MSK_IPAR_LOG_FEAS_REPAIR*
- *MSK_IPAR_LOG_FILE*
- *MSK_IPAR_LOG_INCLUDE_SUMMARY*
- *MSK_IPAR_LOG_INFEAS_ANA*
- *MSK_IPAR_LOG_INTPNT*
- *MSK_IPAR_LOG_LOCAL_INFO*
- *MSK_IPAR_LOG_MIO*
- *MSK_IPAR_LOG_MIO_FREQ*
- *MSK_IPAR_LOG_ORDER*
- *MSK_IPAR_LOG_SENSITIVITY*
- *MSK_IPAR_LOG_SENSITIVITY_OPT*
- *MSK_IPAR_LOG_SIM*
- *MSK_IPAR_LOG_SIM_FREQ*
- *MSK_IPAR_LOG_SIM_FREQ_GIGA_TICKS*
- *MSK_IPAR_LOG_STORAGE*
- *MSK_IPAR_MAX_NUM_WARNINGS*

Overall solver

- *MSK_IPAR_BI_CLEAN_OPTIMIZER*
- *MSK_IPAR_LICENSE_WAIT*
- *MSK_IPAR_MIO_MODE*
- *MSK_IPAR_OPTIMIZER*
- *MSK_IPAR_PRESOLVE_MAX_NUM_REDUCTIONS*
- *MSK_IPAR_PRESOLVE_USE*
- *MSK_IPAR_PRIMAL_REPAIR_OPTIMIZER*
- *MSK_IPAR_SENSITIVITY_ALL*
- *MSK_IPAR_SENSITIVITY_TYPE*
- *MSK_IPAR_SIM_PRECISION*

Overall system

- *MSK_IPAR_AUTO_UPDATE_SOL_INFO*
- *MSK_IPAR_LICENSE_WAIT*
- *MSK_IPAR_LOG_STORAGE*
- *MSK_IPAR_MT_SPINCOUNT*
- *MSK_IPAR_NUM_THREADS*
- *MSK_IPAR_REMOVE_UNUSED_SOLUTIONS*
- *MSK_IPAR_TIMING_LEVEL*
- *MSK_SPAR_REMOTE_OPTSERVER_HOST*
- *MSK_SPAR_REMOTE_TLS_CERT*
- *MSK_SPAR_REMOTE_TLS_CERT_PATH*

Presolve

- *MSK_DPAR_FOLDING_TOL_EQ*
- *MSK_DPAR_PRESOLVE_TOL_ABS_LINDEP*
- *MSK_DPAR_PRESOLVE_TOL_PRIMAL_INFEAS_PERTURBATION*
- *MSK_DPAR_PRESOLVE_TOL_REL_LINDEP*
- *MSK_DPAR_PRESOLVE_TOL_S*
- *MSK_DPAR_PRESOLVE_TOL_X*
- *MSK_IPAR_FOLDING_USE*
- *MSK_IPAR_MIO_PRESOLVE_AGGREGATOR_USE*
- *MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_FILL*
- *MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_NUM_TRIES*
- *MSK_IPAR_PRESOLVE_LINDEP_ABS_WORK_TRH*
- *MSK_IPAR_PRESOLVE_LINDEP_NEW*
- *MSK_IPAR_PRESOLVE_LINDEP_REL_WORK_TRH*
- *MSK_IPAR_PRESOLVE_LINDEP_USE*
- *MSK_IPAR_PRESOLVE_MAX_NUM_PASS*
- *MSK_IPAR_PRESOLVE_MAX_NUM_REDUCTIONS*
- *MSK_IPAR_PRESOLVE_USE*

Primal simplex

- *MSK_IPAR_SIM_PRIMAL_CRASH*
- *MSK_IPAR_SIM_PRIMAL_RESTRICT_SELECTION*
- *MSK_IPAR_SIM_PRIMAL_SELECTION*

Simplex optimizer

- *MSK_DPAR_BASIS_REL_TOL_S*
- *MSK_DPAR_BASIS_TOL_S*
- *MSK_DPAR_BASIS_TOL_X*
- *MSK_DPAR_SIM_LU_TOL_REL_PIV*
- *MSK_DPAR_SIM_PRECISION_SCALING_EXTENDED*
- *MSK_DPAR_SIM_PRECISION_SCALING_NORMAL*
- *MSK_DPAR_SIMPLEX_ABS_TOL_PIV*
- *MSK_IPAR_BASIS_SOLVE_USE_PLUS_ONE*
- *MSK_IPAR_HEARTBEAT_SIM_FREQ_TICKS*
- *MSK_IPAR_LOG_SIM*
- *MSK_IPAR_LOG_SIM_FREQ*
- *MSK_IPAR_LOG_SIM_FREQ_GIGA_TICKS*
- *MSK_IPAR_SIM_BASIS_FACTOR_USE*
- *MSK_IPAR_SIM_DEGEN*
- *MSK_IPAR_SIM_DETECT_PWL*
- *MSK_IPAR_SIM_DUAL_PHASEONE_METHOD*
- *MSK_IPAR_SIM_EXPLOIT_DUPVEC*
- *MSK_IPAR_SIM_HOTSTART*
- *MSK_IPAR_SIM_HOTSTART_LU*
- *MSK_IPAR_SIM_MAX_ITERATIONS*
- *MSK_IPAR_SIM_MAX_NUM_SETBACKS*
- *MSK_IPAR_SIM_NON_SINGULAR*
- *MSK_IPAR_SIM_PRECISION_BOOST*
- *MSK_IPAR_SIM_PRIMAL_PHASEONE_METHOD*
- *MSK_IPAR_SIM_REFACTOR_FREQ*
- *MSK_IPAR_SIM_REFORMULATION*
- *MSK_IPAR_SIM_SAVE_LU*
- *MSK_IPAR_SIM_SCALING*
- *MSK_IPAR_SIM_SCALING_METHOD*
- *MSK_IPAR_SIM_SEED*
- *MSK_IPAR_SIM_SOLVE_FORM*
- *MSK_IPAR_SIM_SWITCH_OPTIMIZER*

Solution input/output

- *MSK_IPAR_INFEAS_REPORT_AUTO*
- *MSK_IPAR_SOL_FILTER_KEEP_BASIC*
- *MSK_IPAR_SOL_READ_NAME_WIDTH*
- *MSK_IPAR_SOL_READ_WIDTH*
- *MSK_IPAR_WRITE_BAS_CONSTRAINTS*
- *MSK_IPAR_WRITE_BAS_HEAD*
- *MSK_IPAR_WRITE_BAS_VARIABLES*
- *MSK_IPAR_WRITE_INT_CONSTRAINTS*
- *MSK_IPAR_WRITE_INT_HEAD*
- *MSK_IPAR_WRITE_INT_VARIABLES*
- *MSK_IPAR_WRITE_SOL_BARVARIABLES*
- *MSK_IPAR_WRITE_SOL_CONSTRAINTS*
- *MSK_IPAR_WRITE_SOL_HEAD*
- *MSK_IPAR_WRITE_SOL_IGNORE_INVALID_NAMES*
- *MSK_IPAR_WRITE_SOL_VARIABLES*
- *MSK_SPAR_BAS_SOL_FILE_NAME*
- *MSK_SPAR_INT_SOL_FILE_NAME*
- *MSK_SPAR_ITR_SOL_FILE_NAME*
- *MSK_SPAR_SOL_FILTER_XC_LOW*
- *MSK_SPAR_SOL_FILTER_XC_UPR*
- *MSK_SPAR_SOL_FILTER_XX_LOW*
- *MSK_SPAR_SOL_FILTER_XX_UPR*

Termination criteria

- *MSK_DPAR_BASIS_REL_TOL_S*
- *MSK_DPAR_BASIS_TOL_S*
- *MSK_DPAR_BASIS_TOL_X*
- *MSK_DPAR_INTPNT_CO_TOL_DFEAS*
- *MSK_DPAR_INTPNT_CO_TOL_INFEAS*
- *MSK_DPAR_INTPNT_CO_TOL_MU_RED*
- *MSK_DPAR_INTPNT_CO_TOL_NEAR_REL*
- *MSK_DPAR_INTPNT_CO_TOL_PFEAS*
- *MSK_DPAR_INTPNT_CO_TOL_REL_GAP*
- *MSK_DPAR_INTPNT_QO_TOL_DFEAS*
- *MSK_DPAR_INTPNT_QO_TOL_INFEAS*

- *MSK_DPAR_INTPNT_QO_TOL_MU_RED*
- *MSK_DPAR_INTPNT_QO_TOL_NEAR_REL*
- *MSK_DPAR_INTPNT_QO_TOL_PFEAS*
- *MSK_DPAR_INTPNT_QO_TOL_REL_GAP*
- *MSK_DPAR_INTPNT_TOL_DFEAS*
- *MSK_DPAR_INTPNT_TOL_INFEAS*
- *MSK_DPAR_INTPNT_TOL_MU_RED*
- *MSK_DPAR_INTPNT_TOL_PFEAS*
- *MSK_DPAR_INTPNT_TOL_REL_GAP*
- *MSK_DPAR_LOWER_OBJ_CUT*
- *MSK_DPAR_LOWER_OBJ_CUT_FINITE_TRH*
- *MSK_DPAR_MIO_MAX_TIME*
- *MSK_DPAR_MIO_REL_GAP_CONST*
- *MSK_DPAR_MIO_TOL_REL_GAP*
- *MSK_DPAR_OPTIMIZER_MAX_TICKS*
- *MSK_DPAR_OPTIMIZER_MAX_TIME*
- *MSK_DPAR_SIM_PRECISION_SCALING_EXTENDED*
- *MSK_DPAR_SIM_PRECISION_SCALING_NORMAL*
- *MSK_DPAR_UPPER_OBJ_CUT*
- *MSK_DPAR_UPPER_OBJ_CUT_FINITE_TRH*
- *MSK_IPAR_BI_MAX_ITERATIONS*
- *MSK_IPAR_INTPNT_MAX_ITERATIONS*
- *MSK_IPAR_MIO_MAX_NUM_BRANCHES*
- *MSK_IPAR_MIO_MAX_NUM_ROOT_CUT_ROUNDS*
- *MSK_IPAR_MIO_MAX_NUM_SOLUTIONS*
- *MSK_IPAR_SIM_MAX_ITERATIONS*

Other

- *MSK_IPAR_COMPRESS_STATFILE*
- *MSK_IPAR_GETDUAL_CONVERT_LMIS*
- *MSK_IPAR_NG*
- *MSK_IPAR_REMOTE_USE_COMPRESSION*

15.5 Parameters (alphabetical list sorted by type)

- *Double parameters*
- *Integer parameters*
- *String parameters*

15.5.1 Double parameters

`dparam`

The enumeration type containing all double parameters.

`MSK_DPAR_ANA_SOL_INFEAS_TOL`

If a constraint violates its bound with an amount larger than this value, the constraint name, index and violation will be printed by the solution analyzer.

Default
1e-6

Accepted
[0.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_ANA_SOL_INFEAS_TOL, 1e-6)
```

Groups
Analysis

`MSK_DPAR_BASIS_REL_TOL_S`

Maximum relative dual bound violation allowed in an optimal basic solution.

Default
1.0e-12

Accepted
[0.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_BASIS_REL_TOL_S, 1.0e-12)
```

Groups
Simplex optimizer, *Termination criteria*

`MSK_DPAR_BASIS_TOL_S`

Maximum absolute dual bound violation in an optimal basic solution.

Default
1.0e-6

Accepted
[1.0e-9; +inf]

Example
```
putdouparam(task, MSK_DPAR_BASIS_TOL_S, 1.0e-6)
```

Groups
Simplex optimizer, *Termination criteria*

`MSK_DPAR_BASIS_TOL_X`

Maximum absolute primal bound violation allowed in an optimal basic solution.

Default
1.0e-6

Accepted
[1.0e-9; +inf]

Example
```
putdouparam(task, MSK_DPAR_BASIS_TOL_X, 1.0e-6)
```

Groups
Simplex optimizer, *Termination criteria*

`MSK_DPAR_DATA_SYM_MAT_TOL`

Absolute zero tolerance for elements in in symmetric matrices. If any value in a symmetric matrix is smaller than this parameter in absolute terms **MOSEK** will treat the values as zero and generate a warning.

Default
1.0e-12
Accepted
[1.0e-16; 1.0e-6]
Example
```
putdouparam(task, MSK_DPAR_DATA_SYM_MAT_TOL, 1.0e-12)
```
Groups
Data check

`MSK_DPAR_DATA_SYM_MAT_TOL_HUGE`

An element in a symmetric matrix which is larger than this value in absolute size causes an error.

Default
1.0e20
Accepted
[0.0; +inf]
Example
```
putdouparam(task, MSK_DPAR_DATA_SYM_MAT_TOL_HUGE, 1.0e20)
```
Groups
Data check

`MSK_DPAR_DATA_SYM_MAT_TOL_LARGE`

An element in a symmetric matrix which is larger than this value in absolute size causes a warning message to be printed.

Default
1.0e10
Accepted
[0.0; +inf]
Example
```
putdouparam(task, MSK_DPAR_DATA_SYM_MAT_TOL_LARGE, 1.0e10)
```
Groups
Data check

`MSK_DPAR_DATA_TOL_AIJ_HUGE`

An element in A which is larger than this value in absolute size causes an error.

Default
1.0e20
Accepted
[0.0; +inf]
Example
```
putdouparam(task, MSK_DPAR_DATA_TOL_AIJ_HUGE, 1.0e20)
```
Groups
Data check

`MSK_DPAR_DATA_TOL_AIJ_LARGE`

An element in A which is larger than this value in absolute size causes a warning message to be printed.

Default
1.0e10
Accepted
[0.0; +inf]
Example
`putdouparam(task, MSK_DPAR_DATA_TOL_AIJ_LARGE, 1.0e10)`
Groups
Data check

`MSK_DPAR_DATA_TOL_BOUND_INF`

Any bound which in absolute value is greater than this parameter is considered infinite.

Default
1.0e16
Accepted
[0.0; +inf]
Example
`putdouparam(task, MSK_DPAR_DATA_TOL_BOUND_INF, 1.0e16)`
Groups
Data check

`MSK_DPAR_DATA_TOL_BOUND_WRN`

If a bound value is larger than this value in absolute size, then a warning message is issued.

Default
1.0e8
Accepted
[0.0; +inf]
Example
`putdouparam(task, MSK_DPAR_DATA_TOL_BOUND_WRN, 1.0e8)`
Groups
Data check

`MSK_DPAR_DATA_TOL_C_HUGE`

An element in c which is larger than the value of this parameter in absolute terms is considered to be huge and generates an error.

Default
1.0e16
Accepted
[0.0; +inf]
Example
`putdouparam(task, MSK_DPAR_DATA_TOL_C_HUGE, 1.0e16)`
Groups
Data check

`MSK_DPAR_DATA_TOL_CJ_LARGE`

An element in c which is larger than this value in absolute terms causes a warning message to be printed.

Default
1.0e8
Accepted
[0.0; +inf]
Example
`putdouparam(task, MSK_DPAR_DATA_TOL_CJ_LARGE, 1.0e8)`
Groups
Data check

`MSK_DPAR_DATA_TOL_QIJ`

Absolute zero tolerance for elements in Q matrices.

Default
1.0e-16
Accepted
[0.0; +inf]
Example
```
putdouparam(task, MSK_DPAR_DATA_TOL_QIJ, 1.0e-16)
```
Groups
Data check

`MSK_DPAR_DATA_TOL_X`

Zero tolerance for constraints and variables i.e. if the distance between the lower and upper bound is less than this value, then the lower and upper bound is considered identical.

Default
1.0e-8
Accepted
[0.0; +inf]
Example
```
putdouparam(task, MSK_DPAR_DATA_TOL_X, 1.0e-8)
```
Groups
Data check

`MSK_DPAR_FOLDING_TOL_EQ`

Tolerance for coefficient equality during folding.

Default
1e-9
Accepted
[0.0; +inf]
Example
```
putdouparam(task, MSK_DPAR_FOLDING_TOL_EQ, 1e-9)
```
Groups
Presolve

`MSK_DPAR_INTPNT_CO_TOL_DFEAS`

Dual feasibility tolerance used by the interior-point optimizer for conic problems.

Default
1.0e-8
Accepted
[0.0; 1.0]
Example
```
putdouparam(task, MSK_DPAR_INTPNT_CO_TOL_DFEAS, 1.0e-8)
```
See also
MSK_DPAR_INTPNT_CO_TOL_NEAR_REL
Groups
Interior-point method, *Termination criteria*, *Conic interior-point method*

`MSK_DPAR_INTPNT_CO_TOL_INFEAS`

Infeasibility tolerance used by the interior-point optimizer for conic problems. Controls when the interior-point optimizer declares the model primal or dual infeasible. A small number means the optimizer gets more conservative about declaring the model infeasible.

Default
1.0e-12

Accepted
[0.0; 1.0]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_CO_TOL_INFEAS, 1.0e-12)
```

Groups
Interior-point method, *Termination criteria*, *Conic interior-point method*

`MSK_DPAR_INTPNT_CO_TOL_MU_RED`

Relative complementarity gap tolerance used by the interior-point optimizer for conic problems.

Default
1.0e-8

Accepted
[0.0; 1.0]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_CO_TOL_MU_RED, 1.0e-8)
```

Groups
Interior-point method, *Termination criteria*, *Conic interior-point method*

`MSK_DPAR_INTPNT_CO_TOL_NEAR_REL`

Optimality tolerance used by the interior-point optimizer for conic problems. If **MOSEK** cannot compute a solution that has the prescribed accuracy then it will check if the solution found satisfies the termination criteria with all tolerances multiplied by the value of this parameter. If yes, then the solution is also declared optimal.

Default
1000

Accepted
[1.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_CO_TOL_NEAR_REL, 1000)
```

Groups
Interior-point method, *Termination criteria*, *Conic interior-point method*

`MSK_DPAR_INTPNT_CO_TOL_PFEAS`

Primal feasibility tolerance used by the interior-point optimizer for conic problems.

Default
1.0e-8

Accepted
[0.0; 1.0]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_CO_TOL_PFEAS, 1.0e-8)
```

See also
MSK_DPAR_INTPNT_CO_TOL_NEAR_REL

Groups
Interior-point method, *Termination criteria*, *Conic interior-point method*

`MSK_DPAR_INTPNT_CO_TOL_REL_GAP`

Relative gap termination tolerance used by the interior-point optimizer for conic problems.

Default
1.0e-8

Accepted
[0.0; 1.0]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_CO_TOL_REL_GAP, 1.0e-8)
```

See also
MSK_DPAR_INTPNT_CO_TOL_NEAR_REL

Groups
Interior-point method, *Termination criteria*, *Conic interior-point method*

`MSK_DPAR_INTPNT_QO_TOL_DFEAS`

Dual feasibility tolerance used by the interior-point optimizer for quadratic problems.

Default
1.0e-8

Accepted
[0.0; 1.0]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_QO_TOL_DFEAS, 1.0e-8)
```

See also
MSK_DPAR_INTPNT_QO_TOL_NEAR_REL

Groups
Interior-point method, *Termination criteria*

`MSK_DPAR_INTPNT_QO_TOL_INFEAS`

Infeasibility tolerance used by the interior-point optimizer for quadratic problems. Controls when the interior-point optimizer declares the model primal or dual infeasible. A small number means the optimizer gets more conservative about declaring the model infeasible.

Default
1.0e-12

Accepted
[0.0; 1.0]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_QO_TOL_INFEAS, 1.0e-12)
```

Groups
Interior-point method, *Termination criteria*

`MSK_DPAR_INTPNT_QO_TOL_MU_RED`

Relative complementarity gap tolerance used by the interior-point optimizer for quadratic problems.

Default
1.0e-8

Accepted
[0.0; 1.0]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_QO_TOL_MU_RED, 1.0e-8)
```

Groups
Interior-point method, *Termination criteria*

`MSK_DPAR_INTPNT_QO_TOL_NEAR_REL`

Optimality tolerance used by the interior-point optimizer for quadratic problems. If **MOSEK** cannot compute a solution that has the prescribed accuracy then it will check if the solution found satisfies the termination criteria with all tolerances multiplied by the value of this parameter. If yes, then the solution is also declared optimal.

Default
1000

Accepted
[1.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_QO_TOL_NEAR_REL, 1000)
```

Groups
Interior-point method, *Termination criteria*

MSK_DPAR_INTPNT_QO_TOL_PFEAS

Primal feasibility tolerance used by the interior-point optimizer for quadratic problems.

Default
1.0e-8

Accepted
[0.0; 1.0]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_QO_TOL_PFEAS, 1.0e-8)
```

See also
MSK_DPAR_INTPNT_QO_TOL_NEAR_REL

Groups
Interior-point method, *Termination criteria*

MSK_DPAR_INTPNT_QO_TOL_REL_GAP

Relative gap termination tolerance used by the interior-point optimizer for quadratic problems.

Default
1.0e-8

Accepted
[0.0; 1.0]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_QO_TOL_REL_GAP, 1.0e-8)
```

See also
MSK_DPAR_INTPNT_QO_TOL_NEAR_REL

Groups
Interior-point method, *Termination criteria*

MSK_DPAR_INTPNT_TOL_DFEAS

Dual feasibility tolerance used by the interior-point optimizer for linear problems.

Default
1.0e-8

Accepted
[0.0; 1.0]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_TOL_DFEAS, 1.0e-8)
```

Groups
Interior-point method, *Termination criteria*

MSK_DPAR_INTPNT_TOL_DSAFE

Controls the initial dual starting point used by the interior-point optimizer. If the interior-point optimizer converges slowly and/or the constraint or variable bounds are very large, then it might be worthwhile to increase this value.

Default
1.0

Accepted
[1.0e-4; +inf]

Example
```
putdouparam(task, MSK_DPAR_INTPNT_TOL_DSAFE, 1.0)
```

Groups
Interior-point method

MSK_DPAR_INTPNT_TOL_INFEAS

Infeasibility tolerance used by the interior-point optimizer for linear problems. Controls when the interior-point optimizer declares the model primal or dual infeasible. A small number means the optimizer gets more conservative about declaring the model infeasible.

Default
1.0e-10
Accepted
[0.0; 1.0]
Example
`putdouparam(task, MSK_DPAR_INTPNT_TOL_INFEAS, 1.0e-10)`
Groups
Interior-point method, *Termination criteria*

`MSK_DPAR_INTPNT_TOL_MU_RED`

Relative complementarity gap tolerance used by the interior-point optimizer for linear problems.

Default
1.0e-16
Accepted
[0.0; 1.0]
Example
`putdouparam(task, MSK_DPAR_INTPNT_TOL_MU_RED, 1.0e-16)`
Groups
Interior-point method, *Termination criteria*

`MSK_DPAR_INTPNT_TOL_PATH`

Controls how close the interior-point optimizer follows the central path. A large value of this parameter means the central path is followed very closely. On numerically unstable problems it may be worthwhile to increase this parameter.

Default
1.0e-8
Accepted
[0.0; 0.9999]
Example
`putdouparam(task, MSK_DPAR_INTPNT_TOL_PATH, 1.0e-8)`
Groups
Interior-point method

`MSK_DPAR_INTPNT_TOL_PFEAS`

Primal feasibility tolerance used by the interior-point optimizer for linear problems.

Default
1.0e-8
Accepted
[0.0; 1.0]
Example
`putdouparam(task, MSK_DPAR_INTPNT_TOL_PFEAS, 1.0e-8)`
Groups
Interior-point method, *Termination criteria*

`MSK_DPAR_INTPNT_TOL_PSAFE`

Controls the initial primal starting point used by the interior-point optimizer. If the interior-point optimizer converges slowly and/or the constraint or variable bounds are very large, then it may be worthwhile to increase this value.

Default
1.0
Accepted
[1.0e-4; +inf]
Example
`putdouparam(task, MSK_DPAR_INTPNT_TOL_PSAFE, 1.0)`

Groups
Interior-point method

`MSK_DPAR_INTPNT_TOL_REL_GAP`

Relative gap termination tolerance used by the interior-point optimizer for linear problems.

Default
1.0e-8
Accepted
[1.0e-14; +inf]
Example
```
putdouparam(task, MSK_DPAR_INTPNT_TOL_REL_GAP, 1.0e-8)
```
Groups
Termination criteria, *Interior-point method*

`MSK_DPAR_INTPNT_TOL_REL_STEP`

Relative step size to the boundary for linear and quadratic optimization problems.

Default
0.9999
Accepted
[1.0e-4; 0.999999]
Example
```
putdouparam(task, MSK_DPAR_INTPNT_TOL_REL_STEP, 0.9999)
```
Groups
Interior-point method

`MSK_DPAR_INTPNT_TOL_STEP_SIZE`

Minimal step size tolerance. If the step size falls below the value of this parameter, then the interior-point optimizer assumes that it is stalled. In other words the interior-point optimizer does not make any progress and therefore it is better to stop.

Default
1.0e-6
Accepted
[0.0; 1.0]
Example
```
putdouparam(task, MSK_DPAR_INTPNT_TOL_STEP_SIZE, 1.0e-6)
```
Groups
Interior-point method

`MSK_DPAR_LOWER_OBJ_CUT`

If either a primal or dual feasible solution is found proving that the optimal objective value is outside the interval [*MSK_DPAR_LOWER_OBJ_CUT*, *MSK_DPAR_UPPER_OBJ_CUT*], then **MOSEK** is terminated.

Default
-INFINITY
Accepted
[-inf; +inf]
Example
```
putdouparam(task, MSK_DPAR_LOWER_OBJ_CUT, -INFINITY)
```
See also
MSK_DPAR_LOWER_OBJ_CUT_FINITE_TRH
Groups
Termination criteria

`MSK_DPAR_LOWER_OBJ_CUT_FINITE_TRH`

If the lower objective cut is less than the value of this parameter value, then the lower objective cut i.e. *MSK_DPAR_LOWER_OBJ_CUT* is treated as $-\infty$.

Default
-0.5e30
Accepted
[-inf; +inf]
Example
```
putdouparam(task, MSK_DPAR_LOWER_OBJ_CUT_FINITE_TRH, -0.5e30)
```
Groups
Termination criteria

`MSK_DPAR_MIO_CLIQUE_TABLE_SIZE_FACTOR`

Controlls the maximum size of the clique table as a factor of the number of nonzeros in the A matrix. A negative value implies **MOSEK** decides.

Default
-1
Accepted
[-1; +inf]
Example
```
putdouparam(task, MSK_DPAR_MIO_CLIQUE_TABLE_SIZE_FACTOR, -1)
```
Groups
Mixed-integer optimization

`MSK_DPAR_MIO_DJC_MAX_BIGM`

Maximum allowed big-M value when reformulating disjunctive constraints to linear constraints. Higher values make it more likely that a disjunction is reformulated to linear constraints, but also increase the risk of numerical problems.

Default
1.0e6
Accepted
[0; +inf]
Example
```
putdouparam(task, MSK_DPAR_MIO_DJC_MAX_BIGM, 1.0e6)
```
Groups
Mixed-integer optimization

`MSK_DPAR_MIO_MAX_TIME`

This parameter limits the maximum time spent by the mixed-integer optimizer (in seconds). A negative number means infinity.

Default
-1.0
Accepted
[-inf; +inf]
Example
```
putdouparam(task, MSK_DPAR_MIO_MAX_TIME, -1.0)
```
Groups
Mixed-integer optimization, *Termination criteria*

`MSK_DPAR_MIO_REL_GAP_CONST`

This value is used to compute the relative gap for the solution to a mixed-integer optimization problem.

Default
1.0e-10

Accepted
[1.0e-15; +inf]
Example
`putdouparam(task, MSK_DPAR_MIO_REL_GAP_CONST, 1.0e-10)`
Groups
Mixed-integer optimization, *Termination criteria*

`MSK_DPAR_MIO_TOL_ABS_GAP`

Absolute optimality tolerance employed by the mixed-integer optimizer.

Default
0.0
Accepted
[0.0; +inf]
Example
`putdouparam(task, MSK_DPAR_MIO_TOL_ABS_GAP, 0.0)`
Groups
Mixed-integer optimization

`MSK_DPAR_MIO_TOL_ABS_RELAX_INT`

Absolute integer feasibility tolerance. If the distance to the nearest integer is less than this tolerance then an integer constraint is assumed to be satisfied.

Default
1.0e-5
Accepted
[1e-9; +inf]
Example
`putdouparam(task, MSK_DPAR_MIO_TOL_ABS_RELAX_INT, 1.0e-5)`
Groups
Mixed-integer optimization

`MSK_DPAR_MIO_TOL_FEAS`

Feasibility tolerance for mixed integer solver.

Default
1.0e-6
Accepted
[1e-9; 1e-3]
Example
`putdouparam(task, MSK_DPAR_MIO_TOL_FEAS, 1.0e-6)`
Groups
Mixed-integer optimization

`MSK_DPAR_MIO_TOL_REL_DUAL_BOUND_IMPROVEMENT`

If the relative improvement of the dual bound is smaller than this value, the solver will terminate the root cut generation. A value of 0.0 means that the value is selected automatically.

Default
0.0
Accepted
[0.0; 1.0]
Example
`putdouparam(task, MSK_DPAR_MIO_TOL_REL_DUAL_BOUND_IMPROVEMENT, 0.0)`
Groups
Mixed-integer optimization

`MSK_DPAR_MIO_TOL_REL_GAP`

Relative optimality tolerance employed by the mixed-integer optimizer.

Default
1.0e-4

Accepted
[0.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_MIO_TOL_REL_GAP, 1.0e-4)
```

Groups
Mixed-integer optimization, *Termination criteria*

`MSK_DPAR_OPTIMIZER_MAX_TICKS`

CURRENTLY NOT IN USE.

Maximum amount of ticks the optimizer is allowed to spent on the optimization. A negative number means infinity.

Default
-1.0

Accepted
[-inf; +inf]

Example
```
putdouparam(task, MSK_DPAR_OPTIMIZER_MAX_TICKS, -1.0)
```

Groups
Termination criteria

`MSK_DPAR_OPTIMIZER_MAX_TIME`

Maximum amount of time the optimizer is allowed to spent on the optimization (in seconds). A negative number means infinity.

Default
-1.0

Accepted
[-inf; +inf]

Example
```
putdouparam(task, MSK_DPAR_OPTIMIZER_MAX_TIME, -1.0)
```

Groups
Termination criteria

`MSK_DPAR_PRESOLVE_TOL_ABS_LINDEP`

Absolute tolerance employed by the linear dependency checker.

Default
1.0e-6

Accepted
[0.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_PRESOLVE_TOL_ABS_LINDEP, 1.0e-6)
```

Groups
Presolve

`MSK_DPAR_PRESOLVE_TOL_PRIMAL_INFEAS_PERTURBATION`

The presolve is allowed to perturb a bound on a constraint or variable by this amount if it removes an infeasibility.

Default
1.0e-6

Accepted
[0.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_PRESOLVE_TOL_PRIMAL_INFEAS_PERTURBATION,
1.0e-6)
```

Groups
Presolve

`MSK_DPAR_PRESOLVE_TOL_REL_LINDEP`

Relative tolerance employed by the linear dependency checker.

Default
1.0e-10

Accepted
[0.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_PRESOLVE_TOL_REL_LINDEP, 1.0e-10)
```

Groups
Presolve

`MSK_DPAR_PRESOLVE_TOL_S`

Absolute zero tolerance employed for s_i in the presolve.

Default
1.0e-8

Accepted
[0.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_PRESOLVE_TOL_S, 1.0e-8)
```

Groups
Presolve

`MSK_DPAR_PRESOLVE_TOL_X`

Absolute zero tolerance employed for x_j in the presolve.

Default
1.0e-8

Accepted
[0.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_PRESOLVE_TOL_X, 1.0e-8)
```

Groups
Presolve

`MSK_DPAR_QCQO_REFORMULATE_REL_DROP_TOL`

This parameter determines when columns are dropped in incomplete Cholesky factorization during reformulation of quadratic problems.

Default
1e-15

Accepted
[0; +inf]

Example
```
putdouparam(task, MSK_DPAR_QCQO_REFORMULATE_REL_DROP_TOL, 1e-15)
```

Groups
Interior-point method

`MSK_DPAR_SEMIDEFINITE_TOL_APPROX`

Tolerance to define a matrix to be positive semidefinite.

Default
1.0e-10

Accepted
[1.0e-15; +inf]

Example
```
putdouparam(task, MSK_DPAR_SEMIDEFINITE_TOL_APPROX, 1.0e-10)
```

Groups
Data check

`MSK_DPAR_SIM_LU_TOL_REL_PIV`

Relative pivot tolerance employed when computing the LU factorization of the basis in the simplex optimizers and in the basis identification procedure. A value closer to 1.0 generally improves numerical stability but typically also implies an increase in the computational work.

Default
0.01

Accepted
[1.0e-6; 0.999999]

Example
```
putdouparam(task, MSK_DPAR_SIM_LU_TOL_REL_PIV, 0.01)
```

Groups
Basis identification, *Simplex optimizer*

`MSK_DPAR_SIM_PRECISION_SCALING_EXTENDED`

Experimental. Usage not recommended.

Default
2.0

Accepted
[1.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_SIM_PRECISION_SCALING_EXTENDED, 2.0)
```

Groups
Simplex optimizer, *Termination criteria*

`MSK_DPAR_SIM_PRECISION_SCALING_NORMAL`

Experimental. Usage not recommended.

Default
1.0

Accepted
[1.0; +inf]

Example
```
putdouparam(task, MSK_DPAR_SIM_PRECISION_SCALING_NORMAL, 1.0)
```

Groups
Simplex optimizer, *Termination criteria*

`MSK_DPAR_SIMPLEX_ABS_TOL_PIV`

Absolute pivot tolerance employed by the simplex optimizers.

Default
1.0e-7

Accepted
[1.0e-12; +inf]

Example
`putdouparam(task, MSK_DPAR_SIMPLEX_ABS_TOL_PIV, 1.0e-7)`

Groups
Simplex optimizer

`MSK_DPAR_UPPER_OBJ_CUT`

If either a primal or dual feasible solution is found proving that the optimal objective value is outside the interval [*MSK_DPAR_LOWER_OBJ_CUT*, *MSK_DPAR_UPPER_OBJ_CUT*], then **MOSEK** is terminated.

Default
INFINITY

Accepted
[-inf; +inf]

Example
`putdouparam(task, MSK_DPAR_UPPER_OBJ_CUT, INFINITY)`

See also
MSK_DPAR_UPPER_OBJ_CUT_FINITE_TRH

Groups
Termination criteria

`MSK_DPAR_UPPER_OBJ_CUT_FINITE_TRH`

If the upper objective cut is greater than the value of this parameter, then the upper objective cut *MSK_DPAR_UPPER_OBJ_CUT* is treated as ∞.

Default
0.5e30

Accepted
[-inf; +inf]

Example
`putdouparam(task, MSK_DPAR_UPPER_OBJ_CUT_FINITE_TRH, 0.5e30)`

Groups
Termination criteria

15.5.2 Integer parameters

`iparam`

The enumeration type containing all integer parameters.

`MSK_IPAR_ANA_SOL_BASIS`

Controls whether the basis matrix is analyzed in solution analyzer.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
`putintparam(task, MSK_IPAR_ANA_SOL_BASIS, MSK_ON)`

Groups
Analysis

`MSK_IPAR_ANA_SOL_PRINT_VIOLATED`

A parameter of the problem analyzer. Controls whether a list of violated constraints is printed. All constraints violated by more than the value set by the parameter *MSK_DPAR_ANA_SOL_INFEAS_TOL* will be printed.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_ANA_SOL_PRINT_VIOLATED, MSK_OFF)
```

Groups
Analysis

`MSK_IPAR_AUTO_SORT_A_BEFORE_OPT`

Controls whether the elements in each column of A are sorted before an optimization is performed. This is not required but makes the optimization more deterministic.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_AUTO_SORT_A_BEFORE_OPT, MSK_OFF)
```

Groups
Debugging

`MSK_IPAR_AUTO_UPDATE_SOL_INFO`

Controls whether the solution information items are automatically updated after an optimization is performed.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_AUTO_UPDATE_SOL_INFO, MSK_OFF)
```

Groups
Overall system

`MSK_IPAR_BASIS_SOLVE_USE_PLUS_ONE`

If a slack variable is in the basis, then the corresponding column in the basis is a unit vector with -1 in the right position. However, if this parameter is set to *MSK_ON*, -1 is replaced by 1.

This has significance for the results returned by the *solvewithbasis* function.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_BASIS_SOLVE_USE_PLUS_ONE, MSK_OFF)
```

Groups
Simplex optimizer

`MSK_IPAR_BI_CLEAN_OPTIMIZER`

Controls which simplex optimizer is used in the clean-up phase. Anything else than *MSK_OPTIMIZER_PRIMAL_SIMPLEX* or *MSK_OPTIMIZER_DUAL_SIMPLEX* is equivalent to *MSK_OPTIMIZER_FREE_SIMPLEX*.

Default
FREE

Accepted
FREE, *INTPNT*, *CONIC*, *PRIMAL_SIMPLEX*, *DUAL_SIMPLEX*, *NEW_PRIMAL_SIMPLEX*, *NEW_DUAL_SIMPLEX*, *FREE_SIMPLEX*, *MIXED_INT* (see *optimizertype*)

Example

```
putintparam(task, MSK_IPAR_BI_CLEAN_OPTIMIZER, MSK_OPTIMIZER_FREE)
```

Groups

Basis identification, *Overall solver*

`MSK_IPAR_BI_IGNORE_MAX_ITER`

If the parameter *MSK_IPAR_INTPNT_BASIS* has the value *MSK_BI_NO_ERROR* and the interior-point optimizer has terminated due to maximum number of iterations, then basis identification is performed if this parameter has the value *MSK_ON*.

Default

OFF

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_BI_IGNORE_MAX_ITER, MSK_OFF)
```

Groups

Interior-point method, *Basis identification*

`MSK_IPAR_BI_IGNORE_NUM_ERROR`

If the parameter *MSK_IPAR_INTPNT_BASIS* has the value *MSK_BI_NO_ERROR* and the interior-point optimizer has terminated due to a numerical problem, then basis identification is performed if this parameter has the value *MSK_ON*.

Default

OFF

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_BI_IGNORE_NUM_ERROR, MSK_OFF)
```

Groups

Interior-point method, *Basis identification*

`MSK_IPAR_BI_MAX_ITERATIONS`

Controls the maximum number of simplex iterations allowed to optimize a basis after the basis identification.

Default

1000000

Accepted

[0; +inf]

Example

```
putintparam(task, MSK_IPAR_BI_MAX_ITERATIONS, 1000000)
```

Groups

Basis identification, *Termination criteria*

`MSK_IPAR_CACHE_LICENSE`

Specifies if the license is kept checked out for the lifetime of the **MOSEK** environment/model/process (*MSK_ON*) or returned to the server immediately after the optimization (*MSK_OFF*).

By default the license is checked out for the lifetime of the **MOSEK** environment by the first call to *optimize*.

Check-in and check-out of licenses have an overhead. Frequent communication with the license server should be avoided.

Default

ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_CACHE_LICENSE, MSK_ON)
```

Groups
License manager

`MSK_IPAR_COMPRESS_STATFILE`

Control compression of stat files.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_COMPRESS_STATFILE, MSK_ON)
```

`MSK_IPAR_FOLDING_USE`

Controls whether and how to use problem folding (symmetry detection for continuous problems). Note that for symmetry detection for mixed-integer problems one should instead use the parameter *MSK_IPAR_MIO_SYMMETRY_LEVEL*.

Default
FREE_UNLESS_BASIC

Accepted
OFF, *FREE*, *FREE_UNLESS_BASIC*, *FORCE* (see *foldingmode*)

Example
```
putintparam(task, MSK_IPAR_FOLDING_USE, MSK_FOLDING_MODE_FREE_UNLESS_BASIC)
```

Groups
Presolve

`MSK_IPAR_GETDUAL_CONVERT_LMIS`

Whether to perform LMI detection and optimization in the user-level dualizer.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_GETDUAL_CONVERT_LMIS, MSK_ON)
```

`MSK_IPAR_HEARTBEAT_SIM_FREQ_TICKS`

Controls how frequent the new simplex optimizer calls the user-defined callback function is called.

- -1. Logging is disabled.
- 0. Logging at highest frequency (every iteration).
- ≥ 1. Logging at given frequency measured in ticks.

Default
1000000

Accepted
[-1; +inf]

Example
```
putintparam(task, MSK_IPAR_HEARTBEAT_SIM_FREQ_TICKS, 1000000)
```

Groups
Simplex optimizer, *Output information*, *Logging*

MSK_IPAR_INFEAS_GENERIC_NAMES

Controls whether generic names are used when an infeasible subproblem is created.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_INFEAS_GENERIC_NAMES, MSK_OFF)
```

Groups
Infeasibility report

MSK_IPAR_INFEAS_REPORT_AUTO

Controls whether an infeasibility report is automatically produced after the optimization if the problem is primal or dual infeasible.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_INFEAS_REPORT_AUTO, MSK_OFF)
```

Groups
Data input/output, *Solution input/output*

MSK_IPAR_INFEAS_REPORT_LEVEL

Controls the amount of information presented in an infeasibility report. Higher values imply more information.

Default
1

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_INFEAS_REPORT_LEVEL, 1)
```

Groups
Infeasibility report, *Output information*

MSK_IPAR_INTPNT_BASIS

Controls whether the interior-point optimizer also computes an optimal basis.

Default
ALWAYS

Accepted
NEVER, *ALWAYS*, *NO_ERROR*, *IF_FEASIBLE*, *RESERVERED* (see *basindtype*)

Example
```
putintparam(task, MSK_IPAR_INTPNT_BASIS, MSK_BI_ALWAYS)
```

See also
MSK_IPAR_BI_IGNORE_MAX_ITER, *MSK_IPAR_BI_IGNORE_NUM_ERROR*, *MSK_IPAR_BI_MAX_ITERATIONS*, *MSK_IPAR_BI_CLEAN_OPTIMIZER*

Groups
Interior-point method, *Basis identification*

MSK_IPAR_INTPNT_DIFF_STEP

Controls whether different step sizes are allowed in the primal and dual space.

Default
ON

Accepted

- *ON*: Different step sizes are allowed.
- *OFF*: Different step sizes are not allowed.

Example

```
putintparam(task, MSK_IPAR_INTPNT_DIFF_STEP, MSK_ON)
```

Groups

Interior-point method

`MSK_IPAR_INTPNT_HOTSTART`

Currently not in use.

Default

NONE

Accepted

NONE, *PRIMAL*, *DUAL*, *PRIMAL_DUAL* (see *intpnthotstart*)

Example

```
putintparam(task, MSK_IPAR_INTPNT_HOTSTART, MSK_INTPNT_HOTSTART_NONE)
```

Groups

Interior-point method

`MSK_IPAR_INTPNT_MAX_ITERATIONS`

Controls the maximum number of iterations allowed in the interior-point optimizer.

Default

400

Accepted

[0; +inf]

Example

```
putintparam(task, MSK_IPAR_INTPNT_MAX_ITERATIONS, 400)
```

Groups

Interior-point method, *Termination criteria*

`MSK_IPAR_INTPNT_MAX_NUM_COR`

Controls the maximum number of correctors allowed by the multiple corrector procedure. A negative value means that **MOSEK** is making the choice.

Default

-1

Accepted

[-1; +inf]

Example

```
putintparam(task, MSK_IPAR_INTPNT_MAX_NUM_COR, -1)
```

Groups

Interior-point method

`MSK_IPAR_INTPNT_OFF_COL_TRH`

Controls how many offending columns are detected in the Jacobian of the constraint matrix.

0	no detection
1	aggressive detection
> 1	higher values mean less aggressive detection

Default

40

Accepted

[0; +inf]

Example
`putintparam(task, MSK_IPAR_INTPNT_OFF_COL_TRH, 40)`

Groups
Interior-point method

`MSK_IPAR_INTPNT_ORDER_GP_NUM_SEEDS`

The GP ordering is dependent on a random seed. Therefore, trying several random seeds may lead to a better ordering. This parameter controls the number of random seeds tried.

A value of 0 means that MOSEK makes the choice.

Default
0

Accepted
[0; +inf]

Example
`putintparam(task, MSK_IPAR_INTPNT_ORDER_GP_NUM_SEEDS, 0)`

Groups
Interior-point method

`MSK_IPAR_INTPNT_ORDER_METHOD`

Controls the ordering strategy used by the interior-point optimizer when factorizing the Newton equation system.

Default
FREE

Accepted
FREE, *APPMINLOC*, *EXPERIMENTAL*, *TRY_GRAPHPAR*, *FORCE_GRAPHPAR*, *NONE* (see *orderingtype*)

Example
```
putintparam(task, MSK_IPAR_INTPNT_ORDER_METHOD,
MSK_ORDER_METHOD_FREE)
```

Groups
Interior-point method

`MSK_IPAR_INTPNT_REGULARIZATION_USE`

Controls whether regularization is allowed.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
`putintparam(task, MSK_IPAR_INTPNT_REGULARIZATION_USE, MSK_ON)`

Groups
Interior-point method

`MSK_IPAR_INTPNT_SCALING`

Controls how the problem is scaled before the interior-point optimizer is used.

Default
FREE

Accepted
FREE, *NONE* (see *scalingtype*)

Example
`putintparam(task, MSK_IPAR_INTPNT_SCALING, MSK_SCALING_FREE)`

Groups
Interior-point method

MSK_IPAR_INTPNT_SOLVE_FORM

Controls whether the primal or the dual problem is solved.

Default
FREE

Accepted
FREE, *PRIMAL*, *DUAL* (see *solveform*)

Example
```
putintparam(task, MSK_IPAR_INTPNT_SOLVE_FORM, MSK_SOLVE_FREE)
```

Groups
Interior-point method

MSK_IPAR_INTPNT_STARTING_POINT

Starting point used by the interior-point optimizer.

Default
FREE

Accepted
FREE, *GUESS*, *CONSTANT* (see *startpointtype*)

Example
```
putintparam(task, MSK_IPAR_INTPNT_STARTING_POINT,
MSK_STARTING_POINT_FREE)
```

Groups
Interior-point method

MSK_IPAR_LICENSE_DEBUG

This option is used to turn on debugging of the license manager.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_LICENSE_DEBUG, MSK_OFF)
```

Groups
License manager

MSK_IPAR_LICENSE_PAUSE_TIME

If *MSK_IPAR_LICENSE_WAIT* is *MSK_ON* and no license is available, then **MOSEK** sleeps a number of milliseconds between each check of whether a license has become free.

Default
100

Accepted
[0; 1000000]

Example
```
putintparam(task, MSK_IPAR_LICENSE_PAUSE_TIME, 100)
```

Groups
License manager

MSK_IPAR_LICENSE_SUPPRESS_EXPIRE_WRNS

Controls whether license features expire warnings are suppressed.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
`putintparam(task, MSK_IPAR_LICENSE_SUPPRESS_EXPIRE_WRNS, MSK_OFF)`

Groups
License manager, *Output information*

`MSK_IPAR_LICENSE_TRH_EXPIRY_WRN`

If a license feature expires in a numbers of days less than the value of this parameter then a warning will be issued.

Default
7

Accepted
[0; +inf]

Example
`putintparam(task, MSK_IPAR_LICENSE_TRH_EXPIRY_WRN, 7)`

Groups
License manager, *Output information*

`MSK_IPAR_LICENSE_WAIT`

If all licenses are in use **MOSEK** returns with an error code. However, by turning on this parameter **MOSEK** will wait for an available license.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
`putintparam(task, MSK_IPAR_LICENSE_WAIT, MSK_OFF)`

Groups
Overall solver, *Overall system*, *License manager*

`MSK_IPAR_LOG`

Controls the amount of log information. The value 0 implies that all log information is suppressed. A higher level implies that more information is logged.

Please note that if a task is employed to solve a sequence of optimization problems the value of this parameter is reduced by the value of *MSK_IPAR_LOG_CUT_SECOND_OPT* for the second and any subsequent optimizations.

Default
10

Accepted
[0; +inf]

Example
`putintparam(task, MSK_IPAR_LOG, 10)`

See also
MSK_IPAR_LOG_CUT_SECOND_OPT

Groups
Output information, *Logging*

`MSK_IPAR_LOG_ANA_PRO`

Controls amount of output from the problem analyzer.

Default
1

Accepted
[0; +inf]

Example
`putintparam(task, MSK_IPAR_LOG_ANA_PRO, 1)`

Groups
Analysis, *Logging*

`MSK_IPAR_LOG_BI`

Controls the amount of output printed by the basis identification procedure. A higher level implies that more information is logged.

Default
1
Accepted
[0; +inf]
Example
```
putintparam(task, MSK_IPAR_LOG_BI, 1)
```
Groups
Basis identification, *Output information*, *Logging*

`MSK_IPAR_LOG_BI_FREQ`

Controls how frequently the optimizer outputs information about the basis identification and how frequent the user-defined callback function is called.

Default
2500
Accepted
[0; +inf]
Example
```
putintparam(task, MSK_IPAR_LOG_BI_FREQ, 2500)
```
Groups
Basis identification, *Output information*, *Logging*

`MSK_IPAR_LOG_CUT_SECOND_OPT`

If a task is employed to solve a sequence of optimization problems, then the value of the log levels is reduced by the value of this parameter. E.g *MSK_IPAR_LOG* and *MSK_IPAR_LOG_SIM* are reduced by the value of this parameter for the second and any subsequent optimizations.

Default
1
Accepted
[0; +inf]
Example
```
putintparam(task, MSK_IPAR_LOG_CUT_SECOND_OPT, 1)
```
See also
MSK_IPAR_LOG, *MSK_IPAR_LOG_INTPNT*, *MSK_IPAR_LOG_MIO*, *MSK_IPAR_LOG_SIM*
Groups
Output information, *Logging*

`MSK_IPAR_LOG_EXPAND`

Controls the amount of logging when a data item such as the maximum number constrains is expanded.

Default
1
Accepted
[0; +inf]
Example
```
putintparam(task, MSK_IPAR_LOG_EXPAND, 1)
```
Groups
Output information, *Logging*

MSK_IPAR_LOG_FEAS_REPAIR

Controls the amount of output printed when performing feasibility repair. A value higher than one means extensive logging.

Default
1

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_LOG_FEAS_REPAIR, 1)
```

Groups
Output information, *Logging*

MSK_IPAR_LOG_FILE

If turned on, then some log info is printed when a file is written or read.

Default
1

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_LOG_FILE, 1)
```

Groups
Data input/output, *Output information*, *Logging*

MSK_IPAR_LOG_INCLUDE_SUMMARY

If on, then the solution summary will be printed by *optimize*, so a separate call to *solutionsummary* is not necessary.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_LOG_INCLUDE_SUMMARY, MSK_OFF)
```

Groups
Output information, *Logging*

MSK_IPAR_LOG_INFEAS_ANA

Controls amount of output printed by the infeasibility analyzer procedures. A higher level implies that more information is logged.

Default
1

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_LOG_INFEAS_ANA, 1)
```

Groups
Infeasibility report, *Output information*, *Logging*

MSK_IPAR_LOG_INTPNT

Controls amount of output printed by the interior-point optimizer. A higher level implies that more information is logged.

Default
1

Accepted
[0; +inf]

Example
`putintparam(task, MSK_IPAR_LOG_INTPNT, 1)`

Groups
Interior-point method, *Output information*, *Logging*

`MSK_IPAR_LOG_LOCAL_INFO`

Controls whether local identifying information like environment variables, filenames, IP addresses etc. are printed to the log.

Note that this will only affect some functions. Some functions that specifically emit system information will not be affected.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
`putintparam(task, MSK_IPAR_LOG_LOCAL_INFO, MSK_ON)`

Groups
Output information, *Logging*

`MSK_IPAR_LOG_MIO`

Controls the log level for the mixed-integer optimizer. A higher level implies that more information is logged.

Default
4

Accepted
[0; +inf]

Example
`putintparam(task, MSK_IPAR_LOG_MIO, 4)`

Groups
Mixed-integer optimization, *Output information*, *Logging*

`MSK_IPAR_LOG_MIO_FREQ`

Controls how frequent the mixed-integer optimizer prints the log line. It will print line every time *MSK_IPAR_LOG_MIO_FREQ* relaxations have been solved.

Default
10

Accepted
[-inf; +inf]

Example
`putintparam(task, MSK_IPAR_LOG_MIO_FREQ, 10)`

Groups
Mixed-integer optimization, *Output information*, *Logging*

`MSK_IPAR_LOG_ORDER`

If turned on, then factor lines are added to the log.

Default
1

Accepted
[0; +inf]

Example
`putintparam(task, MSK_IPAR_LOG_ORDER, 1)`

Groups
Output information, *Logging*

`MSK_IPAR_LOG_PRESOLVE`

Controls amount of output printed by the presolve procedure. A higher level implies that more information is logged.

Default
1

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_LOG_PRESOLVE, 1)
```

Groups
Logging

`MSK_IPAR_LOG_SENSITIVITY`

Controls the amount of logging during the sensitivity analysis.

- 0. Means no logging information is produced.
- 1. Timing information is printed.
- 2. Sensitivity results are printed.

Default
1

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_LOG_SENSITIVITY, 1)
```

Groups
Output information, *Logging*

`MSK_IPAR_LOG_SENSITIVITY_OPT`

Controls the amount of logging from the optimizers employed during the sensitivity analysis. 0 means no logging information is produced.

Default
0

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_LOG_SENSITIVITY_OPT, 0)
```

Groups
Output information, *Logging*

`MSK_IPAR_LOG_SIM`

Controls amount of output printed by the simplex optimizer. A higher level implies that more information is logged.

Default
4

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_LOG_SIM, 4)
```

Groups
Simplex optimizer, *Output information*, *Logging*

`MSK_IPAR_LOG_SIM_FREQ`

Controls how frequent the simplex optimizer outputs information about the optimization and how frequent the user-defined callback function is called.

Default
1000

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_LOG_SIM_FREQ, 1000)
```

Groups
Simplex optimizer, *Output information*, *Logging*

`MSK_IPAR_LOG_SIM_FREQ_GIGA_TICKS`

Controls how frequent the new simplex optimizer outputs information about the optimization and how frequent the user-defined callback function is called.

- -1. Logging is disabled.
- 0. Logging at highest frequency (every iteration).
- ≥ 1. Logging at given frequency measured in giga ticks.

Default
100

Accepted
[-1; +inf]

Example
```
putintparam(task, MSK_IPAR_LOG_SIM_FREQ_GIGA_TICKS, 100)
```

Groups
Simplex optimizer, *Output information*, *Logging*

`MSK_IPAR_LOG_STORAGE`

When turned on, **MOSEK** prints messages regarding the storage usage and allocation.

Default
0

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_LOG_STORAGE, 0)
```

Groups
Output information, *Overall system*, *Logging*

`MSK_IPAR_MAX_NUM_WARNINGS`

Each warning is shown a limited number of times controlled by this parameter. A negative value is identical to infinite number of times.

Default
10

Accepted
[-inf; +inf]

Example
```
putintparam(task, MSK_IPAR_MAX_NUM_WARNINGS, 10)
```

Groups
Output information

MSK_IPAR_MIO_BRANCH_DIR

Controls whether the mixed-integer optimizer is branching up or down by default.

Default
FREE

Accepted
FREE, *UP*, *DOWN*, *NEAR*, *FAR*, *ROOT_LP*, *GUIDED*, *PSEUDOCOST* (see *branchdir*)

Example
```
putintparam(task, MSK_IPAR_MIO_BRANCH_DIR, MSK_BRANCH_DIR_FREE)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_CONFLICT_ANALYSIS_LEVEL

Controls the amount of conflict analysis employed by the mixed-integer optimizer.

- -1. The optimizer chooses the level of conflict analysis employed
- 0. conflict analysis is disabled
- 1. A lower amount of conflict analysis is employed
- 2. A higher amount of conflict analysis is employed

Default
-1

Accepted
[-1; 2]

Example
```
putintparam(task, MSK_IPAR_MIO_CONFLICT_ANALYSIS_LEVEL, -1)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_CONIC_OUTER_APPROXIMATION

If this option is turned on outer approximation is used when solving relaxations of conic problems; otherwise interior point is used.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_MIO_CONIC_OUTER_APPROXIMATION, MSK_OFF)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_CONSTRUCT_SOL

If set to *MSK_ON* and all integer variables have been given a value for which a feasible mixed integer solution exists, then **MOSEK** generates an initial solution to the mixed integer problem by fixing all integer values and solving the remaining problem.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_MIO_CONSTRUCT_SOL, MSK_OFF)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_CROSSOVER_MAX_NODES

Controls the maximum number of nodes allowed in each call to the Crossover heuristic. The default value of -1 means that the value is determined automatically. A value of zero turns off the heuristic.

Default
-1

Accepted
[-1; +inf]

Example
```
putintparam(task, MSK_IPAR_MIO_CROSSOVER_MAX_NODES, -1)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_CUT_CLIQUE

Controls whether clique cuts should be generated.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_MIO_CUT_CLIQUE, MSK_ON)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_CUT_CMIR

Controls whether mixed integer rounding cuts should be generated.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_MIO_CUT_CMIR, MSK_ON)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_CUT_GMI

Controls whether GMI cuts should be generated.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_MIO_CUT_GMI, MSK_ON)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_CUT_IMPLIED_BOUND

Controls whether implied bound cuts should be generated.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_MIO_CUT_IMPLIED_BOUND, MSK_ON)
```

Groups
Mixed-integer optimization

`MSK_IPAR_MIO_CUT_KNAPSACK_COVER`

Controls whether knapsack cover cuts should be generated.

Default
ON
Accepted
ON, *OFF* (see *onoffkey*)
Example
```
putintparam(task, MSK_IPAR_MIO_CUT_KNAPSACK_COVER, MSK_ON)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_CUT_LIPRO`

Controls whether lift-and-project cuts should be generated.

Default
OFF
Accepted
ON, *OFF* (see *onoffkey*)
Example
```
putintparam(task, MSK_IPAR_MIO_CUT_LIPRO, MSK_OFF)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_CUT_SELECTION_LEVEL`

Controls how aggressively generated cuts are selected to be included in the relaxation.

- -1. The optimizer chooses the level of cut selection
- 0. Generated cuts less likely to be added to the relaxation
- 1. Cuts are more aggressively selected to be included in the relaxation

Default
-1
Accepted
[-1; +1]
Example
```
putintparam(task, MSK_IPAR_MIO_CUT_SELECTION_LEVEL, -1)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_DATA_PERMUTATION_METHOD`

Controls what problem data permutation method is appplied to mixed-integer problems.

Default
NONE
Accepted
NONE, *CYCLIC_SHIFT*, *RANDOM* (see *miodatapermmethod*)
Example
```
putintparam(task, MSK_IPAR_MIO_DATA_PERMUTATION_METHOD,
MSK_MIO_DATA_PERMUTATION_METHOD_NONE)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_DUAL_RAY_ANALYSIS_LEVEL`

Controls the amount of dual ray analysis employed by the mixed-integer optimizer.

- -1. The optimizer chooses the level of dual ray analysis employed
- 0. Dual ray analysis is disabled
- 1. A lower amount of dual ray analysis is employed
- 2. A higher amount of dual ray analysis is employed

Default
-1
Accepted
[-1; 2]
Example
```
putintparam(task, MSK_IPAR_MIO_DUAL_RAY_ANALYSIS_LEVEL, -1)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_FEASPUMP_LEVEL`

Controls the way the Feasibility Pump heuristic is employed by the mixed-integer optimizer.

- -1. The optimizer chooses how the Feasibility Pump is used
- 0. The Feasibility Pump is disabled
- 1. The Feasibility Pump is enabled with an effort to improve solution quality
- 2. The Feasibility Pump is enabled with an effort to reach feasibility early

Default
-1
Accepted
[-1; 2]
Example
```
putintparam(task, MSK_IPAR_MIO_FEASPUMP_LEVEL, -1)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_HEURISTIC_LEVEL`

Controls the heuristic employed by the mixed-integer optimizer to locate an initial good integer feasible solution. A value of zero means the heuristic is not used at all. A larger value than 0 means that a gradually more sophisticated heuristic is used which is computationally more expensive. A negative value implies that the optimizer chooses the heuristic. Normally a value around 3 to 5 should be optimal.

Default
-1
Accepted
[-inf; +inf]
Example
```
putintparam(task, MSK_IPAR_MIO_HEURISTIC_LEVEL, -1)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_INDEPENDENT_BLOCK_LEVEL`

Controls the way the mixed-integer optimizer tries to find and exploit a decomposition of the problem into independent blocks.

- -1. The optimizer chooses how independent-block structure is handled
- 0. No independent-block structure is detected
- 1. Independent-block structure may be exploited only in presolve

- 2. Independent-block structure may be exploited through a dedicated algorithm after the root node
- 3. Independent-block structure may be exploited through a dedicated algorithm before the root node

Default
-1
Accepted
[-1; 3]
Example
```
putintparam(task, MSK_IPAR_MIO_INDEPENDENT_BLOCK_LEVEL, -1)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_MAX_NUM_BRANCHES`

Maximum number of branches allowed during the branch and bound search. A negative value means infinite.

Default
-1
Accepted
[-inf; +inf]
Example
```
putintparam(task, MSK_IPAR_MIO_MAX_NUM_BRANCHES, -1)
```
Groups
Mixed-integer optimization, *Termination criteria*

`MSK_IPAR_MIO_MAX_NUM_RELAXS`

Maximum number of relaxations allowed during the branch and bound search. A negative value means infinite.

Default
-1
Accepted
[-inf; +inf]
Example
```
putintparam(task, MSK_IPAR_MIO_MAX_NUM_RELAXS, -1)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_MAX_NUM_RESTARTS`

Maximum number of restarts allowed during the branch and bound search.

Default
10
Accepted
[0; +inf]
Example
```
putintparam(task, MSK_IPAR_MIO_MAX_NUM_RESTARTS, 10)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_MAX_NUM_ROOT_CUT_ROUNDS`

Maximum number of cut separation rounds at the root node.

Default
100

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_MIO_MAX_NUM_ROOT_CUT_ROUNDS, 100)
```

Groups
Mixed-integer optimization, *Termination criteria*

`MSK_IPAR_MIO_MAX_NUM_SOLUTIONS`

The mixed-integer optimizer can be terminated after a certain number of different feasible solutions has been located. If this parameter has the value $n > 0$, then the mixed-integer optimizer will be terminated when n feasible solutions have been located.

Default
-1

Accepted
[-inf; +inf]

Example
```
putintparam(task, MSK_IPAR_MIO_MAX_NUM_SOLUTIONS, -1)
```

Groups
Mixed-integer optimization, *Termination criteria*

`MSK_IPAR_MIO_MEMORY_EMPHASIS_LEVEL`

Controls how much emphasis is put on reducing memory usage. Being more conservative about memory usage may come at the cost of decreased solution speed.

- 0. The optimizer chooses
- 1. More emphasis is put on reducing memory usage and less on speed

Default
0

Accepted
[0; +1]

Example
```
putintparam(task, MSK_IPAR_MIO_MEMORY_EMPHASIS_LEVEL, 0)
```

Groups
Mixed-integer optimization

`MSK_IPAR_MIO_MIN_REL`

Number of times a variable must have been branched on for its pseudocost to be considered reliable.

Default
5

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_MIO_MIN_REL, 5)
```

Groups
Mixed-integer optimization

`MSK_IPAR_MIO_MODE`

Controls whether the optimizer includes the integer restrictions and disjunctive constraints when solving a (mixed) integer optimization problem.

Default
SATISFIED

Accepted
IGNORED, *SATISFIED* (see *miomode*)

Example

```
putintparam(task, MSK_IPAR_MIO_MODE, MSK_MIO_MODE_SATISFIED)
```

Groups

Overall solver

`MSK_IPAR_MIO_NODE_OPTIMIZER`

Controls which optimizer is employed at the non-root nodes in the mixed-integer optimizer.

Default

FREE

Accepted

FREE, *INTPNT*, *CONIC*, *PRIMAL_SIMPLEX*, *DUAL_SIMPLEX*, *NEW_PRIMAL_SIMPLEX*, *NEW_DUAL_SIMPLEX*, *FREE_SIMPLEX*, *MIXED_INT* (see *optimizertype*)

Example

```
putintparam(task, MSK_IPAR_MIO_NODE_OPTIMIZER, MSK_OPTIMIZER_FREE)
```

Groups

Mixed-integer optimization

`MSK_IPAR_MIO_NODE_SELECTION`

Controls the node selection strategy employed by the mixed-integer optimizer.

Default

FREE

Accepted

FREE, *FIRST*, *BEST*, *PSEUDO* (see *mionodeseltype*)

Example

```
putintparam(task, MSK_IPAR_MIO_NODE_SELECTION,
MSK_MIO_NODE_SELECTION_FREE)
```

Groups

Mixed-integer optimization

`MSK_IPAR_MIO_NUMERICAL_EMPHASIS_LEVEL`

Controls how much emphasis is put on reducing numerical problems possibly at the expense of solution speed.

- 0. The optimizer chooses
- 1. More emphasis is put on reducing numerical problems
- 2. Even more emphasis

Default

0

Accepted

[0; +2]

Example

```
putintparam(task, MSK_IPAR_MIO_NUMERICAL_EMPHASIS_LEVEL, 0)
```

Groups

Mixed-integer optimization

`MSK_IPAR_MIO_OPT_FACE_MAX_NODES`

Controls the maximum number of nodes allowed in each call to the optimal face heuristic. The default value of -1 means that the value is determined automatically. A value of zero turns off the heuristic.

Default

-1

Accepted

[-1; +inf]

Example

```
putintparam(task, MSK_IPAR_MIO_OPT_FACE_MAX_NODES, -1)
```

Groups

Mixed-integer optimization

`MSK_IPAR_MIO_PERSPECTIVE_REFORMULATE`

Enables or disables perspective reformulation in presolve.

Default

ON

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_MIO_PERSPECTIVE_REFORMULATE, MSK_ON)
```

Groups

Mixed-integer optimization

`MSK_IPAR_MIO_PRESOLVE_AGGREGATOR_USE`

Controls if the aggregator should be used.

Default

ON

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_MIO_PRESOLVE_AGGREGATOR_USE, MSK_ON)
```

Groups

Presolve

`MSK_IPAR_MIO_PROBING_LEVEL`

Controls the amount of probing employed by the mixed-integer optimizer in presolve.

- -1. The optimizer chooses the level of probing employed
- 0. Probing is disabled
- 1. A low amount of probing is employed
- 2. A medium amount of probing is employed
- 3. A high amount of probing is employed

Default

-1

Accepted

[-1; 3]

Example

```
putintparam(task, MSK_IPAR_MIO_PROBING_LEVEL, -1)
```

Groups

Mixed-integer optimization

`MSK_IPAR_MIO_PROPAGATE_OBJECTIVE_CONSTRAINT`

Use objective domain propagation.

Default

OFF

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_MIO_PROPAGATE_OBJECTIVE_CONSTRAINT,
MSK_OFF)
```

Groups

Mixed-integer optimization

MSK_IPAR_MIO_QCQO_REFORMULATION_METHOD

Controls what reformulation method is applied to mixed-integer quadratic problems.

Default
FREE

Accepted
FREE, *NONE*, *LINEARIZATION*, *EIGEN_VAL_METHOD*, *DIAG_SDP*, *RELAX_SDP* (see *miqcqoreformmethod*)

Example
```
putintparam(task, MSK_IPAR_MIO_QCQO_REFORMULATION_METHOD,
MSK_MIO_QCQO_REFORMULATION_METHOD_FREE)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_RENS_MAX_NODES

Controls the maximum number of nodes allowed in each call to the RENS heuristic. The default value of -1 means that the value is determined automatically. A value of zero turns off the heuristic.

Default
-1

Accepted
[-1; +inf]

Example
```
putintparam(task, MSK_IPAR_MIO_RENS_MAX_NODES, -1)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_RINS_MAX_NODES

Controls the maximum number of nodes allowed in each call to the RINS heuristic. The default value of -1 means that the value is determined automatically. A value of zero turns off the heuristic.

Default
-1

Accepted
[-1; +inf]

Example
```
putintparam(task, MSK_IPAR_MIO_RINS_MAX_NODES, -1)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_ROOT_OPTIMIZER

Controls which optimizer is employed at the root node in the mixed-integer optimizer.

Default
FREE

Accepted
FREE, *INTPNT*, *CONIC*, *PRIMAL_SIMPLEX*, *DUAL_SIMPLEX*, *NEW_PRIMAL_SIMPLEX*, *NEW_DUAL_SIMPLEX*, *FREE_SIMPLEX*, *MIXED_INT* (see *optimizertype*)

Example
```
putintparam(task, MSK_IPAR_MIO_ROOT_OPTIMIZER, MSK_OPTIMIZER_FREE)
```

Groups
Mixed-integer optimization

MSK_IPAR_MIO_SEED

Sets the random seed used for randomization in the mixed integer optimizer. Selecting a different seed can change the path the optimizer takes to the optimal solution.

Default
42

Accepted
[0; +inf]
Example
```
putintparam(task, MSK_IPAR_MIO_SEED, 42)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_SYMMETRY_LEVEL`

Controls the amount of symmetry detection and handling employed by the mixed-integer optimizer in presolve.

- −1. The optimizer chooses the level of symmetry detection and handling employed
- 0. Symmetry detection and handling is disabled
- 1. A low amount of symmetry detection and handling is employed
- 2. A medium amount of symmetry detection and handling is employed
- 3. A high amount of symmetry detection and handling is employed
- 4. An extremely high amount of symmetry detection and handling is employed

Default
-1
Accepted
[-1; 4]
Example
```
putintparam(task, MSK_IPAR_MIO_SYMMETRY_LEVEL, -1)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_VAR_SELECTION`

Controls the variable selection strategy employed by the mixed-integer optimizer.

Default
FREE
Accepted
FREE, *PSEUDOCOST*, *STRONG* (see *miovarseltype*)
Example
```
putintparam(task, MSK_IPAR_MIO_VAR_SELECTION,
MSK_MIO_VAR_SELECTION_FREE)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MIO_VB_DETECTION_LEVEL`

Controls how much effort is put into detecting variable bounds.

- −1. The optimizer chooses
- 0. No variable bounds are detected
- 1. Only detect variable bounds that are directly represented in the problem
- 2. Detect variable bounds in probing

Default
-1
Accepted
[-1; +2]
Example
```
putintparam(task, MSK_IPAR_MIO_VB_DETECTION_LEVEL, -1)
```
Groups
Mixed-integer optimization

`MSK_IPAR_MT_SPINCOUNT`

Set the number of iterations to spin before sleeping.

Default
0

Accepted
[0; 1000000000]

Example
```
putintparam(task, MSK_IPAR_MT_SPINCOUNT, 0)
```

Groups
Overall system

`MSK_IPAR_NG`

Not in use.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_NG, MSK_OFF)
```

`MSK_IPAR_NUM_THREADS`

Controls the number of threads employed by the optimizer. If set to 0 then the number of threads is chosen by the optimizer, typically as minimum(number of cores, 32). If set to a positive value then exactly the selected number of threads will be used.

Default
0

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_NUM_THREADS, 0)
```

Groups
Overall system

`MSK_IPAR_OPF_WRITE_HEADER`

Write a text header with date and **MOSEK** version in an OPF file.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_OPF_WRITE_HEADER, MSK_ON)
```

Groups
Data input/output

`MSK_IPAR_OPF_WRITE_HINTS`

Write a hint section with problem dimensions in the beginning of an OPF file.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_OPF_WRITE_HINTS, MSK_ON)
```

Groups
Data input/output

MSK_IPAR_OPF_WRITE_LINE_LENGTH

Aim to keep lines in OPF files not much longer than this.

Default
80

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_OPF_WRITE_LINE_LENGTH, 80)
```

Groups
Data input/output

MSK_IPAR_OPF_WRITE_PARAMETERS

Write a parameter section in an OPF file.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_OPF_WRITE_PARAMETERS, MSK_OFF)
```

Groups
Data input/output

MSK_IPAR_OPF_WRITE_PROBLEM

Write objective, constraints, bounds etc. to an OPF file.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_OPF_WRITE_PROBLEM, MSK_ON)
```

Groups
Data input/output

MSK_IPAR_OPF_WRITE_SOL_BAS

If *MSK_IPAR_OPF_WRITE_SOLUTIONS* is *MSK_ON* and a basic solution is defined, include the basic solution in OPF files.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_OPF_WRITE_SOL_BAS, MSK_ON)
```

Groups
Data input/output

MSK_IPAR_OPF_WRITE_SOL_ITG

If *MSK_IPAR_OPF_WRITE_SOLUTIONS* is *MSK_ON* and an integer solution is defined, write the integer solution in OPF files.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
putintparam(task, MSK_IPAR_OPF_WRITE_SOL_ITG, MSK_ON)

Groups
Data input/output

MSK_IPAR_OPF_WRITE_SOL_ITR

If *MSK_IPAR_OPF_WRITE_SOLUTIONS* is *MSK_ON* and an interior solution is defined, write the interior solution in OPF files.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
putintparam(task, MSK_IPAR_OPF_WRITE_SOL_ITR, MSK_ON)

Groups
Data input/output

MSK_IPAR_OPF_WRITE_SOLUTIONS

Enable inclusion of solutions in the OPF files.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
putintparam(task, MSK_IPAR_OPF_WRITE_SOLUTIONS, MSK_OFF)

Groups
Data input/output

MSK_IPAR_OPTIMIZER

The parameter controls which optimizer is used to optimize the task.

Default
FREE

Accepted
FREE, *INTPNT*, *CONIC*, *PRIMAL_SIMPLEX*, *DUAL_SIMPLEX*, *NEW_PRIMAL_SIMPLEX*, *NEW_DUAL_SIMPLEX*, *FREE_SIMPLEX*, *MIXED_INT* (see *optimizertype*)

Example
putintparam(task, MSK_IPAR_OPTIMIZER, MSK_OPTIMIZER_FREE)

Groups
Overall solver

MSK_IPAR_PARAM_READ_CASE_NAME

If turned on, then names in the parameter file are case sensitive.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
putintparam(task, MSK_IPAR_PARAM_READ_CASE_NAME, MSK_ON)

Groups
Data input/output

`MSK_IPAR_PARAM_READ_IGN_ERROR`

If turned on, then errors in parameter settings is ignored.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_PARAM_READ_IGN_ERROR, MSK_OFF)
```

Groups
Data input/output

`MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_FILL`

Controls the maximum amount of fill-in that can be created by one pivot in the elimination phase of the presolve. A negative value means the parameter value is selected automatically.

Default
-1

Accepted
[-inf; +inf]

Example
```
putintparam(task, MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_FILL, -1)
```

Groups
Presolve

`MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_NUM_TRIES`

Control the maximum number of times the eliminator is tried. A negative value implies **MOSEK** decides.

Default
-1

Accepted
[-inf; +inf]

Example
```
putintparam(task, MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_NUM_TRIES, -1)
```

Groups
Presolve

`MSK_IPAR_PRESOLVE_LINDEP_ABS_WORK_TRH`

Controls linear dependency check in presolve. The linear dependency check is potentially computationally expensive.

Default
100

Accepted
[-inf; +inf]

Example
```
putintparam(task, MSK_IPAR_PRESOLVE_LINDEP_ABS_WORK_TRH, 100)
```

Groups
Presolve

`MSK_IPAR_PRESOLVE_LINDEP_NEW`

Controls whether a new experimental linear dependency checker is employed.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_PRESOLVE_LINDEP_NEW, MSK_OFF)
```

Groups

Presolve

`MSK_IPAR_PRESOLVE_LINDEP_REL_WORK_TRH`

Controls linear dependency check in presolve. The linear dependency check is potentially computationally expensive.

Default

100

Accepted

[-inf; +inf]

Example

```
putintparam(task, MSK_IPAR_PRESOLVE_LINDEP_REL_WORK_TRH, 100)
```

Groups

Presolve

`MSK_IPAR_PRESOLVE_LINDEP_USE`

Controls whether the linear constraints are checked for linear dependencies.

Default

ON

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_PRESOLVE_LINDEP_USE, MSK_ON)
```

Groups

Presolve

`MSK_IPAR_PRESOLVE_MAX_NUM_PASS`

Control the maximum number of times presolve passes over the problem. A negative value implies **MOSEK** decides.

Default

-1

Accepted

[-inf; +inf]

Example

```
putintparam(task, MSK_IPAR_PRESOLVE_MAX_NUM_PASS, -1)
```

Groups

Presolve

`MSK_IPAR_PRESOLVE_MAX_NUM_REDUCTIONS`

Controls the maximum number of reductions performed by the presolve. The value of the parameter is normally only changed in connection with debugging. A negative value implies that an infinite number of reductions are allowed.

Default

-1

Accepted

[-inf; +inf]

Example

```
putintparam(task, MSK_IPAR_PRESOLVE_MAX_NUM_REDUCTIONS, -1)
```

Groups

Overall solver, *Presolve*

MSK_IPAR_PRESOLVE_USE

Controls whether the presolve is applied to a problem before it is optimized.

Default
FREE

Accepted
OFF, *ON*, *FREE* (see *presolvemode*)

Example
```
putintparam(task, MSK_IPAR_PRESOLVE_USE, MSK_PRESOLVE_MODE_FREE)
```

Groups
Overall solver, *Presolve*

MSK_IPAR_PRIMAL_REPAIR_OPTIMIZER

Controls which optimizer that is used to find the optimal repair.

Default
FREE

Accepted
FREE, *INTPNT*, *CONIC*, *PRIMAL_SIMPLEX*, *DUAL_SIMPLEX*, *NEW_PRIMAL_SIMPLEX*, *NEW_DUAL_SIMPLEX*, *FREE_SIMPLEX*, *MIXED_INT* (see *optimizertype*)

Example
```
putintparam(task, MSK_IPAR_PRIMAL_REPAIR_OPTIMIZER,
MSK_OPTIMIZER_FREE)
```

Groups
Overall solver

MSK_IPAR_PTF_WRITE_PARAMETERS

If *MSK_IPAR_PTF_WRITE_PARAMETERS* is *MSK_ON*, the parameters section is written.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_PTF_WRITE_PARAMETERS, MSK_OFF)
```

Groups
Data input/output

MSK_IPAR_PTF_WRITE_SINGLE_PSD_TERMS

Controls whether PSD terms with a coefficient matrix of just one non-zero are written as a single term instead of as a matrix term.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_PTF_WRITE_SINGLE_PSD_TERMS, MSK_OFF)
```

Groups
Data input/output

MSK_IPAR_PTF_WRITE_SOLUTIONS

If *MSK_IPAR_PTF_WRITE_SOLUTIONS* is *MSK_ON*, the solution section is written if any solutions are available, otherwise solution section is not written even if solutions are available.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_PTF_WRITE_SOLUTIONS, MSK_OFF)
```

Groups

Data input/output

`MSK_IPAR_PTF_WRITE_TRANSFORM`

If *MSK_IPAR_PTF_WRITE_TRANSFORM* is *MSK_ON*, constraint blocks with identifiable conic slacks are transformed into conic constraints and the slacks are eliminated.

Default

ON

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_PTF_WRITE_TRANSFORM, MSK_ON)
```

Groups

Data input/output

`MSK_IPAR_READ_ASYNC`

Controls whether files are read using synchronous or asynchronous reader.

Default

OFF

Accepted

- *ON*: Use asynchronous reader
- *OFF*: Use synchronous reader

Example

```
putintparam(task, MSK_IPAR_READ_ASYNC, MSK_OFF)
```

Groups

Data input/output

`MSK_IPAR_READ_DEBUG`

Turns on additional debugging information when reading files.

Default

OFF

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_READ_DEBUG, MSK_OFF)
```

Groups

Data input/output

`MSK_IPAR_READ_KEEP_FREE_CON`

Controls whether the free constraints are included in the problem. Applies to MPS files.

Default

OFF

Accepted

- *ON*: The free constraints are kept.
- *OFF*: The free constraints are discarded.

Example

```
putintparam(task, MSK_IPAR_READ_KEEP_FREE_CON, MSK_OFF)
```

Groups

Data input/output

MSK_IPAR_READ_MPS_FORMAT

Controls how strictly the MPS file reader interprets the MPS format.

Default
FREE

Accepted
STRICT, *RELAXED*, *FREE*, *CPLEX* (see *mpsformat*)

Example
```
putintparam(task, MSK_IPAR_READ_MPS_FORMAT, MSK_MPS_FORMAT_FREE)
```

Groups
Data input/output

MSK_IPAR_READ_MPS_WIDTH

Controls the maximal number of characters allowed in one line of the MPS file.

Default
1024

Accepted
[80; +inf]

Example
```
putintparam(task, MSK_IPAR_READ_MPS_WIDTH, 1024)
```

Groups
Data input/output

MSK_IPAR_READ_TASK_IGNORE_PARAM

Controls whether **MOSEK** should ignore the parameter setting defined in the task file and use the default parameter setting instead.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_READ_TASK_IGNORE_PARAM, MSK_OFF)
```

Groups
Data input/output

MSK_IPAR_REMOTE_USE_COMPRESSION

Use compression when sending data to an optimization server.

Default
ZSTD

Accepted
NONE, *FREE*, *GZIP*, *ZSTD* (see *compresstype*)

Example
```
putintparam(task, MSK_IPAR_REMOTE_USE_COMPRESSION, MSK_COMPRESS_ZSTD)
```

MSK_IPAR_REMOVE_UNUSED_SOLUTIONS

Removes unused solutions before the optimization is performed.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_REMOVE_UNUSED_SOLUTIONS, MSK_OFF)
```

Groups
Overall system

MSK_IPAR_SENSITIVITY_ALL

If set to *MSK_ON*, then *sensitivityreport* analyzes all bounds and variables instead of reading a specification from the file.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_SENSITIVITY_ALL, MSK_OFF)
```

Groups
Overall solver

MSK_IPAR_SENSITIVITY_TYPE

Controls which type of sensitivity analysis is to be performed.

Default
BASIS

Accepted
BASIS (see *sensitivitytype*)

Example
```
putintparam(task, MSK_IPAR_SENSITIVITY_TYPE, MSK_SENSITIVITY_TYPE_BASIS)
```

Groups
Overall solver

MSK_IPAR_SIM_BASIS_FACTOR_USE

Controls whether an LU factorization of the basis is used in a hot-start. Forcing a refactorization sometimes improves the stability of the simplex optimizers, but in most cases there is a performance penalty.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_SIM_BASIS_FACTOR_USE, MSK_ON)
```

Groups
Simplex optimizer

MSK_IPAR_SIM_DEGEN

Controls how aggressively degeneration is handled.

Default
FREE

Accepted
NONE, *FREE*, *AGGRESSIVE*, *MODERATE*, *MINIMUM* (see *simdegen*)

Example
```
putintparam(task, MSK_IPAR_SIM_DEGEN, MSK_SIM_DEGEN_FREE)
```

Groups
Simplex optimizer

MSK_IPAR_SIM_DETECT_PWL

Not in use.

Default
ON

Accepted
- *ON*: PWL are detected.

- *OFF*: PWL are not detected.

Example

```
putintparam(task, MSK_IPAR_SIM_DETECT_PWL, MSK_ON)
```

Groups

Simplex optimizer

`MSK_IPAR_SIM_DUAL_CRASH`

Controls whether crashing is performed in the dual simplex optimizer. If this parameter is set to x, then a crash will be performed if a basis consists of more than $(100 - x) \mod f_v$ entries, where f_v is the number of fixed variables.

Default

90

Accepted

[0; +inf]

Example

```
putintparam(task, MSK_IPAR_SIM_DUAL_CRASH, 90)
```

Groups

Dual simplex

`MSK_IPAR_SIM_DUAL_PHASEONE_METHOD`

An experimental feature.

Default

0

Accepted

[0; 10]

Example

```
putintparam(task, MSK_IPAR_SIM_DUAL_PHASEONE_METHOD, 0)
```

Groups

Simplex optimizer

`MSK_IPAR_SIM_DUAL_RESTRICT_SELECTION`

The dual simplex optimizer can use a so-called restricted selection/pricing strategy to choose the outgoing variable. Hence, if restricted selection is applied, then the dual simplex optimizer first choose a subset of all the potential outgoing variables. Next, for some time it will choose the outgoing variable only among the subset. From time to time the subset is redefined. A larger value of this parameter implies that the optimizer will be more aggressive in its restriction strategy, i.e. a value of 0 implies that the restriction strategy is not applied at all.

Default

50

Accepted

[0; 100]

Example

```
putintparam(task, MSK_IPAR_SIM_DUAL_RESTRICT_SELECTION, 50)
```

Groups

Dual simplex

`MSK_IPAR_SIM_DUAL_SELECTION`

Controls the choice of the incoming variable, known as the selection strategy, in the dual simplex optimizer.

Default

FREE

Accepted

FREE, *FULL*, *ASE*, *DEVEX*, *SE*, *PARTIAL* (see *simseltype*)

Example

```
putintparam(task, MSK_IPAR_SIM_DUAL_SELECTION,
MSK_SIM_SELECTION_FREE)
```

Groups

Dual simplex

`MSK_IPAR_SIM_EXPLOIT_DUPVEC`

Controls if the simplex optimizers are allowed to exploit duplicated columns.

Default

OFF

Accepted

ON, *OFF*, *FREE* (see *simdupvec*)

Example

```
putintparam(task, MSK_IPAR_SIM_EXPLOIT_DUPVEC,
MSK_SIM_EXPLOIT_DUPVEC_OFF)
```

Groups

Simplex optimizer

`MSK_IPAR_SIM_HOTSTART`

Controls the type of hot-start that the simplex optimizer perform.

Default

FREE

Accepted

NONE, *FREE*, *STATUS_KEYS* (see *simhotstart*)

Example

```
putintparam(task, MSK_IPAR_SIM_HOTSTART, MSK_SIM_HOTSTART_FREE)
```

Groups

Simplex optimizer

`MSK_IPAR_SIM_HOTSTART_LU`

Determines if the simplex optimizer should exploit the initial factorization.

Default

ON

Accepted

- *ON*: Factorization is reused if possible.
- *OFF*: Factorization is recomputed.

Example

```
putintparam(task, MSK_IPAR_SIM_HOTSTART_LU, MSK_ON)
```

Groups

Simplex optimizer

`MSK_IPAR_SIM_MAX_ITERATIONS`

Maximum number of iterations that can be used by a simplex optimizer.

Default

10000000

Accepted

[0; +inf]

Example

```
putintparam(task, MSK_IPAR_SIM_MAX_ITERATIONS, 10000000)
```

Groups

Simplex optimizer, *Termination criteria*

MSK_IPAR_SIM_MAX_NUM_SETBACKS

Controls how many set-backs are allowed within a simplex optimizer. A set-back is an event where the optimizer moves in the wrong direction. This is impossible in theory but may happen due to numerical problems.

Default
250

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_SIM_MAX_NUM_SETBACKS, 250)
```

Groups
Simplex optimizer

MSK_IPAR_SIM_NON_SINGULAR

Controls if the simplex optimizer ensures a non-singular basis, if possible.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_SIM_NON_SINGULAR, MSK_ON)
```

Groups
Simplex optimizer

MSK_IPAR_SIM_PRECISION

Experimental. Usage not recommended.

Default
NORMAL

Accepted
NORMAL, *EXTENDED* (see *simprecision*)

Example
```
putintparam(task, MSK_IPAR_SIM_PRECISION, MSK_SIM_PRECISION_NORMAL)
```

Groups
Overall solver

MSK_IPAR_SIM_PRECISION_BOOST

Controls whether the simplex optimizer is allowed to boost the precision during the computations if possible.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_SIM_PRECISION_BOOST, MSK_OFF)
```

Groups
Simplex optimizer

MSK_IPAR_SIM_PRIMAL_CRASH

Controls whether crashing is performed in the primal simplex optimizer. In general, if a basis consists of more than (100-this parameter value)% fixed variables, then a crash will be performed.

Default
90

Accepted
[0; +inf]

Example

```
putintparam(task, MSK_IPAR_SIM_PRIMAL_CRASH, 90)
```

Groups

Primal simplex

`MSK_IPAR_SIM_PRIMAL_PHASEONE_METHOD`

An experimental feature.

Default

0

Accepted

[0; 10]

Example

```
putintparam(task, MSK_IPAR_SIM_PRIMAL_PHASEONE_METHOD, 0)
```

Groups

Simplex optimizer

`MSK_IPAR_SIM_PRIMAL_RESTRICT_SELECTION`

The primal simplex optimizer can use a so-called restricted selection/pricing strategy to choose the outgoing variable. Hence, if restricted selection is applied, then the primal simplex optimizer first choose a subset of all the potential incoming variables. Next, for some time it will choose the incoming variable only among the subset. From time to time the subset is redefined. A larger value of this parameter implies that the optimizer will be more aggressive in its restriction strategy, i.e. a value of 0 implies that the restriction strategy is not applied at all.

Default

50

Accepted

[0; 100]

Example

```
putintparam(task, MSK_IPAR_SIM_PRIMAL_RESTRICT_SELECTION, 50)
```

Groups

Primal simplex

`MSK_IPAR_SIM_PRIMAL_SELECTION`

Controls the choice of the incoming variable, known as the selection strategy, in the primal simplex optimizer.

Default

FREE

Accepted

FREE, *FULL*, *ASE*, *DEVEX*, *SE*, *PARTIAL* (see *simseltype*)

Example

```
putintparam(task, MSK_IPAR_SIM_PRIMAL_SELECTION,
MSK_SIM_SELECTION_FREE)
```

Groups

Primal simplex

`MSK_IPAR_SIM_REFACTOR_FREQ`

Controls how frequent the basis is refactorized. The value 0 means that the optimizer determines the best point of refactorization. It is strongly recommended NOT to change this parameter.

Default

0

Accepted

[0; +inf]

Example

```
putintparam(task, MSK_IPAR_SIM_REFACTOR_FREQ, 0)
```

Groups
Simplex optimizer

MSK_IPAR_SIM_REFORMULATION

Controls if the simplex optimizers are allowed to reformulate the problem.

Default
OFF

Accepted
ON, *OFF*, *FREE*, *AGGRESSIVE* (see *simreform*)

Example
```
putintparam(task, MSK_IPAR_SIM_REFORMULATION,
MSK_SIM_REFORMULATION_OFF)
```

Groups
Simplex optimizer

MSK_IPAR_SIM_SAVE_LU

Controls if the LU factorization stored should be replaced with the LU factorization corresponding to the initial basis.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_SIM_SAVE_LU, MSK_OFF)
```

Groups
Simplex optimizer

MSK_IPAR_SIM_SCALING

Controls how much effort is used in scaling the problem before a simplex optimizer is used.

Default
FREE

Accepted
FREE, *NONE* (see *scalingtype*)

Example
```
putintparam(task, MSK_IPAR_SIM_SCALING, MSK_SCALING_FREE)
```

Groups
Simplex optimizer

MSK_IPAR_SIM_SCALING_METHOD

Controls how the problem is scaled before a simplex optimizer is used.

Default
POW2

Accepted
POW2, *FREE* (see *scalingmethod*)

Example
```
putintparam(task, MSK_IPAR_SIM_SCALING_METHOD,
MSK_SCALING_METHOD_POW2)
```

Groups
Simplex optimizer

MSK_IPAR_SIM_SEED

Sets the random seed used for randomization in the simplex optimizers.

Default
23456

Accepted
[0; 32749]

Example
```
putintparam(task, MSK_IPAR_SIM_SEED, 23456)
```

Groups
Simplex optimizer

`MSK_IPAR_SIM_SOLVE_FORM`

Controls whether the primal or the dual problem is solved by the primal-/dual-simplex optimizer.

Default
FREE

Accepted
FREE, *PRIMAL*, *DUAL* (see *solveform*)

Example
```
putintparam(task, MSK_IPAR_SIM_SOLVE_FORM, MSK_SOLVE_FREE)
```

Groups
Simplex optimizer

`MSK_IPAR_SIM_SWITCH_OPTIMIZER`

The simplex optimizer sometimes chooses to solve the dual problem instead of the primal problem. This implies that if you have chosen to use the dual simplex optimizer and the problem is dualized, then it actually makes sense to use the primal simplex optimizer instead. If this parameter is on and the problem is dualized and furthermore the simplex optimizer is chosen to be the primal (dual) one, then it is switched to the dual (primal).

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_SIM_SWITCH_OPTIMIZER, MSK_OFF)
```

Groups
Simplex optimizer

`MSK_IPAR_SOL_FILTER_KEEP_BASIC`

If turned on, then basic and super basic constraints and variables are written to the solution file independent of the filter setting.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_SOL_FILTER_KEEP_BASIC, MSK_OFF)
```

Groups
Solution input/output

`MSK_IPAR_SOL_READ_NAME_WIDTH`

When a solution is read by **MOSEK** and some constraint, variable or cone names contain blanks, then a maximum name width much be specified. A negative value implies that no name contain blanks.

Default
-1

Accepted
[-inf; +inf]

Example

```
putintparam(task, MSK_IPAR_SOL_READ_NAME_WIDTH, -1)
```

Groups

Data input/output, *Solution input/output*

`MSK_IPAR_SOL_READ_WIDTH`

Controls the maximal acceptable width of line in the solutions when read by **MOSEK**.

Default

1024

Accepted

[80; +inf]

Example

```
putintparam(task, MSK_IPAR_SOL_READ_WIDTH, 1024)
```

Groups

Data input/output, *Solution input/output*

`MSK_IPAR_TIMING_LEVEL`

Controls the amount of timing performed inside **MOSEK**.

Default

1

Accepted

[0; +inf]

Example

```
putintparam(task, MSK_IPAR_TIMING_LEVEL, 1)
```

Groups

Overall system

`MSK_IPAR_WRITE_ASYNC`

Controls whether files are read using synchronous or asynchronous writer.

Default

OFF

Accepted

- *ON*: Use asynchronous writer
- *OFF*: Use synchronous writer

Example

```
putintparam(task, MSK_IPAR_WRITE_ASYNC, MSK_OFF)
```

Groups

Data input/output

`MSK_IPAR_WRITE_BAS_CONSTRAINTS`

Controls whether the constraint section is written to the basic solution file.

Default

ON

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_WRITE_BAS_CONSTRAINTS, MSK_ON)
```

Groups

Data input/output, *Solution input/output*

`MSK_IPAR_WRITE_BAS_HEAD`

Controls whether the header section is written to the basic solution file.

Default

ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_WRITE_BAS_HEAD, MSK_ON)
```

Groups
Data input/output, *Solution input/output*

`MSK_IPAR_WRITE_BAS_VARIABLES`

Controls whether the variables section is written to the basic solution file.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_WRITE_BAS_VARIABLES, MSK_ON)
```

Groups
Data input/output, *Solution input/output*

`MSK_IPAR_WRITE_COMPRESSION`

Controls whether the data file is compressed while it is written. 0 means no compression while higher values mean more compression.

Default
9

Accepted
[0; +inf]

Example
```
putintparam(task, MSK_IPAR_WRITE_COMPRESSION, 9)
```

Groups
Data input/output

`MSK_IPAR_WRITE_FREE_CON`

Controls whether the free constraints are written to the data file. Applies to MPS files.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_WRITE_FREE_CON, MSK_ON)
```

Groups
Data input/output

`MSK_IPAR_WRITE_GENERIC_NAMES`

Controls whether generic names should be used instead of user-defined names when writing to the data file.

Default
OFF

Accepted
ON, *OFF* (see *onoffkey*)

Example
```
putintparam(task, MSK_IPAR_WRITE_GENERIC_NAMES, MSK_OFF)
```

Groups
Data input/output

MSK_IPAR_WRITE_IGNORE_INCOMPATIBLE_ITEMS

Controls if the writer ignores incompatible problem items when writing files.

Default

OFF

Accepted

- *ON*: Ignore items that cannot be written to the current output file format.
- *OFF*: Produce an error if the problem contains items that cannot the written to the current output file format.

Example

```
putintparam(task, MSK_IPAR_WRITE_IGNORE_INCOMPATIBLE_ITEMS, MSK_OFF)
```

Groups

Data input/output

MSK_IPAR_WRITE_INT_CONSTRAINTS

Controls whether the constraint section is written to the integer solution file.

Default

ON

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_WRITE_INT_CONSTRAINTS, MSK_ON)
```

Groups

Data input/output, *Solution input/output*

MSK_IPAR_WRITE_INT_HEAD

Controls whether the header section is written to the integer solution file.

Default

ON

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_WRITE_INT_HEAD, MSK_ON)
```

Groups

Data input/output, *Solution input/output*

MSK_IPAR_WRITE_INT_VARIABLES

Controls whether the variables section is written to the integer solution file.

Default

ON

Accepted

ON, *OFF* (see *onoffkey*)

Example

```
putintparam(task, MSK_IPAR_WRITE_INT_VARIABLES, MSK_ON)
```

Groups

Data input/output, *Solution input/output*

MSK_IPAR_WRITE_JSON_INDENTATION

When set, the JSON task and solution files are written with indentation for better readability.

Default

OFF

Accepted

ON, *OFF* (see *onoffkey*)

Example
`putintparam(task, MSK_IPAR_WRITE_JSON_INDENTATION, MSK_OFF)`
Groups
Data input/output

`MSK_IPAR_WRITE_LP_FULL_OBJ`

Write all variables, including the ones with 0-coefficients, in the objective.

Default
ON
Accepted
ON, *OFF* (see *onoffkey*)
Example
`putintparam(task, MSK_IPAR_WRITE_LP_FULL_OBJ, MSK_ON)`
Groups
Data input/output

`MSK_IPAR_WRITE_LP_LINE_WIDTH`

Maximum width of line in an LP file written by **MOSEK**.

Default
80
Accepted
[40; +inf]
Example
`putintparam(task, MSK_IPAR_WRITE_LP_LINE_WIDTH, 80)`
Groups
Data input/output

`MSK_IPAR_WRITE_MPS_FORMAT`

Controls in which format the MPS file is written.

Default
FREE
Accepted
STRICT, *RELAXED*, *FREE*, *CPLEX* (see *mpsformat*)
Example
`putintparam(task, MSK_IPAR_WRITE_MPS_FORMAT, MSK_MPS_FORMAT_FREE)`
Groups
Data input/output

`MSK_IPAR_WRITE_MPS_INT`

Controls if marker records are written to the MPS file to indicate whether variables are integer restricted.

Default
ON
Accepted
ON, *OFF* (see *onoffkey*)
Example
`putintparam(task, MSK_IPAR_WRITE_MPS_INT, MSK_ON)`
Groups
Data input/output

`MSK_IPAR_WRITE_SOL_BARVARIABLES`

Controls whether the symmetric matrix variables section is written to the solution file.

Default
ON

Accepted
ON, *OFF* (see *onoffkey*)
Example
`putintparam(task, MSK_IPAR_WRITE_SOL_BARVARIABLES, MSK_ON)`
Groups
Data input/output, *Solution input/output*

`MSK_IPAR_WRITE_SOL_CONSTRAINTS`

Controls whether the constraint section is written to the solution file.

Default
ON
Accepted
ON, *OFF* (see *onoffkey*)
Example
`putintparam(task, MSK_IPAR_WRITE_SOL_CONSTRAINTS, MSK_ON)`
Groups
Data input/output, *Solution input/output*

`MSK_IPAR_WRITE_SOL_HEAD`

Controls whether the header section is written to the solution file.

Default
ON
Accepted
ON, *OFF* (see *onoffkey*)
Example
`putintparam(task, MSK_IPAR_WRITE_SOL_HEAD, MSK_ON)`
Groups
Data input/output, *Solution input/output*

`MSK_IPAR_WRITE_SOL_IGNORE_INVALID_NAMES`

Even if the names are invalid MPS names, then they are employed when writing the solution file.

Default
OFF
Accepted
ON, *OFF* (see *onoffkey*)
Example
`putintparam(task, MSK_IPAR_WRITE_SOL_IGNORE_INVALID_NAMES, MSK_OFF)`
Groups
Data input/output, *Solution input/output*

`MSK_IPAR_WRITE_SOL_VARIABLES`

Controls whether the variables section is written to the solution file.

Default
ON
Accepted
ON, *OFF* (see *onoffkey*)
Example
`putintparam(task, MSK_IPAR_WRITE_SOL_VARIABLES, MSK_ON)`
Groups
Data input/output, *Solution input/output*

15.5.3 String parameters

`sparam`

The enumeration type containing all string parameters.

`MSK_SPAR_BAS_SOL_FILE_NAME`

Name of the `bas` solution file.

Accepted
Any valid file name.

Example
```
putstrparam(task, MSK_SPAR_BAS_SOL_FILE_NAME, "somevalue")
```

Groups
Data input/output, *Solution input/output*

`MSK_SPAR_DATA_FILE_NAME`

Data are read and written to this file.

Accepted
Any valid file name.

Example
```
putstrparam(task, MSK_SPAR_DATA_FILE_NAME, "somevalue")
```

Groups
Data input/output

`MSK_SPAR_DEBUG_FILE_NAME`

MOSEK debug file.

Accepted
Any valid file name.

Example
```
putstrparam(task, MSK_SPAR_DEBUG_FILE_NAME, "somevalue")
```

Groups
Data input/output

`MSK_SPAR_INT_SOL_FILE_NAME`

Name of the `int` solution file.

Accepted
Any valid file name.

Example
```
putstrparam(task, MSK_SPAR_INT_SOL_FILE_NAME, "somevalue")
```

Groups
Data input/output, *Solution input/output*

`MSK_SPAR_ITR_SOL_FILE_NAME`

Name of the `itr` solution file.

Accepted
Any valid file name.

Example
```
putstrparam(task, MSK_SPAR_ITR_SOL_FILE_NAME, "somevalue")
```

Groups
Data input/output, *Solution input/output*

`MSK_SPAR_MIO_DEBUG_STRING`

For internal debugging purposes.

Accepted
Any valid string.

Example
```
putstrparam(task, MSK_SPAR_MIO_DEBUG_STRING, "somevalue")
```
Groups
Data input/output

`MSK_SPAR_PARAM_COMMENT_SIGN`

Only the first character in this string is used. It is considered as a start of comment sign in the **MOSEK** parameter file. Spaces are ignored in the string.

Default
%%
Accepted
Any valid string.
Example
```
putstrparam(task, MSK_SPAR_PARAM_COMMENT_SIGN, "%%")
```
Groups
Data input/output

`MSK_SPAR_PARAM_READ_FILE_NAME`

Modifications to the parameter database is read from this file.

Accepted
Any valid file name.
Example
```
putstrparam(task, MSK_SPAR_PARAM_READ_FILE_NAME, "somevalue")
```
Groups
Data input/output

`MSK_SPAR_PARAM_WRITE_FILE_NAME`

The parameter database is written to this file.

Accepted
Any valid file name.
Example
```
putstrparam(task, MSK_SPAR_PARAM_WRITE_FILE_NAME, "somevalue")
```
Groups
Data input/output

`MSK_SPAR_READ_MPS_BOU_NAME`

Name of the BOUNDS vector used. An empty name means that the first BOUNDS vector is used.

Accepted
Any valid MPS name.
Example
```
putstrparam(task, MSK_SPAR_READ_MPS_BOU_NAME, "somevalue")
```
Groups
Data input/output

`MSK_SPAR_READ_MPS_OBJ_NAME`

Name of the free constraint used as objective function. An empty name means that the first constraint is used as objective function.

Accepted
Any valid MPS name.
Example
```
putstrparam(task, MSK_SPAR_READ_MPS_OBJ_NAME, "somevalue")
```
Groups
Data input/output

`MSK_SPAR_READ_MPS_RAN_NAME`

Name of the RANGE vector used. An empty name means that the first RANGE vector is used.

Accepted
Any valid MPS name.

Example
```
putstrparam(task, MSK_SPAR_READ_MPS_RAN_NAME, "somevalue")
```

Groups
Data input/output

`MSK_SPAR_READ_MPS_RHS_NAME`

Name of the RHS used. An empty name means that the first RHS vector is used.

Accepted
Any valid MPS name.

Example
```
putstrparam(task, MSK_SPAR_READ_MPS_RHS_NAME, "somevalue")
```

Groups
Data input/output

`MSK_SPAR_REMOTE_OPTSERVER_HOST`

URL of the remote optimization server in the format `(http|https)://server:port`. If set, all subsequent calls to any **MOSEK** function that involves synchronous optimization will be sent to the specified OptServer instead of being executed locally. Passing empty string deactivates this redirection.

Accepted
Any valid URL.

Example
```
putstrparam(task, MSK_SPAR_REMOTE_OPTSERVER_HOST, "somevalue")
```

Groups
Overall system

`MSK_SPAR_REMOTE_TLS_CERT`

List of known server certificates in PEM format.

Accepted
PEM files separated by new-lines.

Example
```
putstrparam(task, MSK_SPAR_REMOTE_TLS_CERT, "somevalue")
```

Groups
Overall system

`MSK_SPAR_REMOTE_TLS_CERT_PATH`

Path to known server certificates in PEM format.

Accepted
Any valid path.

Example
```
putstrparam(task, MSK_SPAR_REMOTE_TLS_CERT_PATH, "somevalue")
```

Groups
Overall system

`MSK_SPAR_SENSITIVITY_FILE_NAME`

If defined *sensitivityreport* reads this file as a sensitivity analysis data file specifying the type of analysis to be done.

Accepted
Any valid string.

Example

```
putstrparam(task, MSK_SPAR_SENSITIVITY_FILE_NAME, "somevalue")
```

Groups

Data input/output

`MSK_SPAR_SENSITIVITY_RES_FILE_NAME`

If this is a nonempty string, then *sensitivityreport* writes results to this file.

Accepted

Any valid string.

Example

```
putstrparam(task, MSK_SPAR_SENSITIVITY_RES_FILE_NAME, "somevalue")
```

Groups

Data input/output

`MSK_SPAR_SOL_FILTER_XC_LOW`

A filter used to determine which constraints should be listed in the solution file. A value of 0.5 means that all constraints having `xc[i]>0.5` should be listed, whereas `+0.5` means that all constraints having `xc[i]>=blc[i]+0.5` should be listed. An empty filter means that no filter is applied.

Accepted

Any valid filter.

Example

```
putstrparam(task, MSK_SPAR_SOL_FILTER_XC_LOW, "somevalue")
```

Groups

Data input/output, *Solution input/output*

`MSK_SPAR_SOL_FILTER_XC_UPR`

A filter used to determine which constraints should be listed in the solution file. A value of `0.5` means that all constraints having `xc[i]<0.5` should be listed, whereas `-0.5` means all constraints having `xc[i]<=buc[i]-0.5` should be listed. An empty filter means that no filter is applied.

Accepted

Any valid filter.

Example

```
putstrparam(task, MSK_SPAR_SOL_FILTER_XC_UPR, "somevalue")
```

Groups

Data input/output, *Solution input/output*

`MSK_SPAR_SOL_FILTER_XX_LOW`

A filter used to determine which variables should be listed in the solution file. A value of "0.5" means that all constraints having `xx[j]>=0.5` should be listed, whereas "+0.5" means that all constraints having `xx[j]>=blx[j]+0.5` should be listed. An empty filter means no filter is applied.

Accepted

Any valid filter.

Example

```
putstrparam(task, MSK_SPAR_SOL_FILTER_XX_LOW, "somevalue")
```

Groups

Data input/output, *Solution input/output*

`MSK_SPAR_SOL_FILTER_XX_UPR`

A filter used to determine which variables should be listed in the solution file. A value of "0.5" means that all constraints having `xx[j]<0.5` should be printed, whereas "-0.5" means all constraints having `xx[j]<=bux[j]-0.5` should be listed. An empty filter means no filter is applied.

Accepted

Any valid file name.

Example

```
putstrparam(task, MSK_SPAR_SOL_FILTER_XX_UPR, "somevalue")
```

Groups

Data input/output, *Solution input/output*

`MSK_SPAR_STAT_KEY`

Key used when writing the summary file.

Accepted

Any valid string.

Example

```
putstrparam(task, MSK_SPAR_STAT_KEY, "somevalue")
```

Groups

Data input/output

`MSK_SPAR_STAT_NAME`

Name used when writing the statistics file.

Accepted

Any valid XML string.

Example

```
putstrparam(task, MSK_SPAR_STAT_NAME, "somevalue")
```

Groups

Data input/output

15.6 Response codes

Response codes include:

- *Termination codes*
- *Warnings*
- *Errors*

The numerical code (in brackets) identifies the response in error messages and in the log output.

`rescode`

The enumeration type containing all response codes.

15.6.1 Termination

`MSK_RES_OK (0)`

No error occurred.

`MSK_RES_TRM_MAX_ITERATIONS (100000)`

The optimizer terminated at the maximum number of iterations.

`MSK_RES_TRM_MAX_TIME (100001)`

The optimizer terminated at the maximum amount of time.

`MSK_RES_TRM_OBJECTIVE_RANGE (100002)`

The optimizer terminated with an objective value outside the objective range.

`MSK_RES_TRM_MIO_NUM_RELAXS (100008)`

The mixed-integer optimizer terminated as the maximum number of relaxations was reached.

`MSK_RES_TRM_MIO_NUM_BRANCHES (100009)`

The mixed-integer optimizer terminated as the maximum number of branches was reached.

`MSK_RES_TRM_NUM_MAX_NUM_INT_SOLUTIONS (100015)`

The mixed-integer optimizer terminated as the maximum number of feasible solutions was reached.

MSK_RES_TRM_STALL (100006)

The optimizer is terminated due to slow progress.

Stalling means that numerical problems prevent the optimizer from making reasonable progress and that it makes no sense to continue. In many cases this happens if the problem is badly scaled or otherwise ill-conditioned. There is no guarantee that the solution will be feasible or optimal. However, often stalling happens near the optimum, and the returned solution may be of good quality. Therefore, it is recommended to check the status of the solution. If the solution status is optimal the solution is most likely good enough for most practical purposes.

Please note that if a linear optimization problem is solved using the interior-point optimizer with basis identification turned on, the returned basic solution likely to have high accuracy, even though the optimizer stalled.

Some common causes of stalling are a) badly scaled models, b) near feasible or near infeasible problems.

MSK_RES_TRM_USER_CALLBACK (100007)

The optimizer terminated due to the return of the user-defined callback function.

MSK_RES_TRM_MAX_NUM_SETBACKS (100020)

The optimizer terminated as the maximum number of set-backs was reached. This indicates serious numerical problems and a possibly badly formulated problem.

MSK_RES_TRM_NUMERICAL_PROBLEM (100025)

The optimizer terminated due to numerical problems.

MSK_RES_TRM_LOST_RACE (100027)

Lost a race.

MSK_RES_TRM_INTERNAL (100030)

The optimizer terminated due to some internal reason. Please contact **MOSEK** support.

MSK_RES_TRM_INTERNAL_STOP (100031)

The optimizer terminated for internal reasons. Please contact **MOSEK** support.

MSK_RES_TRM_SERVER_MAX_TIME (100032)

remote server terminated **MOSEK** on time limit criteria.

MSK_RES_TRM_SERVER_MAX_MEMORY (100033)

remote server terminated **MOSEK** on memory limit criteria.

15.6.2 Warnings

MSK_RES_WRN_OPEN_PARAM_FILE (50)

The parameter file could not be opened.

MSK_RES_WRN_LARGE_BOUND (51)

A numerically large bound value is specified.

MSK_RES_WRN_LARGE_LO_BOUND (52)

A numerically large lower bound value is specified.

MSK_RES_WRN_LARGE_UP_BOUND (53)

A numerically large upper bound value is specified.

MSK_RES_WRN_LARGE_CON_FX (54)

An equality constraint is fixed to a numerically large value. This can cause numerical problems.

MSK_RES_WRN_LARGE_CJ (57)

A numerically large value is specified for one c_j.

MSK_RES_WRN_LARGE_AIJ (62)

A numerically large value is specified for an $a_{i,j}$ element in A. The parameter *MSK_DPAR_DATA_TOL_AIJ_LARGE* controls when an $a_{i,j}$ is considered large.

MSK_RES_WRN_ZERO_AIJ (63)

One or more zero elements are specified in A.

MSK_RES_WRN_NAME_MAX_LEN (65)

A name is longer than the buffer that is supposed to hold it.

MSK_RES_WRN_SPAR_MAX_LEN (66)

A value for a string parameter is longer than the buffer that is supposed to hold it.

MSK_RES_WRN_MPS_SPLIT_RHS_VECTOR (70)

An RHS vector is split into several nonadjacent parts in an MPS file.

MSK_RES_WRN_MPS_SPLIT_RAN_VECTOR (71)

A RANGE vector is split into several nonadjacent parts in an MPS file.

MSK_RES_WRN_MPS_SPLIT_BOU_VECTOR (72)

A BOUNDS vector is split into several nonadjacent parts in an MPS file.

MSK_RES_WRN_LP_OLD_QUAD_FORMAT (80)

Missing '/2' after quadratic expressions in bound or objective.

MSK_RES_WRN_LP_DROP_VARIABLE (85)

Ignored a variable because the variable was not previously defined. Usually this implies that a variable appears in the bound section but not in the objective or the constraints.

MSK_RES_WRN_NZ_IN_UPR_TRI (200)

Non-zero elements specified in the upper triangle of a matrix were ignored.

MSK_RES_WRN_DROPPED_NZ_QOBJ (201)

One or more non-zero elements were dropped in the Q matrix in the objective.

MSK_RES_WRN_IGNORE_INTEGER (250)

Ignored integer constraints.

MSK_RES_WRN_NO_GLOBAL_OPTIMIZER (251)

No global optimizer is available.

MSK_RES_WRN_MIO_INFEASIBLE_FINAL (270)

The final mixed-integer problem with all the integer variables fixed at their optimal values is infeasible.

MSK_RES_WRN_SOL_FILTER (300)

Invalid solution filter is specified.

MSK_RES_WRN_UNDEF_SOL_FILE_NAME (350)

Undefined name occurred in a solution.

MSK_RES_WRN_SOL_FILE_IGNORED_CON (351)

One or more lines in the constraint section were ignored when reading a solution file.

MSK_RES_WRN_SOL_FILE_IGNORED_VAR (352)

One or more lines in the variable section were ignored when reading a solution file.

MSK_RES_WRN_TOO_FEW_BASIS_VARS (400)

An incomplete basis has been specified. Too few basis variables are specified.

MSK_RES_WRN_TOO_MANY_BASIS_VARS (405)

A basis with too many variables has been specified.

MSK_RES_WRN_LICENSE_EXPIRE (500)

The license expires.

MSK_RES_WRN_LICENSE_SERVER (501)

The license server is not responding.

MSK_RES_WRN_EMPTY_NAME (502)

A variable or constraint name is empty. The output file may be invalid.

MSK_RES_WRN_USING_GENERIC_NAMES (503)

Generic names are used because a name invalid. For instance when writing an LP file the names must not contain blanks or start with a digit. Also remeber to give the objective function a name.

MSK_RES_WRN_INVALID_MPS_NAME (504)

A name e.g. a row name is not a valid MPS name.

MSK_RES_WRN_INVALID_MPS_OBJ_NAME (505)

The objective name is not a valid MPS name.

MSK_RES_WRN_LICENSE_FEATURE_EXPIRE (509)

The license expires.

MSK_RES_WRN_PARAM_NAME_DOU (510)

The parameter name is not recognized as a double parameter.

MSK_RES_WRN_PARAM_NAME_INT (511)

The parameter name is not recognized as a integer parameter.

MSK_RES_WRN_PARAM_NAME_STR (512)

The parameter name is not recognized as a string parameter.

MSK_RES_WRN_PARAM_STR_VALUE (515)

The string is not recognized as a symbolic value for the parameter.

MSK_RES_WRN_PARAM_IGNORED_CMIO (516)

A parameter was ignored by the conic mixed integer optimizer.

MSK_RES_WRN_ZEROS_IN_SPARSE_ROW (705)

One or more (near) zero elements are specified in a sparse row of a matrix. Since, it is redundant to specify zero elements then it may indicate an error.

MSK_RES_WRN_ZEROS_IN_SPARSE_COL (710)

One or more (near) zero elements are specified in a sparse column of a matrix. It is redundant to specify zero elements. Hence, it may indicate an error.

MSK_RES_WRN_INCOMPLETE_LINEAR_DEPENDENCY_CHECK (800)

The linear dependency check(s) is incomplete. Normally this is not an important warning unless the optimization problem has been formulated with linear dependencies. Linear dependencies may prevent **MOSEK** from solving the problem.

MSK_RES_WRN_ELIMINATOR_SPACE (801)

The eliminator is skipped at least once due to lack of space.

MSK_RES_WRN_PRESOLVE_OUTOFSPACE (802)

The presolve is incomplete due to lack of space.

MSK_RES_WRN_PRESOLVE_PRIMAL_PERTURBATIONS (803)

The presolve perturbed the bounds of the primal problem. This is an indication that the problem is nearly infeasible.

MSK_RES_WRN_WRITE_CHANGED_NAMES (830)

Some names were changed because they were invalid for the output file format.

MSK_RES_WRN_WRITE_DISCARDED_CFIX (831)

The fixed objective term could not be converted to a variable and was discarded in the output file.

MSK_RES_WRN_DUPLICATE_CONSTRAINT_NAMES (850)

Two constraint names are identical.

MSK_RES_WRN_DUPLICATE_VARIABLE_NAMES (851)

Two variable names are identical.

MSK_RES_WRN_DUPLICATE_BARVARIABLE_NAMES (852)

Two barvariable names are identical.

MSK_RES_WRN_DUPLICATE_CONE_NAMES (853)

Two cone names are identical.

MSK_RES_WRN_ANA_LARGE_BOUNDS (900)

This warning is issued by the problem analyzer, if one or more constraint or variable bounds are very large. One should consider omitting these bounds entirely by setting them to +inf or -inf.

MSK_RES_WRN_ANA_C_ZERO (901)

This warning is issued by the problem analyzer, if the coefficients in the linear part of the objective are all zero.

MSK_RES_WRN_ANA_EMPTY_COLS (902)

This warning is issued by the problem analyzer, if columns, in which all coefficients are zero, are found.

MSK_RES_WRN_ANA_CLOSE_BOUNDS (903)

This warning is issued by problem analyzer, if ranged constraints or variables with very close upper and lower bounds are detected. One should consider treating such constraints as equalities and such variables as constants.

MSK_RES_WRN_ANA_ALMOST_INT_BOUNDS (904)

This warning is issued by the problem analyzer if a constraint is bound nearly integral.

MSK_RES_WRN_NO_INFEASIBILITY_REPORT_WHEN_MATRIX_VARIABLES (930)

An infeasibility report is not available when the problem contains matrix variables.

MSK_RES_WRN_GETDUAL_IGNORES_INTEGRALITY (940)

Dualizer ignores integer variables and disjunctive constraints.

MSK_RES_WRN_NO_DUALIZER (950)

No automatic dualizer is available for the specified problem. The primal problem is solved.

MSK_RES_WRN_SYM_MAT_LARGE (960)

A numerically large value is specified for an $e_{i,j}$ element in E. The parameter *MSK_DPAR_DATA_SYM_MAT_TOL_LARGE* controls when an $e_{i,j}$ is considered large.

MSK_RES_WRN_MODIFIED_DOUBLE_PARAMETER (970)

A double parameter related to solver tolerances has a non-default value.

MSK_RES_WRN_LARGE_FIJ (980)

A numerically large value is specified for an $f_{i,j}$ element in F. The parameter *MSK_DPAR_DATA_TOL_AIJ_LARGE* controls when an $f_{i,j}$ is considered large.

MSK_RES_WRN_PTF_UNKNOWN_SECTION (981)

Unexpected section in PTF file

15.6.3 Errors

MSK_RES_ERR_LICENSE (1000)

Invalid license.

MSK_RES_ERR_LICENSE_EXPIRED (1001)

The license has expired.

MSK_RES_ERR_LICENSE_VERSION (1002)

The license is valid for another version of **MOSEK**.

MSK_RES_ERR_LICENSE_OLD_SERVER_VERSION (1003)

The version of the FlexLM license server is too old. You should upgrade the license server to one matching this version of **MOSEK**. It will support this and all older versions of **MOSEK**.

This error can appear if the client was updated to a new version which includes an upgrade of the licensing module, making it incompatible with a much older license server.

MSK_RES_ERR_SIZE_LICENSE (1005)

The problem is bigger than the license.

MSK_RES_ERR_PROB_LICENSE (1006)

The software is not licensed to solve the problem.

MSK_RES_ERR_FILE_LICENSE (1007)

Invalid license file.

MSK_RES_ERR_MISSING_LICENSE_FILE (1008)

MOSEK cannot find license file or a token server. See the **MOSEK** licensing manual for details.

MSK_RES_ERR_SIZE_LICENSE_CON (1010)

The problem has too many constraints to be solved with the available license.

MSK_RES_ERR_SIZE_LICENSE_VAR (1011)

The problem has too many variables to be solved with the available license.

MSK_RES_ERR_SIZE_LICENSE_INTVAR (1012)

The problem contains too many integer variables to be solved with the available license.

MSK_RES_ERR_OPTIMIZER_LICENSE (1013)

The optimizer required is not licensed.

MSK_RES_ERR_FLEXLM (1014)

The FLEXlm license manager reported an error.

MSK_RES_ERR_LICENSE_SERVER (1015)

The license server is not responding.

`MSK_RES_ERR_LICENSE_MAX (1016)`

Maximum number of licenses is reached.

`MSK_RES_ERR_LICENSE_MOSEKLM_DAEMON (1017)`

The MOSEKLM license manager daemon is not up and running.

`MSK_RES_ERR_LICENSE_FEATURE (1018)`

A requested feature is not available in the license file(s). Most likely due to an incorrect license system setup.

`MSK_RES_ERR_PLATFORM_NOT_LICENSED (1019)`

A requested license feature is not available for the required platform.

`MSK_RES_ERR_LICENSE_CANNOT_ALLOCATE (1020)`

The license system cannot allocate the memory required.

`MSK_RES_ERR_LICENSE_CANNOT_CONNECT (1021)`

MOSEK cannot connect to the license server. Most likely the license server is not up and running.

`MSK_RES_ERR_LICENSE_INVALID_HOSTID (1025)`

The host ID specified in the license file does not match the host ID of the computer.

`MSK_RES_ERR_LICENSE_SERVER_VERSION (1026)`

The version specified in the checkout request is greater than the highest version number the daemon supports.

`MSK_RES_ERR_LICENSE_NO_SERVER_SUPPORT (1027)`

The license server does not support the requested feature. Possible reasons for this error include:

- The feature has expired.
- The feature's start date is later than today's date.
- The version requested is higher than feature's the highest supported version.
- A corrupted license file.

Try restarting the license and inspect the license server debug file, usually called `lmgrd.log`.

`MSK_RES_ERR_LICENSE_NO_SERVER_LINE (1028)`

There is no `SERVER` line in the license file. All non-zero license count features need at least one `SERVER` line.

`MSK_RES_ERR_OLDER_DLL (1035)`

The dynamic link library is older than the specified version.

`MSK_RES_ERR_NEWER_DLL (1036)`

The dynamic link library is newer than the specified version.

`MSK_RES_ERR_LINK_FILE_DLL (1040)`

A file cannot be linked to a stream in the DLL version.

`MSK_RES_ERR_THREAD_MUTEX_INIT (1045)`

Could not initialize a mutex.

`MSK_RES_ERR_THREAD_MUTEX_LOCK (1046)`

Could not lock a mutex.

`MSK_RES_ERR_THREAD_MUTEX_UNLOCK (1047)`

Could not unlock a mutex.

`MSK_RES_ERR_THREAD_CREATE (1048)`

Could not create a thread. This error may occur if a large number of environments are created and not deleted again. In any case it is a good practice to minimize the number of environments created.

`MSK_RES_ERR_THREAD_COND_INIT (1049)`

Could not initialize a condition.

`MSK_RES_ERR_UNKNOWN (1050)`

Unknown error.

`MSK_RES_ERR_SPACE (1051)`

Out of space.

`MSK_RES_ERR_FILE_OPEN (1052)`

Error while opening a file.

`MSK_RES_ERR_FILE_READ (1053)`

File read error.

`MSK_RES_ERR_FILE_WRITE (1054)`

File write error.

`MSK_RES_ERR_DATA_FILE_EXT (1055)`

The data file format cannot be determined from the file name.

`MSK_RES_ERR_INVALID_FILE_NAME (1056)`

An invalid file name has been specified.

`MSK_RES_ERR_INVALID_SOL_FILE_NAME (1057)`

An invalid file name has been specified.

`MSK_RES_ERR_END_OF_FILE (1059)`

End of file has been reached unexpectedly.

`MSK_RES_ERR_NULL_ENV (1060)`

`env` is a `nothing` pointer.

`MSK_RES_ERR_NULL_TASK (1061)`

`task` is a `nothing` pointer.

`MSK_RES_ERR_INVALID_STREAM (1062)`

An invalid stream is referenced.

`MSK_RES_ERR_NO_INIT_ENV (1063)`

`env` is not initialized.

`MSK_RES_ERR_INVALID_TASK (1064)`

The `task` is invalid.

`MSK_RES_ERR_NULL_POINTER (1065)`

An argument to a function is unexpectedly a `nothing` pointer.

`MSK_RES_ERR_LIVING_TASKS (1066)`

All tasks associated with an enviroment must be deleted before the environment is deleted. There are still some undeleted tasks.

`MSK_RES_ERR_READ_GZIP (1067)`

Error encountered in GZIP stream.

`MSK_RES_ERR_READ_ZSTD (1068)`

Error encountered in ZSTD stream.

`MSK_RES_ERR_READ_ASYNC (1069)`

Error encountered in async stream.

`MSK_RES_ERR_BLANK_NAME (1070)`

An all blank name has been specified.

`MSK_RES_ERR_DUP_NAME (1071)`

The same name was used multiple times for the same problem item type.

`MSK_RES_ERR_FORMAT_STRING (1072)`

The name format string is invalid.

`MSK_RES_ERR_SPARSITY_SPECIFICATION (1073)`

The sparsity included an index that was out of bounds of the shape.

`MSK_RES_ERR_MISMATCHING_DIMENSION (1074)`

Mismatching dimensions specified in arguments

`MSK_RES_ERR_INVALID_OBJ_NAME (1075)`

An invalid objective name is specified.

`MSK_RES_ERR_INVALID_CON_NAME (1076)`

An invalid constraint name is used.

`MSK_RES_ERR_INVALID_VAR_NAME (1077)`

An invalid variable name is used.

MSK_RES_ERR_INVALID_CONE_NAME (1078)

An invalid cone name is used.

MSK_RES_ERR_INVALID_BARVAR_NAME (1079)

An invalid symmetric matrix variable name is used.

MSK_RES_ERR_SPACE_LEAKING (1080)

MOSEK is leaking memory. This can be due to either an incorrect use of **MOSEK** or a bug.

MSK_RES_ERR_SPACE_NO_INFO (1081)

No available information about the space usage.

MSK_RES_ERR_DIMENSION_SPECIFICATION (1082)

Invalid dimension specification

MSK_RES_ERR_AXIS_NAME_SPECIFICATION (1083)

Invalid axis names specification

MSK_RES_ERR_READ_PREMATURE_EOF (1089)

Encountered premature end-of-file in input stream.

MSK_RES_ERR_READ_FORMAT (1090)

The specified format cannot be read.

MSK_RES_ERR_WRITE_LP_INVALID_VAR_NAMES (1091)

Invalid variable name. Cannot write valid LP file.

MSK_RES_ERR_WRITE_LP_DUPLICATE_VAR_NAMES (1092)

Duplicate variable names. Cannot write valid LP file.

MSK_RES_ERR_WRITE_LP_INVALID_CON_NAMES (1093)

Invalid constraint name. Cannot write valid LP file.

MSK_RES_ERR_WRITE_LP_DUPLICATE_CON_NAMES (1094)

Duplicate constraint names. Cannot write valid LP file.

MSK_RES_ERR_MPS_FILE (1100)

An error occurred while reading an MPS file.

MSK_RES_ERR_MPS_INV_FIELD (1101)

A field in the MPS file is invalid. Probably it is too wide.

MSK_RES_ERR_MPS_INV_MARKER (1102)

An invalid marker has been specified in the MPS file.

MSK_RES_ERR_MPS_NULL_CON_NAME (1103)

An empty constraint name is used in an MPS file.

MSK_RES_ERR_MPS_NULL_VAR_NAME (1104)

An empty variable name is used in an MPS file.

MSK_RES_ERR_MPS_UNDEF_CON_NAME (1105)

An undefined constraint name occurred in an MPS file.

MSK_RES_ERR_MPS_UNDEF_VAR_NAME (1106)

An undefined variable name occurred in an MPS file.

MSK_RES_ERR_MPS_INVALID_CON_KEY (1107)

An invalid constraint key occurred in an MPS file.

MSK_RES_ERR_MPS_INVALID_BOUND_KEY (1108)

An invalid bound key occurred in an MPS file.

MSK_RES_ERR_MPS_INVALID_SEC_NAME (1109)

An invalid section name occurred in an MPS file.

MSK_RES_ERR_MPS_NO_OBJECTIVE (1110)

No objective is defined in an MPS file.

MSK_RES_ERR_MPS_SPLITTED_VAR (1111)

All elements in a column of the A matrix must be specified consecutively. Hence, it is illegal to specify non-zero elements in A for variable 1, then for variable 2 and then variable 1 again.

MSK_RES_ERR_MPS_MUL_CON_NAME (1112)

A constraint name was specified multiple times in the `ROWS` section.

MSK_RES_ERR_MPS_MUL_QSEC (1113)

Multiple QSECTIONs are specified for a constraint in the MPS data file.

MSK_RES_ERR_MPS_MUL_QOBJ (1114)

The Q term in the objective is specified multiple times in the MPS data file.

MSK_RES_ERR_MPS_INV_SEC_ORDER (1115)

The sections in the MPS data file are not in the correct order.

MSK_RES_ERR_MPS_MUL_CSEC (1116)

Multiple CSECTIONs are given the same name.

MSK_RES_ERR_MPS_CONE_TYPE (1117)

Invalid cone type specified in a CSECTION.

MSK_RES_ERR_MPS_CONE_OVERLAP (1118)

A variable is specified to be a member of several cones.

MSK_RES_ERR_MPS_CONE_REPEAT (1119)

A variable is repeated within the CSECTION.

MSK_RES_ERR_MPS_NON_SYMMETRIC_Q (1120)

A non symmetric matrix has been speciefied.

MSK_RES_ERR_MPS_DUPLICATE_Q_ELEMENT (1121)

Duplicate elements is specfied in a Q matrix.

MSK_RES_ERR_MPS_INVALID_OBJSENSE (1122)

An invalid objective sense is specified.

MSK_RES_ERR_MPS_TAB_IN_FIELD2 (1125)

A tab char occurred in field 2.

MSK_RES_ERR_MPS_TAB_IN_FIELD3 (1126)

A tab char occurred in field 3.

MSK_RES_ERR_MPS_TAB_IN_FIELD5 (1127)

A tab char occurred in field 5.

MSK_RES_ERR_MPS_INVALID_OBJ_NAME (1128)

An invalid objective name is specified.

MSK_RES_ERR_MPS_INVALID_KEY (1129)

An invalid indicator key occurred in an MPS file.

MSK_RES_ERR_MPS_INVALID_INDICATOR_CONSTRAINT (1130)

An invalid indicator constraint is used. It must not be a ranged constraint.

MSK_RES_ERR_MPS_INVALID_INDICATOR_VARIABLE (1131)

An invalid indicator variable is specfied. It must be a binary variable.

MSK_RES_ERR_MPS_INVALID_INDICATOR_VALUE (1132)

An invalid indicator value is specfied. It must be either 0 or 1.

MSK_RES_ERR_MPS_INVALID_INDICATOR_QUADRATIC_CONSTRAINT (1133)

A quadratic constraint can be be an indicator constraint.

MSK_RES_ERR_OPF_SYNTAX (1134)

Syntax error in an OPF file

MSK_RES_ERR_OPF_PREMATURE_EOF (1136)

Premature end of file in an OPF file.

MSK_RES_ERR_OPF_MISMATCHED_TAG (1137)

Mismatched end-tag in OPF file

MSK_RES_ERR_OPF_DUPLICATE_BOUND (1138)

Either upper or lower bound was specified twice in OPF file

MSK_RES_ERR_OPF_DUPLICATE_CONSTRAINT_NAME (1139)

Duplicate constraint name in OPF File

MSK_RES_ERR_OPF_INVALID_CONE_TYPE (1140)

Invalid cone type in OPF File

MSK_RES_ERR_OPF_INCORRECT_TAG_PARAM (1141)

Invalid number of parameters in start-tag in OPF File

MSK_RES_ERR_OPF_INVALID_TAG (1142)

Invalid start-tag in OPF File

MSK_RES_ERR_OPF_DUPLICATE_CONE_ENTRY (1143)

Same variable appears in multiple cones in OPF File

MSK_RES_ERR_OPF_TOO_LARGE (1144)

The problem is too large to be correctly loaded

MSK_RES_ERR_OPF_DUAL_INTEGER_SOLUTION (1146)

Dual solution values are not allowed in OPF File

MSK_RES_ERR_LP_EMPTY (1151)

The problem cannot be written to an LP formatted file.

MSK_RES_ERR_WRITE_MPS_INVALID_NAME (1153)

An invalid name is created while writing an MPS file. Usually this will make the MPS file unreadable.

MSK_RES_ERR_LP_INVALID_VAR_NAME (1154)

A variable name is invalid when used in an LP formatted file.

MSK_RES_ERR_WRITE_OPF_INVALID_VAR_NAME (1156)

Empty variable names cannot be written to OPF files.

MSK_RES_ERR_LP_FILE_FORMAT (1157)

Syntax error in an LP file.

MSK_RES_ERR_LP_EXPECTED_NUMBER (1158)

Expected a number in LP file

MSK_RES_ERR_READ_LP_MISSING_END_TAG (1159)

Syntax error in LP file. Possibly missing End tag.

MSK_RES_ERR_LP_INDICATOR_VAR (1160)

An indicator variable was not declared binary

MSK_RES_ERR_LP_EXPECTED_OBJECTIVE (1161)

Expected an objective section in LP file

MSK_RES_ERR_LP_EXPECTED_CONSTRAINT_RELATION (1162)

Expected constraint relation

MSK_RES_ERR_LP_AMBIGUOUS_CONSTRAINT_BOUND (1163)

Constraint has ambiguous or invalid bound

MSK_RES_ERR_LP_DUPLICATE_SECTION (1164)

Duplicate section

MSK_RES_ERR_READ_LP_DELAYED_ROWS_NOT_SUPPORTED (1165)

Duplicate section

MSK_RES_ERR_WRITING_FILE (1166)

An error occurred while writing file

MSK_RES_ERR_WRITE_ASYNC (1167)

An error occurred while performing asynchronous writing

MSK_RES_ERR_INVALID_NAME_IN_SOL_FILE (1170)

An invalid name occurred in a solution file.

MSK_RES_ERR_JSON_SYNTAX (1175)

Syntax error in an JSON data

MSK_RES_ERR_JSON_STRING (1176)

Error in JSON string.

MSK_RES_ERR_JSON_NUMBER_OVERFLOW (1177)

Invalid number entry - wrong type or value overflow.

MSK_RES_ERR_JSON_FORMAT (1178)

Error in an JSON Task file

MSK_RES_ERR_JSON_DATA (1179)

Inconsistent data in JSON Task file

MSK_RES_ERR_JSON_MISSING_DATA (1180)

Missing data section in JSON task file.

MSK_RES_ERR_PTF_INCOMPATIBILITY (1181)

Incompatible item

MSK_RES_ERR_PTF_UNDEFINED_ITEM (1182)

Undefined symbol referenced

MSK_RES_ERR_PTF_INCONSISTENCY (1183)

Inconsistent size of item

MSK_RES_ERR_PTF_FORMAT (1184)

Syntax error in an PTF file

MSK_RES_ERR_ARGUMENT_LENNEQ (1197)

Incorrect length of arguments.

MSK_RES_ERR_ARGUMENT_TYPE (1198)

Incorrect argument type.

MSK_RES_ERR_NUM_ARGUMENTS (1199)

Incorrect number of function arguments.

MSK_RES_ERR_IN_ARGUMENT (1200)

A function argument is incorrect.

MSK_RES_ERR_ARGUMENT_DIMENSION (1201)

A function argument is of incorrect dimension.

MSK_RES_ERR_SHAPE_IS_TOO_LARGE (1202)

The size of the n-dimensional shape is too large.

MSK_RES_ERR_INDEX_IS_TOO_SMALL (1203)

An index in an argument is too small.

MSK_RES_ERR_INDEX_IS_TOO_LARGE (1204)

An index in an argument is too large.

MSK_RES_ERR_INDEX_IS_NOT_UNIQUE (1205)

An index in an argument is not unique.

MSK_RES_ERR_PARAM_NAME (1206)

The parameter name is not correct.

MSK_RES_ERR_PARAM_NAME_DOU (1207)

The parameter name is not correct for a double parameter.

MSK_RES_ERR_PARAM_NAME_INT (1208)

The parameter name is not correct for an integer parameter.

MSK_RES_ERR_PARAM_NAME_STR (1209)

The parameter name is not correct for a string parameter.

MSK_RES_ERR_PARAM_INDEX (1210)

Parameter index is out of range.

MSK_RES_ERR_PARAM_IS_TOO_LARGE (1215)

The parameter value is too large.

MSK_RES_ERR_PARAM_IS_TOO_SMALL (1216)

The parameter value is too small.

MSK_RES_ERR_PARAM_VALUE_STR (1217)

The parameter value string is incorrect.

MSK_RES_ERR_PARAM_TYPE (1218)

The parameter type is invalid.

MSK_RES_ERR_INF_DOU_INDEX (1219)

A double information index is out of range for the specified type.

MSK_RES_ERR_INF_INT_INDEX (1220)

An integer information index is out of range for the specified type.

MSK_RES_ERR_INDEX_ARR_IS_TOO_SMALL (1221)

An index in an array argument is too small.

MSK_RES_ERR_INDEX_ARR_IS_TOO_LARGE (1222)

An index in an array argument is too large.

MSK_RES_ERR_INF_LINT_INDEX (1225)

A long integer information index is out of range for the specified type.

MSK_RES_ERR_ARG_IS_TOO_SMALL (1226)

The value of a argument is too small.

MSK_RES_ERR_ARG_IS_TOO_LARGE (1227)

The value of a argument is too large.

MSK_RES_ERR_INVALID_WHICHSOL (1228)

`whichsol` is invalid.

MSK_RES_ERR_INF_DOU_NAME (1230)

A double information name is invalid.

MSK_RES_ERR_INF_INT_NAME (1231)

An integer information name is invalid.

MSK_RES_ERR_INF_TYPE (1232)

The information type is invalid.

MSK_RES_ERR_INF_LINT_NAME (1234)

A long integer information name is invalid.

MSK_RES_ERR_INDEX (1235)

An index is out of range.

MSK_RES_ERR_WHICHSOL (1236)

The solution defined by `whichsol` does not exists.

MSK_RES_ERR_SOLITEM (1237)

The solution item number `solitem` is invalid. Please note that *MSK_SOL_ITEM_SNX* is invalid for the basic solution.

MSK_RES_ERR_WHICHITEM_NOT_ALLOWED (1238)

`whichitem` is unacceptable.

MSK_RES_ERR_MAXNUMCON (1240)

The maximum number of constraints specified is smaller than the number of constraints in the task.

MSK_RES_ERR_MAXNUMVAR (1241)

The maximum number of variables specified is smaller than the number of variables in the task.

MSK_RES_ERR_MAXNUMBARVAR (1242)

The maximum number of semidefinite variables specified is smaller than the number of semidefinite variables in the task.

MSK_RES_ERR_MAXNUMQNZ (1243)

The maximum number of non-zeros specified for the Q matrices is smaller than the number of non-zeros in the current Q matrices.

MSK_RES_ERR_TOO_SMALL_MAX_NUM_NZ (1245)

The maximum number of non-zeros specified is too small.

MSK_RES_ERR_INVALID_IDX (1246)

A specified index is invalid.

MSK_RES_ERR_INVALID_MAX_NUM (1247)

A specified index is invalid.

MSK_RES_ERR_UNALLOWED_WHICHSOL (1248)

The value od `whichsol` is not allowed.

MSK_RES_ERR_NUMCONLIM (1250)

Maximum number of constraints limit is exceeded.

MSK_RES_ERR_NUMVARLIM (1251)

Maximum number of variables limit is exceeded.

MSK_RES_ERR_TOO_SMALL_MAXNUMANZ (1252)

The maximum number of non-zeros specified for A is smaller than the number of non-zeros in the current A.

MSK_RES_ERR_INV_APTRE (1253)

`aptre[j]` is strictly smaller than `aptrb[j]` for some `j`.

MSK_RES_ERR_MUL_A_ELEMENT (1254)

An element in A is defined multiple times.

MSK_RES_ERR_INV_BK (1255)

Invalid bound key.

MSK_RES_ERR_INV_BKC (1256)

Invalid bound key is specified for a constraint.

MSK_RES_ERR_INV_BKX (1257)

An invalid bound key is specified for a variable.

MSK_RES_ERR_INV_VAR_TYPE (1258)

An invalid variable type is specified for a variable.

MSK_RES_ERR_SOLVER_PROBTYPE (1259)

Problem type does not match the chosen optimizer.

MSK_RES_ERR_OBJECTIVE_RANGE (1260)

Empty objective range.

MSK_RES_ERR_INV_RESCODE (1261)

Invalid response code.

MSK_RES_ERR_INV_IINF (1262)

Invalid integer information item.

MSK_RES_ERR_INV_LIINF (1263)

Invalid long integer information item.

MSK_RES_ERR_INV_DINF (1264)

Invalid double information item.

MSK_RES_ERR_BASIS (1266)

An invalid basis is specified. Either too many or too few basis variables are specified.

MSK_RES_ERR_INV_SKC (1267)

Invalid value in `skc`.

MSK_RES_ERR_INV_SKX (1268)

Invalid value in `skx`.

MSK_RES_ERR_INV_SKN (1274)

Invalid value in `skn`.

MSK_RES_ERR_INV_SK_STR (1269)

Invalid status key string encountered.

MSK_RES_ERR_INV_SK (1270)

Invalid status key code.

MSK_RES_ERR_INV_CONE_TYPE_STR (1271)

Invalid cone type string encountered.

MSK_RES_ERR_INV_CONE_TYPE (1272)

Invalid cone type code is encountered.

MSK_RES_ERR_INVALID_SURPLUS (1275)

Invalid surplus.

MSK_RES_ERR_INV_NAME_ITEM (1280)

An invalid name item code is used.

MSK_RES_ERR_PRO_ITEM (1281)

An invalid problem is used.

MSK_RES_ERR_INVALID_FORMAT_TYPE (1283)

Invalid format type.

MSK_RES_ERR_FIRSTI (1285)

Invalid `firsti`.

MSK_RES_ERR_LASTI (1286)

Invalid `lasti`.

MSK_RES_ERR_FIRSTJ (1287)

Invalid `firstj`.

MSK_RES_ERR_LASTJ (1288)

Invalid `lastj`.

MSK_RES_ERR_MAX_LEN_IS_TOO_SMALL (1289)

A maximum length that is too small has been specified.

MSK_RES_ERR_NONLINEAR_EQUALITY (1290)

The model contains a nonlinear equality which defines a nonconvex set.

MSK_RES_ERR_NONCONVEX (1291)

The optimization problem is nonconvex.

MSK_RES_ERR_NONLINEAR_RANGED (1292)

Nonlinear constraints with finite lower and upper bound always define a nonconvex feasible set.

MSK_RES_ERR_CON_Q_NOT_PSD (1293)

The quadratic constraint matrix is not positive semidefinite as expected for a constraint with finite upper bound. This results in a nonconvex problem.

MSK_RES_ERR_CON_Q_NOT_NSD (1294)

The quadratic constraint matrix is not negative semidefinite as expected for a constraint with finite lower bound. This results in a nonconvex problem.

MSK_RES_ERR_OBJ_Q_NOT_PSD (1295)

The quadratic coefficient matrix in the objective is not positive semidefinite as expected for a minimization problem.

MSK_RES_ERR_OBJ_Q_NOT_NSD (1296)

The quadratic coefficient matrix in the objective is not negative semidefinite as expected for a maximization problem.

MSK_RES_ERR_ARGUMENT_PERM_ARRAY (1299)

An invalid permutation array is specified.

MSK_RES_ERR_CONE_INDEX (1300)

An index of a non-existing cone has been specified.

MSK_RES_ERR_CONE_SIZE (1301)

A cone with incorrect number of members is specified.

MSK_RES_ERR_CONE_OVERLAP (1302)

One or more of the variables in the cone to be added is already member of another cone. Now assume the variable is x_j then add a new variable say x_k and the constraint

$$x_j = x_k$$

and then let x_k be member of the cone to be appended.

MSK_RES_ERR_CONE_REP_VAR (1303)

A variable is included multiple times in the cone.

MSK_RES_ERR_MAXNUMCONE (1304)

The value specified for `maxnumcone` is too small.

MSK_RES_ERR_CONE_TYPE (1305)

Invalid cone type specified.

MSK_RES_ERR_CONE_TYPE_STR (1306)

Invalid cone type specified.

MSK_RES_ERR_CONE_OVERLAP_APPEND (1307)

The cone to be appended has one variable which is already member of another cone.

MSK_RES_ERR_REMOVE_CONE_VARIABLE (1310)

A variable cannot be removed because it will make a cone invalid.

MSK_RES_ERR_APPENDING_TOO_BIG_CONE (1311)

Trying to append a too big cone.

MSK_RES_ERR_CONE_PARAMETER (1320)

An invalid cone parameter.

MSK_RES_ERR_SOL_FILE_INVALID_NUMBER (1350)

An invalid number is specified in a solution file.

MSK_RES_ERR_HUGE_C (1375)

A huge value in absolute size is specified for one c_j.

MSK_RES_ERR_HUGE_AIJ (1380)

A numerically huge value is specified for an $a_{i,j}$ element in A. The parameter *MSK_DPAR_DATA_TOL_AIJ_HUGE* controls when an $a_{i,j}$ is considered huge.

MSK_RES_ERR_DUPLICATE_AIJ (1385)

An element in the A matrix is specified twice.

MSK_RES_ERR_LOWER_BOUND_IS_A_NAN (1390)

The lower bound specified is not a number (nan) or is not finite.

MSK_RES_ERR_UPPER_BOUND_IS_A_NAN (1391)

The upper bound specified is not a number (nan) or is not finite.

MSK_RES_ERR_INFINITE_BOUND (1400)

A numerically huge bound value is specified.

MSK_RES_ERR_INV_QOBJ_SUBI (1401)

Invalid value in `qosubi`.

MSK_RES_ERR_INV_QOBJ_SUBJ (1402)

Invalid value in `qosubj`.

MSK_RES_ERR_INV_QOBJ_VAL (1403)

Invalid value in `qoval`.

MSK_RES_ERR_INV_QCON_SUBK (1404)

Invalid value in `qcsubk`.

MSK_RES_ERR_INV_QCON_SUBI (1405)

Invalid value in `qcsubi`.

MSK_RES_ERR_INV_QCON_SUBJ (1406)

Invalid value in `qcsubj`.

MSK_RES_ERR_INV_QCON_VAL (1407)

Invalid value in `qcval`.

MSK_RES_ERR_QCON_SUBI_TOO_SMALL (1408)

Invalid value in `qcsubi`.

MSK_RES_ERR_QCON_SUBI_TOO_LARGE (1409)

Invalid value in `qcsubi`.

MSK_RES_ERR_QOBJ_UPPER_TRIANGLE (1415)

An element in the upper triangle of Q^o is specified. Only elements in the lower triangle should be specified.

MSK_RES_ERR_QCON_UPPER_TRIANGLE (1417)

An element in the upper triangle of a Q^k is specified. Only elements in the lower triangle should be specified.

MSK_RES_ERR_FIXED_BOUND_VALUES (1420)

A fixed constraint/variable has been specified using the bound keys but the numerical value of the lower and upper bound is different.

MSK_RES_ERR_TOO_SMALL_A_TRUNCATION_VALUE (1421)

A too small value for the A trucation value is specified.

MSK_RES_ERR_INVALID_OBJECTIVE_SENSE (1445)

An invalid objective sense is specified.

MSK_RES_ERR_UNDEFINED_OBJECTIVE_SENSE (1446)

The objective sense has not been specified before the optimization.

MSK_RES_ERR_Y_IS_UNDEFINED (1449)

The solution item y is undefined.

MSK_RES_ERR_NAN_IN_DOUBLE_DATA (1450)

An invalid floating point value was used in some double data.

MSK_RES_ERR_INF_IN_DOUBLE_DATA (1451)

An infinite floating point value was used in some double data.

MSK_RES_ERR_NAN_IN_BLC (1461)

l^c contains an invalid floating point value, i.e. a NaN or Inf.

MSK_RES_ERR_NAN_IN_BUC (1462)

u^c contains an invalid floating point value, i.e. a NaN of Inf.

MSK_RES_ERR_INVALID_CFIX (1469)

An invalid fixed term in the objective is speficied.

MSK_RES_ERR_NAN_IN_C (1470)

c contains an invalid floating point value, i.e. a NaN or Inf.

MSK_RES_ERR_NAN_IN_BLX (1471)

l^x contains an invalid floating point value, i.e. a NaN or Inf.

MSK_RES_ERR_NAN_IN_BUX (1472)

u^x contains an invalid floating point value, i.e. a NaN or Inf.

MSK_RES_ERR_INVALID_AIJ (1473)

$a_{i,j}$ contains an invalid floating point value, i.e. a NaN or an infinite value.

MSK_RES_ERR_INVALID_CJ (1474)

c_j contains an invalid floating point value, i.e. a NaN or an infinite value.

MSK_RES_ERR_SYM_MAT_INVALID (1480)

A symmetric matrix contains an invalid floating point value, i.e. a NaN or an infinite value.

MSK_RES_ERR_SYM_MAT_HUGE (1482)

A symmetric matrix contains a huge value in absolute size. The parameter *MSK_DPAR_DATA_SYM_MAT_TOL_HUGE* controls when an $e_{i,j}$ is considered huge.

MSK_RES_ERR_INV_PROBLEM (1500)

Invalid problem type. Probably a nonconvex problem has been specified.

MSK_RES_ERR_MIXED_CONIC_AND_NL (1501)

The problem contains nonlinear terms conic constraints. The requested operation cannot be applied to this type of problem.

MSK_RES_ERR_GLOBAL_INV_CONIC_PROBLEM (1503)

The global optimizer can only be applied to problems without semidefinite variables.

MSK_RES_ERR_INV_OPTIMIZER (1550)

An invalid optimizer has been chosen for the problem.

MSK_RES_ERR_MIO_NO_OPTIMIZER (1551)

No optimizer is available for the current class of integer optimization problems.

MSK_RES_ERR_NO_OPTIMIZER_VAR_TYPE (1552)

No optimizer is available for this class of optimization problems.

MSK_RES_ERR_FINAL_SOLUTION (1560)

An error occurred during the solution finalization.

MSK_RES_ERR_FIRST (1570)

Invalid first.

MSK_RES_ERR_LAST (1571)

Invalid index last. A given index was out of expected range.

MSK_RES_ERR_SLICE_SIZE (1572)

Invalid slice size specified.

MSK_RES_ERR_NEGATIVE_SURPLUS (1573)

Negative surplus.

MSK_RES_ERR_NEGATIVE_APPEND (1578)

Cannot append a negative number.

MSK_RES_ERR_POSTSOLVE (1580)

An error occurred during the postsolve. Please contact **MOSEK** support.

MSK_RES_ERR_OVERFLOW (1590)

A computation produced an overflow i.e. a very large number.

`MSK_RES_ERR_NO_BASIS_SOL (1600)`

No basic solution is defined.

`MSK_RES_ERR_BASIS_FACTOR (1610)`

The factorization of the basis is invalid.

`MSK_RES_ERR_BASIS_SINGULAR (1615)`

The basis is singular and hence cannot be factored.

`MSK_RES_ERR_FACTOR (1650)`

An error occurred while factorizing a matrix.

`MSK_RES_ERR_FEASREPAIR_CANNOT_RELAX (1700)`

An optimization problem cannot be relaxed.

`MSK_RES_ERR_FEASREPAIR_SOLVING_RELAXED (1701)`

The relaxed problem could not be solved to optimality. Please consult the log file for further details.

`MSK_RES_ERR_FEASREPAIR_INCONSISTENT_BOUND (1702)`

The upper bound is less than the lower bound for a variable or a constraint. Please correct this before running the feasibility repair.

`MSK_RES_ERR_REPAIR_INVALID_PROBLEM (1710)`

The feasibility repair does not support the specified problem type.

`MSK_RES_ERR_REPAIR_OPTIMIZATION_FAILED (1711)`

Computation the optimal relaxation failed. The cause may have been numerical problems.

`MSK_RES_ERR_NAME_MAX_LEN (1750)`

A name is longer than the buffer that is supposed to hold it.

`MSK_RES_ERR_NAME_IS_NULL (1760)`

The name buffer is a `nothing` pointer.

`MSK_RES_ERR_INVALID_COMPRESSION (1800)`

Invalid compression type.

`MSK_RES_ERR_INVALID_IOMODE (1801)`

Invalid io mode.

`MSK_RES_ERR_NO_PRIMAL_INFEAS_CER (2000)`

A certificate of primal infeasibility is not available.

`MSK_RES_ERR_NO_DUAL_INFEAS_CER (2001)`

A certificate of infeasibility is not available.

`MSK_RES_ERR_NO_SOLUTION_IN_CALLBACK (2500)`

The required solution is not available.

`MSK_RES_ERR_INV_MARKI (2501)`

Invalid value in marki.

`MSK_RES_ERR_INV_MARKJ (2502)`

Invalid value in markj.

`MSK_RES_ERR_INV_NUMI (2503)`

Invalid numi.

`MSK_RES_ERR_INV_NUMJ (2504)`

Invalid numj.

`MSK_RES_ERR_TASK_INCOMPATIBLE (2560)`

The Task file is incompatible with this platform. This results from reading a file on a 32 bit platform generated on a 64 bit platform.

`MSK_RES_ERR_TASK_INVALID (2561)`

The Task file is invalid.

`MSK_RES_ERR_TASK_WRITE (2562)`

Failed to write the task file.

`MSK_RES_ERR_READ_WRITE (2563)`

Failed to read or write due to an I/O error.

`MSK_RES_ERR_TASK_PREMATURE_EOF (2564)`

The Task file ended prematurely.

MSK_RES_ERR_LU_MAX_NUM_TRIES (2800)

Could not compute the LU factors of the matrix within the maximum number of allowed tries.

MSK_RES_ERR_INVALID_UTF8 (2900)

An invalid UTF8 string is encountered.

MSK_RES_ERR_INVALID_WCHAR (2901)

An invalid `wchar` string is encountered.

MSK_RES_ERR_NO_DUAL_FOR_ITG_SOL (2950)

No dual information is available for the integer solution.

MSK_RES_ERR_NO_SNX_FOR_BAS_SOL (2953)

s_n^x is not available for the basis solution.

MSK_RES_ERR_INTERNAL (3000)

An internal error occurred. Please report this problem.

MSK_RES_ERR_API_ARRAY_TOO_SMALL (3001)

An input array was too short.

MSK_RES_ERR_API_CB_CONNECT (3002)

Failed to connect a callback object.

MSK_RES_ERR_API_FATAL_ERROR (3005)

An internal error occurred in the API. Please report this problem.

MSK_RES_ERR_API_INTERNAL (3999)

An internal fatal error occurred in an interface function.

MSK_RES_ERR_SEN_FORMAT (3050)

Syntax error in sensitivity analysis file.

MSK_RES_ERR_SEN_UNDEF_NAME (3051)

An undefined name was encountered in the sensitivity analysis file.

MSK_RES_ERR_SEN_INDEX_RANGE (3052)

Index out of range in the sensitivity analysis file.

MSK_RES_ERR_SEN_BOUND_INVALID_UP (3053)

Analysis of upper bound requested for an index, where no upper bound exists.

MSK_RES_ERR_SEN_BOUND_INVALID_LO (3054)

Analysis of lower bound requested for an index, where no lower bound exists.

MSK_RES_ERR_SEN_INDEX_INVALID (3055)

Invalid range given in the sensitivity file.

MSK_RES_ERR_SEN_INVALID_REGEXP (3056)

Syntax error in regexp or regexp longer than 1024.

MSK_RES_ERR_SEN_SOLUTION_STATUS (3057)

No optimal solution found to the original problem given for sensitivity analysis.

MSK_RES_ERR_SEN_NUMERICAL (3058)

Numerical difficulties encountered performing the sensitivity analysis.

MSK_RES_ERR_SEN_UNHANDLED_PROBLEM_TYPE (3080)

Sensitivity analysis cannot be performed for the specified problem. Sensitivity analysis is only possible for linear problems.

MSK_RES_ERR_UNB_STEP_SIZE (3100)

A step size in an optimizer was unexpectedly unbounded. For instance, if the step-size becomes unbounded in phase 1 of the simplex algorithm then an error occurs. Normally this will happen only if the problem is badly formulated. Please contact **MOSEK** support if this error occurs.

MSK_RES_ERR_IDENTICAL_TASKS (3101)

Some tasks related to this function call were identical. Unique tasks were expected.

MSK_RES_ERR_AD_INVALID_CODELIST (3102)

The code list data was invalid.

MSK_RES_ERR_INTERNAL_TEST_FAILED (3500)

An internal unit test function failed.

MSK_RES_ERR_INT64_TO_INT32_CAST (3800)

A 64 bit integer could not be cast to a 32 bit integer.

MSK_RES_ERR_INFEAS_UNDEFINED (3910)

The requested value is not defined for this solution type.

MSK_RES_ERR_NO_BARX_FOR_SOLUTION (3915)

There is no $\overline{X}$ available for the solution specified. In particular note there are no $\overline{X}$ defined for the basic and integer solutions.

MSK_RES_ERR_NO_BARS_FOR_SOLUTION (3916)

There is no $\bar{s}$ available for the solution specified. In particular note there are no $\bar{s}$ defined for the basic and integer solutions.

MSK_RES_ERR_BAR_VAR_DIM (3920)

The dimension of a symmetric matrix variable has to be greater than 0.

MSK_RES_ERR_SYM_MAT_INVALID_ROW_INDEX (3940)

A row index specified for sparse symmetric matrix is invalid.

MSK_RES_ERR_SYM_MAT_INVALID_COL_INDEX (3941)

A column index specified for sparse symmetric matrix is invalid.

MSK_RES_ERR_SYM_MAT_NOT_LOWER_TRINGULAR (3942)

Only the lower triangular part of sparse symmetric matrix should be specified.

MSK_RES_ERR_SYM_MAT_INVALID_VALUE (3943)

The numerical value specified in a sparse symmetric matrix is not a floating point value.

MSK_RES_ERR_SYM_MAT_DUPLICATE (3944)

A value in a symmetric matric as been specified more than once.

MSK_RES_ERR_INVALID_SYM_MAT_DIM (3950)

A sparse symmetric matrix of invalid dimension is specified.

MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_SYM_MAT (4000)

The file format does not support a problem with symmetric matrix variables.

MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_CFIX (4001)

The file format does not support a problem with nonzero fixed term in c.

MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_RANGED_CONSTRAINTS (4002)

The file format does not support a problem with ranged constraints.

MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_FREE_CONSTRAINTS (4003)

The file format does not support a problem with free constraints.

MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_CONES (4005)

The file format does not support a problem with the simple cones (deprecated).

MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_QUADRATIC_TERMS (4006)

The file format does not support a problem with quadratic terms.

MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_NONLINEAR (4010)

The file format does not support a problem with nonlinear terms.

MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_DISJUNCTIVE_CONSTRAINTS (4011)

The file format does not support a problem with disjunctive constraints.

MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_AFFINE_CONIC_CONSTRAINTS (4012)

The file format does not support a problem with affine conic constraints.

MSK_RES_ERR_DUPLICATE_CONSTRAINT_NAMES (4500)

Two constraint names are identical.

MSK_RES_ERR_DUPLICATE_VARIABLE_NAMES (4501)

Two variable names are identical.

MSK_RES_ERR_DUPLICATE_BARVARIABLE_NAMES (4502)

Two barvariable names are identical.

MSK_RES_ERR_DUPLICATE_CONE_NAMES (4503)

Two cone names are identical.

MSK_RES_ERR_DUPLICATE_DOMAIN_NAMES (4504)

Two domain names are identical.

`MSK_RES_ERR_DUPLICATE_DJC_NAMES (4505)`

Two disjunctive constraint names are identical.

`MSK_RES_ERR_NON_UNIQUE_ARRAY (5000)`

An array does not contain unique elements.

`MSK_RES_ERR_ARGUMENT_IS_TOO_SMALL (5004)`

The value of a function argument is too small.

`MSK_RES_ERR_ARGUMENT_IS_TOO_LARGE (5005)`

The value of a function argument is too large.

`MSK_RES_ERR_MIO_INTERNAL (5010)`

A fatal error occurred in the mixed integer optimizer. Please contact **MOSEK** support.

`MSK_RES_ERR_INVALID_PROBLEM_TYPE (6000)`

An invalid problem type.

`MSK_RES_ERR_UNHANDLED_SOLUTION_STATUS (6010)`

Unhandled solution status.

`MSK_RES_ERR_UPPER_TRIANGLE (6020)`

An element in the upper triangle of a lower triangular matrix is specified.

`MSK_RES_ERR_LAU_SINGULAR_MATRIX (7000)`

A matrix is singular.

`MSK_RES_ERR_LAU_NOT_POSITIVE_DEFINITE (7001)`

A matrix is not positive definite.

`MSK_RES_ERR_LAU_INVALID_LOWER_TRIANGULAR_MATRIX (7002)`

An invalid lower triangular matrix.

`MSK_RES_ERR_LAU_UNKNOWN (7005)`

An unknown error.

`MSK_RES_ERR_LAU_ARG_M (7010)`

Invalid argument m.

`MSK_RES_ERR_LAU_ARG_N (7011)`

Invalid argument n.

`MSK_RES_ERR_LAU_ARG_K (7012)`

Invalid argument k.

`MSK_RES_ERR_LAU_ARG_TRANSA (7015)`

Invalid argument transa.

`MSK_RES_ERR_LAU_ARG_TRANSB (7016)`

Invalid argument transb.

`MSK_RES_ERR_LAU_ARG_UPLO (7017)`

Invalid argument uplo.

`MSK_RES_ERR_LAU_ARG_TRANS (7018)`

Invalid argument trans.

`MSK_RES_ERR_LAU_INVALID_SPARSE_SYMMETRIC_MATRIX (7019)`

An invalid sparse symmetric matrix is specfified. Note only the lower triangular part with no duplicates is specifed.

`MSK_RES_ERR_CBF_PARSE (7100)`

An error occurred while parsing an CBF file.

`MSK_RES_ERR_CBF_OBJ_SENSE (7101)`

An invalid objective sense is specified.

`MSK_RES_ERR_CBF_NO_VARIABLES (7102)`

No variables are specified.

`MSK_RES_ERR_CBF_TOO_MANY_CONSTRAINTS (7103)`

Too many constraints specified.

`MSK_RES_ERR_CBF_TOO_MANY_VARIABLES (7104)`

Too many variables specified.

MSK_RES_ERR_CBF_NO_VERSION_SPECIFIED (7105)

No version specified.

MSK_RES_ERR_CBF_SYNTAX (7106)

Invalid syntax.

MSK_RES_ERR_CBF_DUPLICATE_OBJ (7107)

Duplicate OBJ keyword.

MSK_RES_ERR_CBF_DUPLICATE_CON (7108)

Duplicate CON keyword.

MSK_RES_ERR_CBF_DUPLICATE_VAR (7110)

Duplicate VAR keyword.

MSK_RES_ERR_CBF_DUPLICATE_INT (7111)

Duplicate INT keyword.

MSK_RES_ERR_CBF_INVALID_VAR_TYPE (7112)

Invalid variable type.

MSK_RES_ERR_CBF_INVALID_CON_TYPE (7113)

Invalid constraint type.

MSK_RES_ERR_CBF_INVALID_DOMAIN_DIMENSION (7114)

Invalid domain dimension.

MSK_RES_ERR_CBF_DUPLICATE_OBJACOORD (7115)

Duplicate index in OBJCOORD.

MSK_RES_ERR_CBF_DUPLICATE_BCOORD (7116)

Duplicate index in BCOORD.

MSK_RES_ERR_CBF_DUPLICATE_ACOORD (7117)

Duplicate index in ACOORD.

MSK_RES_ERR_CBF_TOO_FEW_VARIABLES (7118)

Too few variables defined.

MSK_RES_ERR_CBF_TOO_FEW_CONSTRAINTS (7119)

Too few constraints defined.

MSK_RES_ERR_CBF_TOO_FEW_INTS (7120)

Too few ints are specified.

MSK_RES_ERR_CBF_TOO_MANY_INTS (7121)

Too many ints are specified.

MSK_RES_ERR_CBF_INVALID_INT_INDEX (7122)

Invalid INT index.

MSK_RES_ERR_CBF_UNSUPPORTED (7123)

Unsupported feature is present.

MSK_RES_ERR_CBF_DUPLICATE_PSDVAR (7124)

Duplicate PSDVAR keyword.

MSK_RES_ERR_CBF_INVALID_PSDVAR_DIMENSION (7125)

Invalid PSDVAR dimension.

MSK_RES_ERR_CBF_TOO_FEW_PSDVAR (7126)

Too few variables defined.

MSK_RES_ERR_CBF_INVALID_EXP_DIMENSION (7127)

Invalid dimension of a exponential cone.

MSK_RES_ERR_CBF_DUPLICATE_POW_CONES (7130)

Multiple POWCONES specified.

MSK_RES_ERR_CBF_DUPLICATE_POW_STAR_CONES (7131)

Multiple POW*CONES specified.

MSK_RES_ERR_CBF_INVALID_POWER (7132)

Invalid power specified.

MSK_RES_ERR_CBF_POWER_CONE_IS_TOO_LONG (7133)

Power cone is too long.

MSK_RES_ERR_CBF_INVALID_POWER_CONE_INDEX (7134)

Invalid power cone index.

MSK_RES_ERR_CBF_INVALID_POWER_STAR_CONE_INDEX (7135)

Invalid power star cone index.

MSK_RES_ERR_CBF_UNHANDLED_POWER_CONE_TYPE (7136)

An unhandled power cone type.

MSK_RES_ERR_CBF_UNHANDLED_POWER_STAR_CONE_TYPE (7137)

An unhandled power star cone type.

MSK_RES_ERR_CBF_POWER_CONE_MISMATCH (7138)

The power cone does not match with it definition.

MSK_RES_ERR_CBF_POWER_STAR_CONE_MISMATCH (7139)

The power star cone does not match with it definition.

MSK_RES_ERR_CBF_INVALID_NUMBER_OF_CONES (7140)

Invalid number of cones.

MSK_RES_ERR_CBF_INVALID_DIMENSION_OF_CONES (7141)

Invalid number of cones.

MSK_RES_ERR_CBF_INVALID_NUM_OBJACOORD (7150)

Invalid number of OBJACOORD.

MSK_RES_ERR_CBF_INVALID_NUM_OBJFCOORD (7151)

Invalid number of OBJFCOORD.

MSK_RES_ERR_CBF_INVALID_NUM_ACOORD (7152)

Invalid number of ACOORD.

MSK_RES_ERR_CBF_INVALID_NUM_BCOORD (7153)

Invalid number of BCOORD.

MSK_RES_ERR_CBF_INVALID_NUM_FCOORD (7155)

Invalid number of FCOORD.

MSK_RES_ERR_CBF_INVALID_NUM_HCOORD (7156)

Invalid number of HCOORD.

MSK_RES_ERR_CBF_INVALID_NUM_DCOORD (7157)

Invalid number of DCOORD.

MSK_RES_ERR_CBF_EXPECTED_A_KEYWORD (7158)

Expected a key word.

MSK_RES_ERR_CBF_INVALID_NUM_PSDCON (7200)

Invalid number of PSDCON.

MSK_RES_ERR_CBF_DUPLICATE_PSDCON (7201)

Duplicate CON keyword.

MSK_RES_ERR_CBF_INVALID_DIMENSION_OF_PSDCON (7202)

Invalid PSDCON dimension.

MSK_RES_ERR_CBF_INVALID_PSDCON_INDEX (7203)

Invalid PSDCON index.

MSK_RES_ERR_CBF_INVALID_PSDCON_VARIABLE_INDEX (7204)

Invalid PSDCON index.

MSK_RES_ERR_CBF_INVALID_PSDCON_BLOCK_INDEX (7205)

Invalid PSDCON index.

MSK_RES_ERR_CBF_UNSUPPORTED_CHANGE (7210)

The CHANGE section is not supported.

MSK_RES_ERR_MIO_INVALID_ROOT_OPTIMIZER (7700)

An invalid root optimizer was selected for the problem type.

MSK_RES_ERR_MIO_INVALID_NODE_OPTIMIZER (7701)

An invalid node optimizer was selected for the problem type.

MSK_RES_ERR_MPS_WRITE_CPLEX_INVALID_CONE_TYPE (7750)

An invalid cone type occurs when writing a CPLEX formatted MPS file.

MSK_RES_ERR_TOCONIC_CONSTR_Q_NOT_PSD (7800)

The matrix defining the quadratric part of constraint is not positive semidefinite.

MSK_RES_ERR_TOCONIC_CONSTRAINT_FX (7801)

The quadratic constraint is an equality, thus not convex.

MSK_RES_ERR_TOCONIC_CONSTRAINT_RA (7802)

The quadratic constraint has finite lower and upper bound, and therefore it is not convex.

MSK_RES_ERR_TOCONIC_CONSTR_NOT_CONIC (7803)

The constraint is not conic representable.

MSK_RES_ERR_TOCONIC_OBJECTIVE_NOT_PSD (7804)

The matrix defining the quadratric part of the objective function is not positive semidefinite.

MSK_RES_ERR_GETDUAL_NOT_AVAILABLE (7820)

The simple dualizer is not available for this problem class.

MSK_RES_ERR_SERVER_CONNECT (8000)

Failed to connect to remote solver server. The server string or the port string were invalid, or the server did not accept connection.

MSK_RES_ERR_SERVER_PROTOCOL (8001)

Unexpected message or data from solver server.

MSK_RES_ERR_SERVER_STATUS (8002)

Server returned non-ok HTTP status code

MSK_RES_ERR_SERVER_TOKEN (8003)

The job ID specified is incorrect or invalid

MSK_RES_ERR_SERVER_ADDRESS (8004)

Invalid address string

MSK_RES_ERR_SERVER_CERTIFICATE (8005)

Invalid TLS certificate format or path

MSK_RES_ERR_SERVER_TLS_CLIENT (8006)

Failed to create TLS cleint

MSK_RES_ERR_SERVER_ACCESS_TOKEN (8007)

Invalid access token

MSK_RES_ERR_SERVER_PROBLEM_SIZE (8008)

The size of the problem exceeds the dimensions permitted by the instance of the OptServer where it was run.

MSK_RES_ERR_SERVER_HARD_TIMEOUT (8009)

The hard timeout limit was reached on solver server, and the solver process was killed

MSK_RES_ERR_DUPLICATE_INDEX_IN_A_SPARSE_MATRIX (20050)

An element in a sparse matrix is specified twice.

MSK_RES_ERR_DUPLICATE_INDEX_IN_AFEIDX_LIST (20060)

An index is specified twice in an affine expression list.

MSK_RES_ERR_DUPLICATE_FIJ (20100)

An element in the F matrix is specified twice.

MSK_RES_ERR_INVALID_FIJ (20101)

$f_{i,j}$ contains an invalid floating point value, i.e. a NaN or an infinite value.

MSK_RES_ERR_HUGE_FIJ (20102)

A numerically huge value is specified for an $f_{i,j}$ element in F. The parameter *MSK_DPAR_DATA_TOL_AIJ_HUGE* controls when an $f_{i,j}$ is considered huge.

MSK_RES_ERR_INVALID_G (20103)

g contains an invalid floating point value, i.e. a NaN or an infinite value.

MSK_RES_ERR_INVALID_B (20150)

b contains an invalid floating point value, i.e. a NaN or an infinite value.

MSK_RES_ERR_DOMAIN_INVALID_INDEX (20400)

A domain index is invalid.

MSK_RES_ERR_DOMAIN_DIMENSION (20401)

A domain dimension is invalid.

MSK_RES_ERR_DOMAIN_DIMENSION_PSD (20402)

A PSD domain dimension is invalid.

MSK_RES_ERR_NOT_POWER_DOMAIN (20403)

The function is only applicable to primal and dual power cone domains.

MSK_RES_ERR_DOMAIN_POWER_INVALID_ALPHA (20404)

Alpha contains an invalid floating point value, i.e. a NaN or an infinite value.

MSK_RES_ERR_DOMAIN_POWER_NEGATIVE_ALPHA (20405)

Alpha contains a negative value or zero.

MSK_RES_ERR_DOMAIN_POWER_NLEFT (20406)

The value of n_{left} is not in $[1, n-1]$ where n is the dimension.

MSK_RES_ERR_AFE_INVALID_INDEX (20500)

An affine expression index is invalid.

MSK_RES_ERR_ACC_INVALID_INDEX (20600)

A affine conic constraint index is invalid.

MSK_RES_ERR_ACC_INVALID_ENTRY_INDEX (20601)

The index of an element in an affine conic constraint is invalid.

MSK_RES_ERR_ACC_AFE_DOMAIN_MISMATCH (20602)

There is a mismatch between between the number of affine expressions and total dimension of the domain(s).

MSK_RES_ERR_DJC_INVALID_INDEX (20700)

A disjunctive constraint index is invalid.

MSK_RES_ERR_DJC_UNSUPPORTED_DOMAIN_TYPE (20701)

An unsupported domain type has been used in a disjunctive constraint.

MSK_RES_ERR_DJC_AFE_DOMAIN_MISMATCH (20702)

There is a mismatch between the number of affine expressions and total dimension of the domain(s).

MSK_RES_ERR_DJC_INVALID_TERM_SIZE (20703)

A termize is invalid.

MSK_RES_ERR_DJC_DOMAIN_TERMSIZE_MISMATCH (20704)

There is a mismatch between the number of domains and the term sizes.

MSK_RES_ERR_DJC_TOTAL_NUM_TERMS_MISMATCH (20705)

There total number of terms in all domains does not match.

MSK_RES_ERR_UNDEF_SOLUTION (22000)

MOSEK has the following solution types:

- an interior-point solution,
- a basic solution,
- and an integer solution.

Each optimizer may set one or more of these solutions; e.g by default a successful optimization with the interior-point optimizer defines the interior-point solution and, for linear problems, also the basic solution. This error occurs when asking for a solution or for information about a solution that is not defined.

MSK_RES_ERR_NO_DOTY (22010)

No doty is available

15.7 Enumerations

`basindtype`

Basis identification

`MSK_BI_NEVER`

Never do basis identification.

`MSK_BI_ALWAYS`

Basis identification is always performed even if the interior-point optimizer terminates abnormally.

`MSK_BI_NO_ERROR`

Basis identification is performed if the interior-point optimizer terminates without an error.

`MSK_BI_IF_FEASIBLE`

Basis identification is not performed if the interior-point optimizer terminates with a problem status saying that the problem is primal or dual infeasible.

`MSK_BI_RESERVERED`

Not currently in use.

`boundkey`

Bound keys

`MSK_BK_LO`

The constraint or variable has a finite lower bound and an infinite upper bound.

`MSK_BK_UP`

The constraint or variable has an infinite lower bound and an finite upper bound.

`MSK_BK_FX`

The constraint or variable is fixed.

`MSK_BK_FR`

The constraint or variable is free.

`MSK_BK_RA`

The constraint or variable is ranged.

`mark`

Mark

`MSK_MARK_LO`

The lower bound is selected for sensitivity analysis.

`MSK_MARK_UP`

The upper bound is selected for sensitivity analysis.

`simprecision`

Experimental. Usage not recommended.

`MSK_SIM_PRECISION_NORMAL`

Experimental. Usage not recommended.

`MSK_SIM_PRECISION_EXTENDED`

Experimental. Usage not recommended.

`simdegen`

Degeneracy strategies

`MSK_SIM_DEGEN_NONE`

The simplex optimizer should use no degeneration strategy.

`MSK_SIM_DEGEN_FREE`

The simplex optimizer chooses the degeneration strategy.

MSK_SIM_DEGEN_AGGRESSIVE

The simplex optimizer should use an aggressive degeneration strategy.

MSK_SIM_DEGEN_MODERATE

The simplex optimizer should use a moderate degeneration strategy.

MSK_SIM_DEGEN_MINIMUM

The simplex optimizer should use a minimum degeneration strategy.

transpose

Transposed matrix.

MSK_TRANSPOSE_NO

No transpose is applied.

MSK_TRANSPOSE_YES

A transpose is applied.

uplo

Triangular part of a symmetric matrix.

MSK_UPLO_LO

Lower part.

MSK_UPLO_UP

Upper part.

simreform

Problem reformulation.

MSK_SIM_REFORMULATION_ON

Allow the simplex optimizer to reformulate the problem.

MSK_SIM_REFORMULATION_OFF

Disallow the simplex optimizer to reformulate the problem.

MSK_SIM_REFORMULATION_FREE

The simplex optimizer can choose freely.

MSK_SIM_REFORMULATION_AGGRESSIVE

The simplex optimizer should use an aggressive reformulation strategy.

simdupvec

Exploit duplicate columns.

MSK_SIM_EXPLOIT_DUPVEC_ON

Allow the simplex optimizer to exploit duplicated columns.

MSK_SIM_EXPLOIT_DUPVEC_OFF

Disallow the simplex optimizer to exploit duplicated columns.

MSK_SIM_EXPLOIT_DUPVEC_FREE

The simplex optimizer can choose freely.

simhotstart

Hot-start type employed by the simplex optimizer

MSK_SIM_HOTSTART_NONE

The simplex optimizer performs a coldstart.

MSK_SIM_HOTSTART_FREE

The simplex optimize chooses the hot-start type.

MSK_SIM_HOTSTART_STATUS_KEYS

Only the status keys of the constraints and variables are used to choose the type of hot-start.

intpnthotstart

Hot-start type employed by the interior-point optimizers.

MSK_INTPNT_HOTSTART_NONE

The interior-point optimizer performs a coldstart.

MSK_INTPNT_HOTSTART_PRIMAL

The interior-point optimizer exploits the primal solution only.

MSK_INTPNT_HOTSTART_DUAL

The interior-point optimizer exploits the dual solution only.

MSK_INTPNT_HOTSTART_PRIMAL_DUAL

The interior-point optimizer exploits both the primal and dual solution.

callbackcode

Progress callback codes

MSK_CALLBACK_BEGIN_BI

The basis identification procedure has been started.

MSK_CALLBACK_BEGIN_CONIC

The callback function is called when the conic optimizer is started.

MSK_CALLBACK_BEGIN_DUAL_BI

The callback function is called from within the basis identification procedure when the dual phase is started.

MSK_CALLBACK_BEGIN_DUAL_SENSITIVITY

Dual sensitivity analysis is started.

MSK_CALLBACK_BEGIN_DUAL_SETUP_BI

The callback function is called when the dual BI phase is started.

MSK_CALLBACK_BEGIN_DUAL_SIMPLEX

The callback function is called when the dual simplex optimizer started.

MSK_CALLBACK_BEGIN_DUAL_SIMPLEX_BI

The callback function is called from within the basis identification procedure when the dual simplex clean-up phase is started.

MSK_CALLBACK_BEGIN_FOLDING

The calback function is called at the beginning of folding.

MSK_CALLBACK_BEGIN_FOLDING_BI

TBD

MSK_CALLBACK_BEGIN_FOLDING_BI_DUAL

TBD

MSK_CALLBACK_BEGIN_FOLDING_BI_INITIALIZE

TBD

MSK_CALLBACK_BEGIN_FOLDING_BI_OPTIMIZER

TBD

MSK_CALLBACK_BEGIN_FOLDING_BI_PRIMAL

TBD

MSK_CALLBACK_BEGIN_INFEAS_ANA

The callback function is called when the infeasibility analyzer is started.

MSK_CALLBACK_BEGIN_INITIALIZE_BI

The callback function is called from within the basis identification procedure when the initialization phase is started.

MSK_CALLBACK_BEGIN_INTPNT

The callback function is called when the interior-point optimizer is started.

`MSK_CALLBACK_BEGIN_LICENSE_WAIT`

Begin waiting for license.

`MSK_CALLBACK_BEGIN_MIO`

The callback function is called when the mixed-integer optimizer is started.

`MSK_CALLBACK_BEGIN_OPTIMIZE_BI`

TBD.

`MSK_CALLBACK_BEGIN_OPTIMIZER`

The callback function is called when the optimizer is started.

`MSK_CALLBACK_BEGIN_PRESOLVE`

The callback function is called when the presolve is started.

`MSK_CALLBACK_BEGIN_PRIMAL_BI`

The callback function is called from within the basis identification procedure when the primal phase is started.

`MSK_CALLBACK_BEGIN_PRIMAL_REPAIR`

Begin primal feasibility repair.

`MSK_CALLBACK_BEGIN_PRIMAL_SENSITIVITY`

Primal sensitivity analysis is started.

`MSK_CALLBACK_BEGIN_PRIMAL_SETUP_BI`

The callback function is called when the primal BI setup is started.

`MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX`

The callback function is called when the primal simplex optimizer is started.

`MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX_BI`

The callback function is called from within the basis identification procedure when the primal simplex clean-up phase is started.

`MSK_CALLBACK_BEGIN_QCQO_REFORMULATE`

Begin QCQO reformulation.

`MSK_CALLBACK_BEGIN_READ`

MOSEK has started reading a problem file.

`MSK_CALLBACK_BEGIN_ROOT_CUTGEN`

The callback function is called when root cut generation is started.

`MSK_CALLBACK_BEGIN_SIMPLEX`

The callback function is called when the simplex optimizer is started.

`MSK_CALLBACK_BEGIN_SOLVE_ROOT_RELAX`

The callback function is called when solution of root relaxation is started.

`MSK_CALLBACK_BEGIN_TO_CONIC`

Begin conic reformulation.

`MSK_CALLBACK_BEGIN_WRITE`

MOSEK has started writing a problem file.

`MSK_CALLBACK_CONIC`

The callback function is called from within the conic optimizer after the information database has been updated.

`MSK_CALLBACK_DECOMP_MIO`

The callback function is called when the dedicated algorithm for independent blocks inside the mixed-integer solver is started.

`MSK_CALLBACK_DUAL_SIMPLEX`

The callback function is called from within the dual simplex optimizer.

MSK_CALLBACK_END_BI

The callback function is called when the basis identification procedure is terminated.

MSK_CALLBACK_END_CONIC

The callback function is called when the conic optimizer is terminated.

MSK_CALLBACK_END_DUAL_BI

The callback function is called from within the basis identification procedure when the dual phase is terminated.

MSK_CALLBACK_END_DUAL_SENSITIVITY

Dual sensitivity analysis is terminated.

MSK_CALLBACK_END_DUAL_SETUP_BI

The callback function is called when the dual BI phase is terminated.

MSK_CALLBACK_END_DUAL_SIMPLEX

The callback function is called when the dual simplex optimizer is terminated.

MSK_CALLBACK_END_DUAL_SIMPLEX_BI

The callback function is called from within the basis identification procedure when the dual clean-up phase is terminated.

MSK_CALLBACK_END_FOLDING

The calback function is called at the end of folding.

MSK_CALLBACK_END_FOLDING_BI

TBD

MSK_CALLBACK_END_FOLDING_BI_DUAL

TBD

MSK_CALLBACK_END_FOLDING_BI_INITIALIZE

TBD

MSK_CALLBACK_END_FOLDING_BI_OPTIMIZER

TBD

MSK_CALLBACK_END_FOLDING_BI_PRIMAL

TBD

MSK_CALLBACK_END_INFEAS_ANA

The callback function is called when the infeasibility analyzer is terminated.

MSK_CALLBACK_END_INITIALIZE_BI

The callback function is called from within the basis identification procedure when the initialization phase is terminated.

MSK_CALLBACK_END_INTPNT

The callback function is called when the interior-point optimizer is terminated.

MSK_CALLBACK_END_LICENSE_WAIT

End waiting for license.

MSK_CALLBACK_END_MIO

The callback function is called when the mixed-integer optimizer is terminated.

MSK_CALLBACK_END_OPTIMIZE_BI

TBD.

MSK_CALLBACK_END_OPTIMIZER

The callback function is called when the optimizer is terminated.

MSK_CALLBACK_END_PRESOLVE

The callback function is called when the presolve is completed.

MSK_CALLBACK_END_PRIMAL_BI

The callback function is called from within the basis identification procedure when the primal phase is terminated.

MSK_CALLBACK_END_PRIMAL_REPAIR

End primal feasibility repair.

MSK_CALLBACK_END_PRIMAL_SENSITIVITY

Primal sensitivity analysis is terminated.

MSK_CALLBACK_END_PRIMAL_SETUP_BI

The callback function is called when the primal BI setup is terminated.

MSK_CALLBACK_END_PRIMAL_SIMPLEX

The callback function is called when the primal simplex optimizer is terminated.

MSK_CALLBACK_END_PRIMAL_SIMPLEX_BI

The callback function is called from within the basis identification procedure when the primal clean-up phase is terminated.

MSK_CALLBACK_END_QCQO_REFORMULATE

End QCQO reformulation.

MSK_CALLBACK_END_READ

MOSEK has finished reading a problem file.

MSK_CALLBACK_END_ROOT_CUTGEN

The callback function is called when root cut generation is terminated.

MSK_CALLBACK_END_SIMPLEX

The callback function is called when the simplex optimizer is terminated.

MSK_CALLBACK_END_SIMPLEX_BI

The callback function is called from within the basis identification procedure when the simplex clean-up phase is terminated.

MSK_CALLBACK_END_SOLVE_ROOT_RELAX

The callback function is called when solution of root relaxation is terminated.

MSK_CALLBACK_END_TO_CONIC

End conic reformulation.

MSK_CALLBACK_END_WRITE

MOSEK has finished writing a problem file.

MSK_CALLBACK_FOLDING_BI_DUAL

TBD

MSK_CALLBACK_FOLDING_BI_OPTIMIZER

TBD

MSK_CALLBACK_FOLDING_BI_PRIMAL

TBD

MSK_CALLBACK_HEARTBEAT

A heartbeat callback.

MSK_CALLBACK_IM_DUAL_SENSIVITY

The callback function is called at an intermediate stage of the dual sensitivity analysis.

MSK_CALLBACK_IM_DUAL_SIMPLEX

The callback function is called at an intermediate point in the dual simplex optimizer.

MSK_CALLBACK_IM_LICENSE_WAIT

MOSEK is waiting for a license.

MSK_CALLBACK_IM_LU

The callback function is called from within the LU factorization procedure at an intermediate point.

MSK_CALLBACK_IM_MIO

The callback function is called at an intermediate point in the mixed-integer optimizer.

MSK_CALLBACK_IM_MIO_DUAL_SIMPLEX

The callback function is called at an intermediate point in the mixed-integer optimizer while running the dual simplex optimizer.

MSK_CALLBACK_IM_MIO_INTPNT

The callback function is called at an intermediate point in the mixed-integer optimizer while running the interior-point optimizer.

MSK_CALLBACK_IM_MIO_PRIMAL_SIMPLEX

The callback function is called at an intermediate point in the mixed-integer optimizer while running the primal simplex optimizer.

MSK_CALLBACK_IM_ORDER

The callback function is called from within the matrix ordering procedure at an intermediate point.

MSK_CALLBACK_IM_PRIMAL_SENSIVITY

The callback function is called at an intermediate stage of the primal sensitivity analysis.

MSK_CALLBACK_IM_PRIMAL_SIMPLEX

The callback function is called at an intermediate point in the primal simplex optimizer.

MSK_CALLBACK_IM_READ

Intermediate stage in reading.

MSK_CALLBACK_IM_ROOT_CUTGEN

The callback is called from within root cut generation at an intermediate stage.

MSK_CALLBACK_IM_SIMPLEX

The callback function is called from within the simplex optimizer at an intermediate point.

MSK_CALLBACK_INTPNT

The callback function is called from within the interior-point optimizer after the information database has been updated.

MSK_CALLBACK_NEW_INT_MIO

The callback function is called after a new integer solution has been located by the mixed-integer optimizer.

MSK_CALLBACK_OPTIMIZE_BI

TBD.

MSK_CALLBACK_PRIMAL_SIMPLEX

The callback function is called from within the primal simplex optimizer.

MSK_CALLBACK_QO_REFORMULATE

The callback function is called at an intermediate stage of the conic quadratic reformulation.

MSK_CALLBACK_READ_OPF

The callback function is called from the OPF reader.

MSK_CALLBACK_READ_OPF_SECTION

A chunk of Q non-zeros has been read from a problem file.

MSK_CALLBACK_RESTART_MIO

The callback function is called when the mixed-integer optimizer is restarted.

MSK_CALLBACK_SOLVING_REMOTE

The callback function is called while the task is being solved on a remote server.

MSK_CALLBACK_UPDATE_DUAL_BI

The callback function is called from within the basis identification procedure at an intermediate point in the dual phase.

MSK_CALLBACK_UPDATE_DUAL_SIMPLEX

The callback function is called in the dual simplex optimizer.

MSK_CALLBACK_UPDATE_DUAL_SIMPLEX_BI

The callback function is called from within the basis identification procedure at an intermediate point in the dual simplex clean-up phase. The frequency of the callbacks is controlled by the *MSK_IPAR_LOG_SIM_FREQ* parameter.

MSK_CALLBACK_UPDATE_PRESOLVE

The callback function is called from within the presolve procedure.

MSK_CALLBACK_UPDATE_PRIMAL_BI

The callback function is called from within the basis identification procedure at an intermediate point in the primal phase.

MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX

The callback function is called in the primal simplex optimizer.

MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX_BI

The callback function is called from within the basis identification procedure at an intermediate point in the primal simplex clean-up phase. The frequency of the callbacks is controlled by the *MSK_IPAR_LOG_SIM_FREQ* parameter.

MSK_CALLBACK_UPDATE_SIMPLEX

The callback function is called from simplex optimizer.

MSK_CALLBACK_WRITE_OPF

The callback function is called from the OPF writer.

compresstype

Compression types

MSK_COMPRESS_NONE

No compression is used.

MSK_COMPRESS_FREE

The type of compression used is chosen automatically.

MSK_COMPRESS_GZIP

The type of compression used is gzip compatible.

MSK_COMPRESS_ZSTD

The type of compression used is zstd compatible.

conetype

Cone types

MSK_CT_QUAD

The cone is a quadratic cone.

MSK_CT_RQUAD

The cone is a rotated quadratic cone.

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.

domaintype

Cone types

MSK_DOMAIN_R

R.

MSK_DOMAIN_RZERO

The zero vector.

MSK_DOMAIN_RPLUS

The positive orthant.

MSK_DOMAIN_RMINUS

The negative orthant.

MSK_DOMAIN_QUADRATIC_CONE

The quadratic cone.

MSK_DOMAIN_RQUADRATIC_CONE

The rotated quadratic cone.

MSK_DOMAIN_PRIMAL_EXP_CONE

The primal exponential cone.

MSK_DOMAIN_DUAL_EXP_CONE

The dual exponential cone.

MSK_DOMAIN_PRIMAL_POWER_CONE

The primal power cone.

MSK_DOMAIN_DUAL_POWER_CONE

The dual power cone.

MSK_DOMAIN_PRIMAL_GEO_MEAN_CONE

The primal geometric mean cone.

MSK_DOMAIN_DUAL_GEO_MEAN_CONE

The dual geometric mean cone.

MSK_DOMAIN_SVEC_PSD_CONE

The vectorized positive semidefinite 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.

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_DATA_FORMAT_JSON_TASK
: JSON based task format.

`solformat`
: Data format types

MSK_SOL_FORMAT_EXTENSION
: The file extension is used to determine the data file format.

MSK_SOL_FORMAT_B
: Simple binary format

MSK_SOL_FORMAT_TASK
: Tar based format.

MSK_SOL_FORMAT_JSON_TASK
: JSON based format.

`dinfitem`
: Double information items

MSK_DINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_DENSITY
: Density percentage of the scalarized constraint matrix.

MSK_DINF_BI_CLEAN_TIME
: Time spent within the clean-up phase of the basis identification procedure since its invocation (in seconds).

MSK_DINF_BI_DUAL_TIME
: Time spent within the dual phase basis identification procedure since its invocation (in seconds).

MSK_DINF_BI_PRIMAL_TIME
: Time spent within the primal phase of the basis identification procedure since its invocation (in seconds).

MSK_DINF_BI_TIME
: Time spent within the basis identification procedure since its invocation (in seconds).

MSK_DINF_FOLDING_BI_OPTIMIZE_TIME
: TBD

MSK_DINF_FOLDING_BI_UNFOLD_DUAL_TIME
: TBD

MSK_DINF_FOLDING_BI_UNFOLD_INITIALIZE_TIME

TBD

MSK_DINF_FOLDING_BI_UNFOLD_PRIMAL_TIME

TBD

MSK_DINF_FOLDING_BI_UNFOLD_TIME

TBD

MSK_DINF_FOLDING_FACTOR

Problem size after folding as a fraction of the original size.

MSK_DINF_FOLDING_TIME

Total time spent in folding for continuous problems (in seconds).

MSK_DINF_INTPNT_DUAL_FEAS

Dual feasibility measure reported by the interior-point optimizer. (For the interior-point optimizer this measure is not directly related to the original problem because a homogeneous model is employed.)

MSK_DINF_INTPNT_DUAL_OBJ

Dual objective value reported by the interior-point optimizer.

MSK_DINF_INTPNT_FACTOR_NUM_FLOPS

An estimate of the number of flops used in the factorization.

MSK_DINF_INTPNT_OPT_STATUS

A measure of optimality of the solution. It should converge to $+1$ if the problem has a primal-dual optimal solution, and converge to -1 if the problem is (strictly) primal or dual infeasible. If the measure converges to another constant, or fails to settle, the problem is usually ill-posed.

MSK_DINF_INTPNT_ORDER_TIME

Order time (in seconds).

MSK_DINF_INTPNT_PRIMAL_FEAS

Primal feasibility measure reported by the interior-point optimizer. (For the interior-point optimizer this measure is not directly related to the original problem because a homogeneous model is employed).

MSK_DINF_INTPNT_PRIMAL_OBJ

Primal objective value reported by the interior-point optimizer.

MSK_DINF_INTPNT_TIME

Time spent within the interior-point optimizer since its invocation (in seconds).

MSK_DINF_MIO_CLIQUE_SELECTION_TIME

Selection time for clique cuts (in seconds).

MSK_DINF_MIO_CLIQUE_SEPARATION_TIME

Separation time for clique cuts (in seconds).

MSK_DINF_MIO_CMIR_SELECTION_TIME

Selection time for CMIR cuts (in seconds).

MSK_DINF_MIO_CMIR_SEPARATION_TIME

Separation time for CMIR cuts (in seconds).

MSK_DINF_MIO_CONSTRUCT_SOLUTION_OBJ

If **MOSEK** has successfully constructed an integer feasible solution, then this item contains the optimal objective value corresponding to the feasible solution.

MSK_DINF_MIO_DUAL_BOUND_AFTER_PRESOLVE

Value of the dual bound after presolve but before cut generation.

MSK_DINF_MIO_GMI_SELECTION_TIME

Selection time for GMI cuts (in seconds).

MSK_DINF_MIO_GMI_SEPARATION_TIME

Separation time for GMI cuts (in seconds).

MSK_DINF_MIO_IMPLIED_BOUND_SELECTION_TIME

Selection time for implied bound cuts (in seconds).

MSK_DINF_MIO_IMPLIED_BOUND_SEPARATION_TIME

Separation time for implied bound cuts (in seconds).

MSK_DINF_MIO_INITIAL_FEASIBLE_SOLUTION_OBJ

If the user provided solution was found to be feasible this information item contains it's objective value.

MSK_DINF_MIO_KNAPSACK_COVER_SELECTION_TIME

Selection time for knapsack cover (in seconds).

MSK_DINF_MIO_KNAPSACK_COVER_SEPARATION_TIME

Separation time for knapsack cover (in seconds).

MSK_DINF_MIO_LIPRO_SELECTION_TIME

Selection time for lift-and-project cuts (in seconds).

MSK_DINF_MIO_LIPRO_SEPARATION_TIME

Separation time for lift-and-project cuts (in seconds).

MSK_DINF_MIO_OBJ_ABS_GAP

Given the mixed-integer optimizer has computed a feasible solution and a bound on the optimal objective value, then this item contains the absolute gap defined by

$$|(\text{objective value of feasible solution}) - (\text{objective bound})|.$$

Otherwise it has the value -1.0.

MSK_DINF_MIO_OBJ_BOUND

The best known bound on the objective function. This value is undefined until at least one relaxation has been solved: To see if this is the case check that *MSK_IINF_MIO_NUM_RELAX* is strictly positive.

MSK_DINF_MIO_OBJ_INT

The primal objective value corresponding to the best integer feasible solution. Please note that at least one integer feasible solution must have been located i.e. check *MSK_IINF_MIO_NUM_INT_SOLUTIONS*.

MSK_DINF_MIO_OBJ_REL_GAP

Given that the mixed-integer optimizer has computed a feasible solution and a bound on the optimal objective value, then this item contains the relative gap defined by

$$\frac{|(\text{objective value of feasible solution}) - (\text{objective bound})|}{\max(\delta, |(\text{objective value of feasible solution})|)}.$$

where δ is given by the parameter *MSK_DPAR_MIO_REL_GAP_CONST*. Otherwise it has the value -1.0.

MSK_DINF_MIO_PROBING_TIME

Total time for probing (in seconds).

MSK_DINF_MIO_ROOT_CUT_SELECTION_TIME

Total time for cut selection (in seconds).

MSK_DINF_MIO_ROOT_CUT_SEPARATION_TIME

Total time for cut separation (in seconds).

MSK_DINF_MIO_ROOT_OPTIMIZER_TIME

Time spent in the contiuous optimizer while processing the root node relaxation (in seconds).

MSK_DINF_MIO_ROOT_PRESOLVE_TIME

Time spent presolving the problem at the root node (in seconds).

MSK_DINF_MIO_ROOT_TIME

Time spent processing the root node (in seconds).

MSK_DINF_MIO_SYMMETRY_DETECTION_TIME

Total time for symmetry detection (in seconds).

MSK_DINF_MIO_SYMMETRY_FACTOR

Degree to which the problem is affected by detected symmetry.

MSK_DINF_MIO_TIME

Time spent in the mixed-integer optimizer (in seconds).

MSK_DINF_MIO_USER_OBJ_CUT

If the objective cut is used, then this information item has the value of the cut.

MSK_DINF_OPTIMIZER_TICKS

Total number of ticks spent in the optimizer since it was invoked. It is strictly negative if it is not available.

MSK_DINF_OPTIMIZER_TIME

Total time spent in the optimizer since it was invoked (in seconds).

MSK_DINF_PRESOLVE_ELI_TIME

Total time spent in the eliminator since the presolve was invoked (in seconds).

MSK_DINF_PRESOLVE_LINDEP_TIME

Total time spent in the linear dependency checker since the presolve was invoked (in seconds).

MSK_DINF_PRESOLVE_TIME

Total time spent in the presolve since it was invoked (in seconds).

MSK_DINF_PRESOLVE_TOTAL_PRIMAL_PERTURBATION

Total perturbation of the bounds of the primal problem.

MSK_DINF_PRIMAL_REPAIR_PENALTY_OBJ

The optimal objective value of the penalty function.

MSK_DINF_QCQO_REFORMULATE_MAX_PERTURBATION

Maximum absolute diagonal perturbation occurring during the QCQO reformulation.

MSK_DINF_QCQO_REFORMULATE_TIME

Time spent with conic quadratic reformulation (in seconds).

MSK_DINF_QCQO_REFORMULATE_WORST_CHOLESKY_COLUMN_SCALING

Worst Cholesky column scaling.

MSK_DINF_QCQO_REFORMULATE_WORST_CHOLESKY_DIAG_SCALING

Worst Cholesky diagonal scaling.

MSK_DINF_READ_DATA_TIME

Time spent reading the data file (in seconds).

MSK_DINF_REMOTE_TIME

The total real time in seconds spent when optimizing on a server by the process performing the optimization on the server (in seconds).

MSK_DINF_SIM_DUAL_TIME

Time spent in the dual simplex optimizer since invoking it (in seconds).

MSK_DINF_SIM_FEAS

Feasibility measure reported by the simplex optimizer.

MSK_DINF_SIM_OBJ

Objective value reported by the simplex optimizer.

MSK_DINF_SIM_PRIMAL_TIME

Time spent in the primal simplex optimizer since invoking it (in seconds).

MSK_DINF_SIM_TIME

Time spent in the simplex optimizer since invoking it (in seconds).

MSK_DINF_SOL_BAS_DUAL_OBJ

Dual objective value of the basic solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_BAS_DVIOLCON

Maximal dual bound violation for x^c in the basic solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_BAS_DVIOLVAR

Maximal dual bound violation for x^x in the basic solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_BAS_NRM_BARX

Infinity norm of $\overline{X}$ in the basic solution.

MSK_DINF_SOL_BAS_NRM_SLC

Infinity norm of s_l^c in the basic solution.

MSK_DINF_SOL_BAS_NRM_SLX

Infinity norm of s_l^x in the basic solution.

MSK_DINF_SOL_BAS_NRM_SUC

Infinity norm of s_u^c in the basic solution.

MSK_DINF_SOL_BAS_NRM_SUX

Infinity norm of s_u^X in the basic solution.

MSK_DINF_SOL_BAS_NRM_XC

Infinity norm of x^c in the basic solution.

MSK_DINF_SOL_BAS_NRM_XX

Infinity norm of x^x in the basic solution.

MSK_DINF_SOL_BAS_NRM_Y

Infinity norm of y in the basic solution.

MSK_DINF_SOL_BAS_PRIMAL_OBJ

Primal objective value of the basic solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_BAS_PVIOLCON

Maximal primal bound violation for x^c in the basic solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_BAS_PVIOLVAR

Maximal primal bound violation for x^x in the basic solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITG_NRM_BARX

Infinity norm of $\overline{X}$ in the integer solution.

MSK_DINF_SOL_ITG_NRM_XC

Infinity norm of x^c in the integer solution.

MSK_DINF_SOL_ITG_NRM_XX

Infinity norm of x^x in the integer solution.

MSK_DINF_SOL_ITG_PRIMAL_OBJ

Primal objective value of the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITG_PVIOLACC

Maximal primal violation for affine conic constraints in the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITG_PVIOLBARVAR

Maximal primal bound violation for $\overline{X}$ in the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITG_PVIOLCON

Maximal primal bound violation for x^c in the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITG_PVIOLCONES

Maximal primal violation for primal conic constraints in the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITG_PVIOLDJC

Maximal primal violation for disjunctive constraints in the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITG_PVIOLITG

Maximal violation for the integer constraints in the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITG_PVIOLVAR

Maximal primal bound violation for x^x in the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_DUAL_OBJ

Dual objective value of the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_DVIOLACC

Maximal dual violation for the affine conic constraints in the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_DVIOLBARVAR

Maximal dual bound violation for $\overline{X}$ in the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_DVIOLCON

Maximal dual bound violation for x^c in the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_DVIOLCONES

Maximal dual violation for conic constraints in the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_DVIOLVAR

Maximal dual bound violation for x^x in the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_NRM_BARS

Infinity norm of $\overline{S}$ in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_BARX

Infinity norm of $\overline{X}$ in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SLC

Infinity norm of s_l^c in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SLX

Infinity norm of s_l^x in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SNX

Infinity norm of s_n^x in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SUC

Infinity norm of s_u^c in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SUX

Infinity norm of s_u^X in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_XC

Infinity norm of x^c in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_XX

Infinity norm of x^x in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_Y

Infinity norm of y in the interior-point solution.

MSK_DINF_SOL_ITR_PRIMAL_OBJ

Primal objective value of the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_PVIOLACC

Maximal primal violation for affine conic constraints in the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_PVIOLBARVAR

Maximal primal bound violation for $\overline{X}$ in the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_PVIOLCON

Maximal primal bound violation for x^c in the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_PVIOLCONES

Maximal primal violation for conic constraints in the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_SOL_ITR_PVIOLVAR

Maximal primal bound violation for x^x in the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *updatesolutioninfo*.

MSK_DINF_TO_CONIC_TIME

Time spent in the last to conic reformulation (in seconds).

MSK_DINF_WRITE_DATA_TIME

Time spent writing the data file (in seconds).

feature

License feature

MSK_FEATURE_PTS

Base system.

MSK_FEATURE_PTON

Conic extension.

liinfitem

Long integer information items.

MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_COLUMNS

Number of columns in the scalarized constraint matrix.

MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_NZ

Number of non-zero entries in the scalarized constraint matrix.

MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_ROWS

Number of rows in the scalarized constraint matrix.

MSK_LIINF_BI_CLEAN_ITER

Number of clean iterations performed in the basis identification.

MSK_LIINF_BI_DUAL_ITER

Number of dual pivots performed in the basis identification.

MSK_LIINF_BI_PRIMAL_ITER

Number of primal pivots performed in the basis identification.

MSK_LIINF_FOLDING_BI_DUAL_ITER

TBD

MSK_LIINF_FOLDING_BI_OPTIMIZER_ITER

TBD

MSK_LIINF_FOLDING_BI_PRIMAL_ITER

TBD

MSK_LIINF_INTPNT_FACTOR_NUM_NZ

Number of non-zeros in factorization.

MSK_LIINF_MIO_ANZ

Number of non-zero entries in the constraint matrix of the problem to be solved by the mixed-integer optimizer.

MSK_LIINF_MIO_FINAL_ANZ

Number of non-zero entries in the constraint matrix of the mixed-integer optimizer's final problem.

MSK_LIINF_MIO_INTPNT_ITER

Number of interior-point iterations performed by the mixed-integer optimizer.

MSK_LIINF_MIO_NUM_DUAL_ILLPOSED_CER

Number of dual illposed certificates encountered by the mixed-integer optimizer.

MSK_LIINF_MIO_NUM_PRIM_ILLPOSED_CER

Number of primal illposed certificates encountered by the mixed-integer optimizer.

MSK_LIINF_MIO_PRESOLVED_ANZ

Number of non-zero entries in the constraint matrix of the problem after the mixed-integer optimizer's presolve.

MSK_LIINF_MIO_SIMPLEX_ITER

Number of simplex iterations performed by the mixed-integer optimizer.

MSK_LIINF_RD_NUMACC

Number of affince conic constraints.

MSK_LIINF_RD_NUMANZ

Number of non-zeros in A that is read.

MSK_LIINF_RD_NUMDJC

Number of disjuncive constraints.

MSK_LIINF_RD_NUMQNZ

Number of Q non-zeros.

MSK_LIINF_SIMPLEX_ITER

Number of iterations performed by the simplex optimizer.

iinfitem

Integer information items.

MSK_IINF_ANA_PRO_NUM_CON

Number of constraints in the problem. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_CON_EQ

Number of equality constraints. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_CON_FR

Number of unbounded constraints. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_CON_LO

Number of constraints with a lower bound and an infinite upper bound. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_CON_RA

Number of constraints with finite lower and upper bounds. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_CON_UP

Number of constraints with an upper bound and an infinite lower bound. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR

Number of variables in the problem. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_BIN

Number of binary (0-1) variables. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_CONT

Number of continuous variables. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_EQ

Number of fixed variables. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_FR

Number of free variables. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_INT

Number of general integer variables. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_LO

Number of variables with a lower bound and an infinite upper bound. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_RA

Number of variables with finite lower and upper bounds. This value is set by *analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_UP

Number of variables with an upper bound and an infinite lower bound. This value is set by *analyzeproblem*.

MSK_IINF_FOLDING_APPLIED

Non-zero if folding was exploited.

MSK_IINF_INTPNT_FACTOR_DIM_DENSE

Dimension of the dense sub system in factorization.

MSK_IINF_INTPNT_ITER

Number of interior-point iterations since invoking the interior-point optimizer.

MSK_IINF_INTPNT_NUM_THREADS

Number of threads that the interior-point optimizer is using.

MSK_IINF_INTPNT_SOLVE_DUAL

Non-zero if the interior-point optimizer is solving the dual problem.

MSK_IINF_MIO_ABSGAP_SATISFIED

Non-zero if absolute gap is within tolerances.

`MSK_IINF_MIO_CLIQUE_TABLE_SIZE`

Size of the clique table.

`MSK_IINF_MIO_CONSTRUCT_SOLUTION`

This item informs if **MOSEK** constructed an initial integer feasible solution.

- -1: tried, but failed,
- 0: no partial solution supplied by the user,
- 1: constructed feasible solution.

`MSK_IINF_MIO_FINAL_NUMBIN`

Number of binary variables in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMBINCONEVAR`

Number of binary cone variables in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMCON`

Number of constraints in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMCONE`

Number of cones in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMCONEVAR`

Number of cone variables in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMCONT`

Number of continuous variables in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMCONTCONEVAR`

Number of continuous cone variables in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMDEXPCONES`

Number of dual exponential cones in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMDJC`

Number of disjunctive constraints in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMDPOWCONES`

Number of dual power cones in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMINT`

Number of integer variables in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMINTCONEVAR`

Number of integer cone variables in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMPEXPCONES`

Number of primal exponential cones in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMPPOWCONES`

Number of primal power cones in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMQCONES`

Number of quadratic cones in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMRQCONES`

Number of rotated quadratic cones in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_FINAL_NUMVAR`

Number of variables in the mixed-integer optimizer's final problem.

`MSK_IINF_MIO_INITIAL_FEASIBLE_SOLUTION`

This item informs if **MOSEK** found the solution provided by the user to be feasible

- 0: solution provided by the user was not found to be feasible for the current problem,
- 1: user provided solution was found to be feasible.

MSK_IINF_MIO_NODE_DEPTH

Depth of the last node solved.

MSK_IINF_MIO_NUM_ACTIVE_NODES

Number of active branch and bound nodes.

MSK_IINF_MIO_NUM_ACTIVE_ROOT_CUTS

Number of active cuts in the final relaxation after the mixed-integer optimizer's root cut generation.

MSK_IINF_MIO_NUM_BLOCKS_SOLVED_IN_BB

Number of independent decomposition blocks solved though a dedicated algorithm.

MSK_IINF_MIO_NUM_BLOCKS_SOLVED_IN_PRESOLVE

Number of independent decomposition blocks solved during presolve.

MSK_IINF_MIO_NUM_BRANCH

Number of branches performed during the optimization.

MSK_IINF_MIO_NUM_INT_SOLUTIONS

Number of integer feasible solutions that have been found.

MSK_IINF_MIO_NUM_RELAX

Number of relaxations solved during the optimization.

MSK_IINF_MIO_NUM_REPEATED_PRESOLVE

Number of times presolve was repeated at root.

MSK_IINF_MIO_NUM_RESTARTS

Number of restarts performed during the optimization.

MSK_IINF_MIO_NUM_ROOT_CUT_ROUNDS

Number of cut separation rounds at the root node of the mixed-integer optimizer.

MSK_IINF_MIO_NUM_SELECTED_CLIQUE_CUTS

Number of clique cuts selected to be included in the relaxation.

MSK_IINF_MIO_NUM_SELECTED_CMIR_CUTS

Number of Complemented Mixed Integer Rounding (CMIR) cuts selected to be included in the relaxation.

MSK_IINF_MIO_NUM_SELECTED_GOMORY_CUTS

Number of Gomory cuts selected to be included in the relaxation.

MSK_IINF_MIO_NUM_SELECTED_IMPLIED_BOUND_CUTS

Number of implied bound cuts selected to be included in the relaxation.

MSK_IINF_MIO_NUM_SELECTED_KNAPSACK_COVER_CUTS

Number of clique cuts selected to be included in the relaxation.

MSK_IINF_MIO_NUM_SELECTED_LIPRO_CUTS

Number of lift-and-project cuts selected to be included in the relaxation.

MSK_IINF_MIO_NUM_SEPARATED_CLIQUE_CUTS

Number of separated clique cuts.

MSK_IINF_MIO_NUM_SEPARATED_CMIR_CUTS

Number of separated Complemented Mixed Integer Rounding (CMIR) cuts.

MSK_IINF_MIO_NUM_SEPARATED_GOMORY_CUTS

Number of separated Gomory cuts.

MSK_IINF_MIO_NUM_SEPARATED_IMPLIED_BOUND_CUTS

Number of separated implied bound cuts.

MSK_IINF_MIO_NUM_SEPARATED_KNAPSACK_COVER_CUTS

Number of separated clique cuts.

MSK_IINF_MIO_NUM_SEPARATED_LIPRO_CUTS

Number of separated lift-and-project cuts.

MSK_IINF_MIO_NUM_SOLVED_NODES

Number of branch and bounds nodes solved in the main branch and bound tree.

MSK_IINF_MIO_NUMBIN

Number of binary variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMBINCONEVAR

Number of binary cone variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMCON

Number of constraints in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMCONE

Number of cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMCONEVAR

Number of cone variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMCONT

Number of continuous variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMCONTCONEVAR

Number of continuous cone variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMDEXPCONES

Number of dual exponential cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMDJC

Number of disjunctive constraints in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMDPOWCONES

Number of dual power cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMINT

Number of integer variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMINTCONEVAR

Number of integer cone variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMPEXPCONES

Number of primal exponential cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMPPOWCONES

Number of primal power cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMQCONES

Number of quadratic cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMRQCONES

Number of rotated quadratic cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMVAR

Number of variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_OBJ_BOUND_DEFINED

Non-zero if a valid objective bound has been found, otherwise zero.

MSK_IINF_MIO_PRESOLVED_NUMBIN

Number of binary variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMBINCONEVAR

Number of binary cone variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMCON

Number of constraints in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMCONE

Number of cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMCONEVAR

Number of cone variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMCONT

Number of continuous variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMCONTCONEVAR

Number of continuous cone variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMDEXPCONES

Number of dual exponential cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMDJC

Number of disjunctive constraints in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMDPOWCONES

Number of dual power cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMINT

Number of integer variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMINTCONEVAR

Number of integer cone variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMPEXPCONES

Number of primal exponential cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMPPOWCONES

Number of primal power cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMQCONES

Number of quadratic cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMRQCONES

Number of rotated quadratic cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMVAR

Number of variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_RELGAP_SATISFIED

Non-zero if relative gap is within tolerances.

MSK_IINF_MIO_TOTAL_NUM_SELECTED_CUTS

Total number of cuts selected to be included in the relaxation by the mixed-integer optimizer.

MSK_IINF_MIO_TOTAL_NUM_SEPARATED_CUTS

Total number of cuts separated by the mixed-integer optimizer.

MSK_IINF_MIO_USER_OBJ_CUT

If it is non-zero, then the objective cut is used.

MSK_IINF_OPT_NUMCON

Number of constraints in the problem solved when the optimizer is called.

MSK_IINF_OPT_NUMVAR

Number of variables in the problem solved when the optimizer is called

MSK_IINF_OPTIMIZE_RESPONSE

The response code returned by optimize.

MSK_IINF_PRESOLVE_NUM_PRIMAL_PERTURBATIONS

Number perturbations to thhe bounds of the primal problem.

MSK_IINF_PURIFY_DUAL_SUCCESS

Is nonzero if the dual solution is purified.

MSK_IINF_PURIFY_PRIMAL_SUCCESS

Is nonzero if the primal solution is purified.

MSK_IINF_RD_NUMBARVAR

Number of symmetric variables read.

MSK_IINF_RD_NUMCON

Number of constraints read.

MSK_IINF_RD_NUMCONE

Number of conic constraints read.

MSK_IINF_RD_NUMINTVAR

Number of integer-constrained variables read.

MSK_IINF_RD_NUMQ

Number of nonempty Q matrices read.

MSK_IINF_RD_NUMVAR

Number of variables read.

MSK_IINF_RD_PROTYPE

Problem type.

MSK_IINF_SIM_DUAL_DEG_ITER

The number of dual degenerate iterations.

MSK_IINF_SIM_DUAL_HOTSTART

If 1 then the dual simplex algorithm is solving from an advanced basis.

MSK_IINF_SIM_DUAL_HOTSTART_LU

If 1 then a valid basis factorization of full rank was located and used by the dual simplex algorithm.

MSK_IINF_SIM_DUAL_INF_ITER

The number of iterations taken with dual infeasibility.

MSK_IINF_SIM_DUAL_ITER

Number of dual simplex iterations during the last optimization.

MSK_IINF_SIM_NUMCON

Number of constraints in the problem solved by the simplex optimizer.

MSK_IINF_SIM_NUMVAR

Number of variables in the problem solved by the simplex optimizer.

MSK_IINF_SIM_PRIMAL_DEG_ITER

The number of primal degenerate iterations.

MSK_IINF_SIM_PRIMAL_HOTSTART

If 1 then the primal simplex algorithm is solving from an advanced basis.

MSK_IINF_SIM_PRIMAL_HOTSTART_LU

If 1 then a valid basis factorization of full rank was located and used by the primal simplex algorithm.

MSK_IINF_SIM_PRIMAL_INF_ITER

The number of iterations taken with primal infeasibility.

MSK_IINF_SIM_PRIMAL_ITER

Number of primal simplex iterations during the last optimization.

MSK_IINF_SIM_SOLVE_DUAL

Is non-zero if dual problem is solved.

MSK_IINF_SOL_BAS_PROSTA

Problem status of the basic solution. Updated after each optimization.

MSK_IINF_SOL_BAS_SOLSTA

Solution status of the basic solution. Updated after each optimization.

MSK_IINF_SOL_ITG_PROSTA

Problem status of the integer solution. Updated after each optimization.

MSK_IINF_SOL_ITG_SOLSTA

Solution status of the integer solution. Updated after each optimization.

MSK_IINF_SOL_ITR_PROSTA

Problem status of the interior-point solution. Updated after each optimization.

MSK_IINF_SOL_ITR_SOLSTA

Solution status of the interior-point solution. Updated after each optimization.

MSK_IINF_STO_NUM_A_REALLOC

Number of times the storage for storing A has been changed. A large value may indicates that memory fragmentation may occur.

inftype

Information item types

MSK_INF_DOU_TYPE

Is a double information type.

MSK_INF_INT_TYPE

Is an integer.

MSK_INF_LINT_TYPE

Is a long integer.

iomode

Input/output modes

MSK_IOMODE_READ

The file is read-only.

MSK_IOMODE_WRITE

The file is write-only. If the file exists then it is truncated when it is opened. Otherwise it is created when it is opened.

MSK_IOMODE_READWRITE

The file is to read and write.

branchdir

Specifies the branching direction.

MSK_BRANCH_DIR_FREE

The mixed-integer optimizer decides which branch to choose.

MSK_BRANCH_DIR_UP

The mixed-integer optimizer always chooses the up branch first.

MSK_BRANCH_DIR_DOWN

The mixed-integer optimizer always chooses the down branch first.

MSK_BRANCH_DIR_NEAR

Branch in direction nearest to selected fractional variable.

MSK_BRANCH_DIR_FAR

Branch in direction farthest from selected fractional variable.

MSK_BRANCH_DIR_ROOT_LP

Chose direction based on root lp value of selected variable.

MSK_BRANCH_DIR_GUIDED

Branch in direction of current incumbent.

MSK_BRANCH_DIR_PSEUDOCOST

Branch based on the pseudocost of the variable.

miqcqoreformmethod

Specifies the reformulation method for mixed-integer quadratic problems.

MSK_MIO_QCQO_REFORMULATION_METHOD_FREE

The mixed-integer optimizer decides which reformulation method to apply.

MSK_MIO_QCQO_REFORMULATION_METHOD_NONE

No reformulation method is applied.

MSK_MIO_QCQO_REFORMULATION_METHOD_LINEARIZATION

A reformulation via linearization is applied.

MSK_MIO_QCQO_REFORMULATION_METHOD_EIGEN_VAL_METHOD

The eigenvalue method is applied.

MSK_MIO_QCQO_REFORMULATION_METHOD_DIAG_SDP

A perturbation of matrix diagonals via the solution of SDPs is applied.

MSK_MIO_QCQO_REFORMULATION_METHOD_RELAX_SDP

A Reformulation based on the solution of an SDP-relaxation of the problem is applied.

miodatapermmethod

Specifies the problem data permutation method for mixed-integer problems.

MSK_MIO_DATA_PERMUTATION_METHOD_NONE

No problem data permutation is applied.

MSK_MIO_DATA_PERMUTATION_METHOD_CYCLIC_SHIFT

A random cyclic shift is applied to permute the problem data.

MSK_MIO_DATA_PERMUTATION_METHOD_RANDOM

A random permutation is applied to the problem data.

miocontsoltype

Continuous mixed-integer solution type

MSK_MIO_CONT_SOL_NONE

No interior-point or basic solution are reported when the mixed-integer optimizer is used.

MSK_MIO_CONT_SOL_ROOT

The reported interior-point and basic solutions are a solution to the root node problem when mixed-integer optimizer is used.

MSK_MIO_CONT_SOL_ITG

The reported interior-point and basic solutions are a solution to the problem with all integer variables fixed at the value they have in the integer solution. A solution is only reported in case the problem has a primal feasible solution.

MSK_MIO_CONT_SOL_ITG_REL

In case the problem is primal feasible then the reported interior-point and basic solutions are a solution to the problem with all integer variables fixed at the value they have in the integer solution. If the problem is primal infeasible, then the solution to the root node problem is reported.

miomode

Integer restrictions

MSK_MIO_MODE_IGNORED

The integer constraints are ignored and the problem is solved as a continuous problem.

MSK_MIO_MODE_SATISFIED

Integer restrictions should be satisfied.

`mionodeseltype`

Mixed-integer node selection types

MSK_MIO_NODE_SELECTION_FREE

The optimizer decides the node selection strategy.

MSK_MIO_NODE_SELECTION_FIRST

The optimizer employs a depth first node selection strategy.

MSK_MIO_NODE_SELECTION_BEST

The optimizer employs a best bound node selection strategy.

MSK_MIO_NODE_SELECTION_PSEUDO

The optimizer employs selects the node based on a pseudo cost estimate.

`miovarseltype`

Mixed-integer variable selection types

MSK_MIO_VAR_SELECTION_FREE

The optimizer decides the variable selection strategy.

MSK_MIO_VAR_SELECTION_PSEUDOCOST

The optimizer employs pseudocost variable selection.

MSK_MIO_VAR_SELECTION_STRONG

The optimizer employs strong branching varfiable selection

`mpsformat`

MPS file format type

MSK_MPS_FORMAT_STRICT

It is assumed that the input file satisfies the MPS format strictly.

MSK_MPS_FORMAT_RELAXED

It is assumed that the input file satisfies a slightly relaxed version of the MPS format.

MSK_MPS_FORMAT_FREE

It is assumed that the input file satisfies the free MPS format. This implies that spaces are not allowed in names. Otherwise the format is free.

MSK_MPS_FORMAT_CPLEX

The CPLEX compatible version of the MPS format is employed.

`objsense`

Objective sense types

MSK_OBJECTIVE_SENSE_MINIMIZE

The problem should be minimized.

MSK_OBJECTIVE_SENSE_MAXIMIZE

The problem should be maximized.

`onoffkey`

On/off

MSK_ON

Switch the option on.

MSK_OFF

Switch the option off.

`optimizertype`

Optimizer types

MSK_OPTIMIZER_CONIC

The optimizer for problems having conic constraints.

MSK_OPTIMIZER_DUAL_SIMPLEX

The dual simplex optimizer is used.

MSK_OPTIMIZER_FREE

The optimizer is chosen automatically.

MSK_OPTIMIZER_FREE_SIMPLEX

One of the simplex optimizers is used.

MSK_OPTIMIZER_INTPNT

The interior-point optimizer is used.

MSK_OPTIMIZER_MIXED_INT

The mixed-integer optimizer.

MSK_OPTIMIZER_NEW_DUAL_SIMPLEX

The new dual simplex optimizer is used.

MSK_OPTIMIZER_NEW_PRIMAL_SIMPLEX

The new primal simplex optimizer is used. It is not recommended to use this option.

MSK_OPTIMIZER_PRIMAL_SIMPLEX

The primal simplex optimizer is used.

orderingtype

Ordering strategies

MSK_ORDER_METHOD_FREE

The ordering method is chosen automatically.

MSK_ORDER_METHOD_APPMINLOC

Approximate minimum local fill-in ordering is employed.

MSK_ORDER_METHOD_EXPERIMENTAL

This option should not be used.

MSK_ORDER_METHOD_TRY_GRAPHPAR

Always try the graph partitioning based ordering.

MSK_ORDER_METHOD_FORCE_GRAPHPAR

Always use the graph partitioning based ordering even if it is worse than the approximate minimum local fill ordering.

MSK_ORDER_METHOD_NONE

No ordering is used. Note using this value almost always leads to a significantly slow down.

presolvemode

Presolve method.

MSK_PRESOLVE_MODE_OFF

The problem is not presolved before it is optimized.

MSK_PRESOLVE_MODE_ON

The problem is presolved before it is optimized.

MSK_PRESOLVE_MODE_FREE

It is decided automatically whether to presolve before the problem is optimized.

foldingmode

Method of folding (symmetry detection for continuous problems).

MSK_FOLDING_MODE_OFF

Disabled.

MSK_FOLDING_MODE_FREE

The solver decides on the usage and amount of folding.

MSK_FOLDING_MODE_FREE_UNLESS_BASIC

If only the interior-point solution is requested then the solver decides; if the basic solution is requested then folding is disabled.

MSK_FOLDING_MODE_FORCE

Full folding is always performed regardless of workload.

parametertype

Parameter type

MSK_PAR_INVALID_TYPE

Not a valid parameter.

MSK_PAR_DOU_TYPE

Is a double parameter.

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.

problemtype

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.

rescodetype

Response code type

MSK_RESPONSE_OK

The response code is OK.

MSK_RESPONSE_WRN

The response code is a warning.

MSK_RESPONSE_TRM

The response code is an optimizer termination status.

MSK_RESPONSE_ERR

The response code is an error.

MSK_RESPONSE_UNK

The response code does not belong to any class.

scalingtype

Scaling type

MSK_SCALING_FREE

The optimizer chooses the scaling heuristic.

MSK_SCALING_NONE

No scaling is performed.

scalingmethod

Scaling method

MSK_SCALING_METHOD_POW2

Scales only with power of 2 leaving the mantissa untouched.

MSK_SCALING_METHOD_FREE

The optimizer chooses the scaling heuristic.

sensitivitytype

Sensitivity types

MSK_SENSITIVITY_TYPE_BASIS

Basis sensitivity analysis is performed.

simseltype

Simplex selection strategy

MSK_SIM_SELECTION_FREE

The optimizer chooses the pricing strategy.

MSK_SIM_SELECTION_FULL

The optimizer uses full pricing.

MSK_SIM_SELECTION_ASE

The optimizer uses approximate steepest-edge pricing.

MSK_SIM_SELECTION_DEVEX

The optimizer uses devex steepest-edge pricing (or if it is not available an approximate steep-edge selection).

MSK_SIM_SELECTION_SE

The optimizer uses steepest-edge selection (or if it is not available an approximate steep-edge selection).

MSK_SIM_SELECTION_PARTIAL

The optimizer uses a partial selection approach. The approach is usually beneficial if the number of variables is much larger than the number of constraints.

solitem

Solution items

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.

soltype

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.

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

`MSK_VAR_TYPE_CONT`

Is a continuous variable.

`MSK_VAR_TYPE_INT`

Is an integer variable.

15.8 Supported domains

This section lists the domains supported by **MOSEK**. See Sec. 6 for how to apply domains to specify affine conic constraints (ACCs) and disjunctive constraints (DJCs).

15.8.1 Linear domains

Each linear domain is determined by the dimension n.

- *appendrzerodomain* : the **zero domain**, consisting of the origin $0^n \in \mathbb{R}^n$.
- *appendrplusdomain* : the **nonnegative orthant domain** $\mathbb{R}^n_{\geq 0}$.
- *appendrminusdomain* : the **nonpositive orthant domain** $\mathbb{R}^n_{\leq 0}$.
- *appendrdomain* : the **free domain**, consisting of the whole $\mathbb{R}^n$.

Membership in a linear domain is equivalent to imposing the corresponding set of n linear constraints, for instance $Fx + g \in 0^n$ is equivalent to $Fx + g = 0$ and so on. The free domain imposes no restriction.

15.8.2 Quadratic cone domains

The quadratic domains are determined by the dimension n.

- *appendquadraticconedomain* : the **quadratic cone domain** is the subset of $\mathbb{R}^n$ defined as

$$\mathcal{Q}^n = \left\{ x \in \mathbb{R}^n \ : \ x_1 \geq \sqrt{x_2^2 + \cdots + x_n^2} \right\}.$$

- *appendrquadraticconedomain* : the **rotated quadratic cone domain** is the subset of $\mathbb{R}^n$ defined as

$$\mathcal{Q}_r^n = \left\{ x \in \mathbb{R}^n \ : \ 2x_1 x_2 \geq x_3^2 + \cdots + x_n^2, \ x_1, x_2 \geq 0 \right\}.$$

15.8.3 Exponential cone domains

- *appendprimalexpconedomain* : the **primal exponential cone domain** is the subset of $\mathbb{R}^3$ defined as

$$K_{\exp} = \left\{(x_1, x_2, x_3) \in \mathbb{R}^3 \ : \ x_1 \geq x_2 \exp(x_3/x_2),\ x_1, x_2 \geq 0\right\}.$$

- *appenddualexpconedomain* : the **dual exponential cone domain** is the subset of $\mathbb{R}^3$ defined as

$$K^*_{\exp} = \left\{(x_1, x_2, x_3) \in \mathbb{R}^3 \ : \ x_1 \geq -x_3 \exp(x_2/x_3 - 1),\ x_1 \geq 0, x_3 \leq 0\right\}.$$

15.8.4 Power cone domains

A power cone domain is determined by the dimension n and a sequence of $1 \leq n_l < n$ positive real numbers (weights) $\alpha_1, \ldots, \alpha_{n_l}$.

- *appendprimalpowerconedomain* : the **primal power cone domain** is the subset of $\mathbb{R}^n$ defined as

$$\mathcal{P}_n^{(\alpha_1,\ldots,\alpha_{n_l})} = \left\{ x \in \mathbb{R}^n \ : \ \prod_{i=1}^{n_l} x_i^{\beta_i} \geq \sqrt{x_{n_l+1}^2 + \cdots + x_n^2},\ x_1, \ldots, x_{n_l} \geq 0 \right\}.$$

  where β_i are the weights normalized to add up to 1, ie. $\beta_i = \alpha_i / (\sum_j \alpha_j)$ for $i = 1, \ldots, n_l$. The name n_l reads as "n left", the length of the product on the left-hand side of the definition.

- *appenddualpowerconedomain* : the **dual power cone domain** is the subset of $\mathbb{R}^n$ defined as

$$\left(\mathcal{P}_n^{(\alpha_1,\ldots,\alpha_{n_l})}\right)^* = \left\{ x \in \mathbb{R}^n \ : \ \prod_{i=1}^{n_l} \left(\frac{x_i}{\beta_i}\right)^{\beta_i} \geq \sqrt{x_{n_l+1}^2 + \cdots + x_n^2},\ x_1, \ldots, x_{n_l} \geq 0 \right\}.$$

  where β_i are the weights normalized to add up to 1, ie. $\beta_i = \alpha_i / (\sum_j \alpha_j)$ for $i = 1, \ldots, n_l$. The name n_l reads as "n left", the length of the product on the left-hand side of the definition.

- **Remark:** in MOSEK 9 power cones were available only in the special case with $n_l = 2$ and weights $(\alpha, 1-\alpha)$ for some $0 < \alpha < 1$ specified as cone parameter.

15.8.5 Geometric mean cone domains

A geometric mean cone domain is determined by the dimension n.

- *appendprimalgeomeanconedomain* : the **primal geometric mean cone domain** is the subset of $\mathbb{R}^n$ defined as

$$\mathcal{GM}^n = \left\{ x \in \mathbb{R}^n \ : \ \left(\prod_{i=1}^{n-1} x_i\right)^{1/(n-1)} \geq |x_n|,\ x_1, \ldots, x_{n-1} \geq 0 \right\}.$$

  It is a special case of the primal power cone domain with $n_l = n-1$ and weights $\alpha = (1, \ldots, 1)$.

- *appenddualgeomeanconedomain* : the **dual geometric mean cone domain** is the subset of $\mathbb{R}^n$ defined as

$$(\mathcal{GM}^n)^* = \left\{ x \in \mathbb{R}^n \ : \ (n-1)\left(\prod_{i=1}^{n-1} x_i\right)^{1/(n-1)} \geq |x_n|,\ x_1, \ldots, x_{n-1} \geq 0 \right\}.$$

  It is a special case of the dual power cone domain with $n_l = n-1$ and weights $\alpha = (1, \ldots, 1)$.

15.8.6 Vectorized semidefinite domain

- *appendsvecpsdconedomain* : the **vectorized PSD cone domain** is determined by the dimension n, which must be of the form $n = d(d+1)/2$. Then the domain is defined as

$$\mathcal{S}_+^{d,\mathrm{vec}} = \left\{(x_1, \ldots, x_{d(d+1)/2}) \in \mathbb{R}^n \ : \ \mathrm{sMat}(x) \in \mathcal{S}_+^d\right\},$$

where

$$\mathrm{sMat}(x) = \begin{bmatrix} x_1 & x_2/\sqrt{2} & \cdots & x_d/\sqrt{2} \\ x_2/\sqrt{2} & x_{d+1} & \cdots & x_{2d-1}/\sqrt{2} \\ \cdots & \cdots & \cdots & \cdots \\ x_d/\sqrt{2} & x_{2d-1}/\sqrt{2} & \cdots & x_{d(d+1)/2} \end{bmatrix},$$

or equivalently

$$\mathcal{S}_+^{d,\mathrm{vec}} = \left\{\mathrm{sVec}(X) \ : \ X \in \mathcal{S}_+^d\right\},$$

where

$$\mathrm{sVec}(X) = (X_{11}, \sqrt{2}X_{21}, \ldots, \sqrt{2}X_{d1}, X_{22}, \sqrt{2}X_{32}, \ldots, X_{dd}).$$

In other words, the domain consists of vectorizations of the lower-triangular part of a positive semidefinite matrix, with the non-diagonal elements additionally rescaled.

15.9 Environment variables

This section lists operating system environment variables which can globally affect the behavior of **MOSEK**.

It is recommended that any environment variables are set in the environment *before* launching a process using **MOSEK**, or at the very least before the first time any **MOSEK** library or package is loaded by the process.

- `MOSEKLM_LICENSE_FILE` - location of license file.

  Commonly used to point **MOSEK** to a floating license token server or change the default license file search path. For details see the licensing guide.

- `MOSEK_SYS_NUM_CORES` - set the number of cores.

  When **MOSEK** is loaded it detects the number of cores available on the machine. Setting this environment variable overrides that detection.

  Most users will never need it. Typical applications would be:

  - When **MOSEK** fails to detect the number of cores correctly.
  - To limit the default number of cores available to **MOSEK** in a way transparent to the users.

- `PATH` - system search path.

  In all automated cases (MSI, package managers) the installation process will ensure that the **MOSEK** binaries can be located on runtime, either by adding them to the system search path or via other mechanisms.

  It may be needed to set up by hand for manual, modified or other custom installations.

- `LD_LIBRARY_PATH`, `DYLD_LIBRARY_PATH` - shared objects search path.

  Affects the locations where loader looks for shared libraries. Should never be needed for a correct installation.

- `MIMALLOC_PURGE_DELAY` - mimalloc page release delay.

  Setting it to `0` gives the most aggressive memory release behavior at the cost of speed.

  Most users should never change it. Setting `0` may decrease memory consumption in some special scenarios. Known cases include optimizing very large models many times in a row in the same process. Do not consider it unless memory use becomes an actual issue. Usage at own risk.

Chapter 16

Supported File Formats

MOSEK supports a range of problem and solution formats listed in Table 16.1 and Table 16.2.
The most important are:

- the **Task format**, **MOSEK**'s native binary format which supports all features that **MOSEK** supports. It is the closest possible representation of the internal data in a task and it is ideal for submitting problem data support questions.
- the **PTF format**, **MOSEK**'s human-readable format that supports all linear, conic and mixed-integer features. It is ideal for debugging. It is not an exact copy of all the data in the task, but it contains all information required to reconstruct it, presented in a readable fashion.
- **MPS**, **LP**, **CBF** formats are industry standards, each supporting some limited set of features, and potentially requiring some degree of reformulation during read/write.

Problem formats

Table 16.1: List of supported file formats for optimization problems.

Format Type	Ext.	Binary/Text	LP	QCQO	ACC	SDP	DJC	Sol	Param
LP	lp	plain text	X	X					
MPS	mps	plain text	X	X					
PTF	ptf	plain text	X		X	X	X	X	X
CBF	cbf	plain text	X		X	X			
Task format	task	binary	X	X	X	X	X	X	X
Jtask format	jtask	text/JSON	X	X	X	X	X	X	X
OPF (deprecated for conic problems)	opf	plain text	X	X				X	X

The columns of the table indicate if the specified file format supports:

- LP - linear problems, possibly with integer variables,
- QCQO - quadratic objective or constraints,
- ACC - affine conic constraints,
- SDP - semidefinite cone/variables,
- DJC - disjunctive constraints,
- Sol - solutions,
- Param - optimizer parameters.

Solution formats

Table 16.2: List of supported solution formats.

Format Type	Ext.	Binary/Text	Description
SOL	sol	plain text	Interior Solution
	bas	plain text	Basic Solution
	int	plain text	Integer
Jsol format	jsol	text/JSON	All solutions

Compression

MOSEK supports GZIP and Zstandard compression. Problem files with extension `.gz` (for GZIP) and `.zst` (for Zstandard) are assumed to be compressed when read, and are automatically compressed when written. For example, a file called

```
problem.mps.zst
```

will be considered as a Zstandard compressed MPS file.

16.1 The LP File Format

MOSEK supports the LP file format with some extensions. The LP format is not a completely well-defined standard and hence different optimization packages may interpret the same LP file in slightly different ways. **MOSEK** tries to emulate as closely as possible CPLEX's behavior, but tries to stay backward compatible.

The LP file format can specify problems of the form

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

where

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

$$q^o(x) = x^T Q^o x$$

and it is assumed that

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

- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $l^c \in \mathbb{R}^m$ is the lower limit on the activity for the constraints.
- $u^c \in \mathbb{R}^m$ is the upper limit on the activity for the constraints.
- $l^x \in \mathbb{R}^n$ is the lower limit on the activity for the variables.
- $u^x \in \mathbb{R}^n$ is the upper limit on the activity for the variables.
- $q : \mathbb{R}^n \to \mathbb{R}$ is a vector of quadratic functions. Hence,

$$q_i(x) = x^T Q^i x$$

where it is assumed that

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

- $\mathcal{J} \subseteq \{1, 2, \ldots, n\}$ is an index set of the integer constrained variables.

16.1.1 File Sections

An LP formatted file contains a number of sections specifying the objective, constraints, variable bounds, and variable types. The section keywords may be any mix of upper and lower case letters.

Objective Function

The first section beginning with one of the keywords

```
max
maximum
maximize
min
minimum
minimize
```

defines the objective sense and the objective function, i.e.

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

The objective may be given a name by writing

```
myname:
```

before the expressions.
The objective function contains linear and quadratic terms. The linear terms are written as

```
4 x1 + x2 - 0.1 x3
```

and so forth. The quadratic terms are written in square brackets (`[ ]/2`) and are either squared or multiplied as in the examples

```
x1^2
```

and

```
x1 * x2
```

There may be zero or more pairs of brackets containing quadratic expressions.
An example of an objective section is

```
minimize
myobj: 4 x1 + x2 - 0.1 x3 + [ x1^2 + 2.1 x1 * x2 ]/2
```

Please note that the quadratic expressions are multiplied with $\frac{1}{2}$, so that the above expression means

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

If the same variable occurs more than once in the linear part, the coefficients are added, so that `4 x1 + 2 x1` is equivalent to `6 x1`. In the quadratic expressions `x1 * x2` is equivalent to `x2 * x1` and, as in the linear part, if the same variables multiplied or squared occur several times their coefficients are added.

Constraints

The second section beginning with one of the keywords

```
subj to
subject to
s.t.
st
```

defines the linear constraint matrix A and the quadratic matrices Q^i.

A constraint contains a name (optional), expressions adhering to the same rules as in the objective and a bound:

```
subject to
con1: x1 + x2 + [ x3^2 ]/2 <= 5.1
```

The bound type (here `<=`) may be any of `<`, `<=`, `=`, `>`, `>=` (`<` and `<=` mean the same), and the bound may be any number.

Ranged constraints cannot be written in LP format, and have to be split into a separate upper and lower bound.

Bounds

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

```
bound
bounds
```

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

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

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

Variable Types

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

```
bin
binaries
binary
```

and

```
gen
general
```

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

```
general
x1 x2
```

(continues on next page)

(continued from previous page)

```
binary
x3 x4
```

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

Terminating Section

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

```
end
```

16.1.2 LP File Examples

Linear example `lo1.lp`

```
\ File: lo1.lp
maximize
obj: 3 x1 + x2 + 5 x3 + x4
subject to
c1:  3 x1 + x2 + 2 x3 = 30
c2:  2 x1 + x2 + 3 x3 + x4 >= 15
c3:  2 x2 + 3 x4 <= 25
bounds
 0 <= x1 <= +infinity
 0 <= x2 <= 10
 0 <= x3 <= +infinity
 0 <= x4 <= +infinity
end
```

Mixed integer example `milo1.lp`

```
maximize
obj: x1 + 6.4e-01 x2
subject to
c1:  5e+01 x1 + 3.1e+01 x2 <= 2.5e+02
c2:  3e+00 x1 - 2e+00 x2 >= -4e+00
bounds
 0 <= x1 <= +infinity
 0 <= x2 <= +infinity
general
 x1 x2
end
```

16.1.3 LP Format peculiarities

Comments

Anything on a line after a \ is ignored and is treated as a comment.

Names

A name for an objective, a constraint or a variable may contain the letters `a-z`, `A-Z`, the digits `0-9` and the characters

```
!"#$%&()/,.;?@_'`|~
```

The first character in a name must not be a number, a period or the letter `e` or `E`. Keywords must not be used as names.

MOSEK accepts any character as valid for names, except `\0`. A name that is not allowed in LP file will be changed and a warning will be issued.

The algorithm for making names LP valid works as follows: The name is interpreted as an `utf-8` string. For a Unicode character `c`:

- If `c==_` (underscore), the output is `__` (two underscores).
- If `c` is a valid LP name character, the output is just `c`.
- If `c` is another character in the ASCII range, the output is `_XX`, where `XX` is the hexadecimal code for the character.
- If `c` is a character in the range *127-65535*, the output is `_uXXXX`, where `XXXX` is the hexadecimal code for the character.
- If `c` is a character above 65535, the output is `_UXXXXXXXX`, where `XXXXXXXX` is the hexadecimal code for the character.

Invalid `utf-8` substrings are escaped as `_XX'`, and if a name starts with a period, `e` or `E`, that character is escaped as `_XX`.

Variable Bounds

Specifying several upper or lower bounds on one variable is possible but **MOSEK** uses only the tightest bounds. If a variable is fixed (with `=`), then it is considered the tightest bound.

16.2 The MPS File Format

MOSEK supports the standard MPS format with some extensions. For a detailed description of the MPS format see the book by Nazareth [Naz87].

16.2.1 MPS File Structure

The version of the MPS format supported by **MOSEK** allows specification of an optimization problem of the form

$$\begin{array}{rccccl} \text{maximize/minimize} & & c^T x + q_0(x) & & & \\ l^c & \leq & Ax + q(x) & \leq & u^c, & \\ l^x & \leq & x & \leq & u^x, & \\ & & x \in \mathcal{K}, & & & \\ & & x_{\mathcal{J}} \ \text{integer}, & & & \end{array} \tag{16.1}$$

where

- $x \in \mathbb{R}^n$ is the vector of decision variables.
- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $l^c \in \mathbb{R}^m$ is the lower limit on the activity for the constraints.
- $u^c \in \mathbb{R}^m$ is the upper limit on the activity for the constraints.
- $l^x \in \mathbb{R}^n$ is the lower limit on the activity for the variables.
- $u^x \in \mathbb{R}^n$ is the upper limit on the activity for the variables.

- $q : \mathbb{R}^n \to \mathbb{R}$ is a vector of quadratic functions. Hence,

$$q_i(x) = \frac{1}{2} x^T Q^i x$$

  where it is assumed that $Q^i = (Q^i)^T$. Please note the explicit $\frac{1}{2}$ in the quadratic term and that Q^i is required to be symmetric. The same applies to q_0.

- $\mathcal{K}$ is a convex cone.
- $\mathcal{J} \subseteq \{1, 2, \ldots, n\}$ is an index set of the integer-constrained variables.
- c is the vector of objective coefficients.

An MPS file with one row and one column can be illustrated like this:

```
*        1         2         3         4         5         6
*234567890123456789012345678901234567890123456789012345678901234567890
NAME          [name]
OBJSENSE
    [objsense]
OBJNAME       [objname]
ROWS
 ?  [cname1]
COLUMNS
    [vname1]  [cname1]  [value1]       [cname2]  [value2]
RHS
    [name]    [cname1]  [value1]       [cname2]  [value2]
RANGES
    [name]    [cname1]  [value1]       [cname2]  [value2]
QSECTION      [cname1]
    [vname1]  [vname2]  [value1]       [vname3]  [value2]
QMATRIX
    [vname1]  [vname2]  [value1]
QUADOBJ
    [vname1]  [vname2]  [value1]
QCMATRIX      [cname1]
    [vname1]  [vname2]  [value1]
BOUNDS
 ?? [name]    [vname1]  [value1]
CSECTION      [kname1]  [value1]       [ktype]
    [vname1]
ENDATA
```

Here the names in capitals are keywords of the MPS format and names in brackets are custom defined names or values. A couple of notes on the structure:

- Fields: All items surrounded by brackets appear in *fields*. The fields named "`valueN`" are numerical values. Hence, they must have the format

  ```
  [+|-]XXXXXXX.XXXXXX[[e|E][+|-]XXX]
  ```

  where

  ```
  X = [0|1|2|3|4|5|6|7|8|9].
  ```

- Sections: The MPS file consists of several sections where the names in capitals indicate the beginning of a new section. For example, `COLUMNS` denotes the beginning of the columns section.
- Comments: Lines starting with an `*` are comment lines and are ignored by **MOSEK**.
- Keys: The question marks represent keys to be specified later.

- Extensions: The sections `QSECTION` and `CSECTION` are specific **MOSEK** extensions of the MPS format. The sections `QMATRIX`, `QUADOBJ` and `QCMATRIX` are included for sake of compatibility with other vendors extensions to the MPS format.
- The standard MPS format is a fixed format, i.e. everything in the MPS file must be within certain fixed positions. **MOSEK** also supports a *free format*. See Sec. 16.2.5 for details.

Linear example `lo1.mps`

A concrete example of a MPS file is presented below:

```
* File: lo1.mps
NAME          lo1
OBJSENSE
    MAX
ROWS
 N  obj
 E  c1
 G  c2
 L  c3
COLUMNS
    x1        obj       3
    x1        c1        3
    x1        c2        2
    x2        obj       1
    x2        c1        1
    x2        c2        1
    x2        c3        2
    x3        obj       5
    x3        c1        2
    x3        c2        3
    x4        obj       1
    x4        c2        1
    x4        c3        3
RHS
    rhs       c1        30
    rhs       c2        15
    rhs       c3        25
RANGES
BOUNDS
 UP bound     x2        10
ENDATA
```

Subsequently each individual section in the MPS format is discussed.

NAME (optional)

In this section a name ([name]) is assigned to the problem.

OBJSENSE (optional)

This is an optional section that can be used to specify the sense of the objective function. The OBJSENSE section contains one line at most which can be one of the following:

```
MIN
MINIMIZE
MAX
MAXIMIZE
```

It should be obvious what the implication is of each of these four lines.

OBJNAME (optional)

This is an optional section that can be used to specify the name of the row that is used as objective function. objname should be a valid row name.

ROWS

A record in the ROWS section has the form

```
?  [cname1]
```

where the requirements for the fields are as follows:

Field	Starting Position	Max Width	required	Description
?	2	1	Yes	Constraint key
[cname1]	5	8	Yes	Constraint name

Hence, in this section each constraint is assigned a unique name denoted by [cname1]. Please note that [cname1] starts in position 5 and the field can be at most 8 characters wide. An initial key ? must be present to specify the type of the constraint. The key can have values E, G, L, or N with the following interpretation:

Constraint type	l_i^c	u_i^c
E (equal)	finite	$= l_i^c$
G (greater)	finite	∞
L (lower)	$-\infty$	finite
N (none)	$-\infty$	∞

In the MPS format the objective vector is not specified explicitly, but one of the constraints having the key N will be used as the objective vector c . In general, if multiple N type constraints are specified, then the first will be used as the objective vector c, unless something else was specified in the section OBJNAME.

COLUMNS

In this section the elements of A are specified using one or more records having the form:

```
[vname1]   [cname1]     [value1]      [cname2]   [value2]
```

where the requirements for each field are as follows:

Field	Starting Position	Max Width	required	Description
[vname1]	5	8	Yes	Variable name
[cname1]	15	8	Yes	Constraint name
[value1]	25	12	Yes	Numerical value
[cname2]	40	8	No	Constraint name
[value2]	50	12	No	Numerical value

Hence, a record specifies one or two elements a_{ij} of A using the principle that [vname1] and [cname1] determines j and i respectively. Please note that [cname1] must be a constraint name specified in the ROWS section. Finally, [value1] denotes the numerical value of a_{ij} . Another optional element is specified by [cname2], and [value2] for the variable specified by [vname1]. Some important comments are:

- All elements belonging to one variable must be grouped together.
- Zero elements of A should not be specified.
- At least one element for each variable should be specified.

RHS (optional)

A record in this section has the format

```
[name]    [cname1]    [value1]    [cname2]  [value2]
```

where the requirements for each field are as follows:

Field	Starting Position	Max Width	required	Description
[name]	5	8	Yes	Name of the RHS vector
[cname1]	15	8	Yes	Constraint name
[value1]	25	12	Yes	Numerical value
[cname2]	40	8	No	Constraint name
[value2]	50	12	No	Numerical value

The interpretation of a record is that [name] is the name of the RHS vector to be specified. In general, several vectors can be specified. [cname1] denotes a constraint name previously specified in the ROWS section. Now, assume that this name has been assigned to the i -h constraint and v_1 denotes the value specified by [value1], then the interpretation of v_1 is:

Constraint	l_i^c	u_i^c
E	v_1	v_1
G	v_1	
L		v_1
N		

An optional second element is specified by [cname2] and [value2] and is interpreted in the same way. Please note that it is not necessary to specify zero elements, because elements are assumed to be zero.

RANGES (optional)

A record in this section has the form

```
[name]    [cname1]    [value1]    [cname2]  [value2]
```

where the requirements for each fields are as follows:

Field	Starting Position	Max Width	required	Description
[name]	5	8	Yes	Name of the RANGE vector
[cname1]	15	8	Yes	Constraint name
[value1]	25	12	Yes	Numerical value
[cname2]	40	8	No	Constraint name
[value2]	50	12	No	Numerical value

The records in this section are used to modify the bound vectors for the constraints, i.e. the values in l^c and u^c . A record has the following interpretation:[name] is the name of the RANGE vector and [cname1] is a valid constraint name. Assume that [cname1] is assigned to the i-th constraint and let v_1 be the value specified by [value1], then a record has the interpretation:

Constraint type	Sign of v_1	l_i^c	u_i^c
E	$-$	$u_i^c + v_1$	
E	$+$		$l_i^c + v_1$
G	$-$ or $+$		$l_i^c + \|v_1\|$
L	$-$ or $+$	$u_i^c - \|v_1\|$	
N			

Another constraint bound can optionally be modified using [cname2] and [value2] the same way.

QSECTION (optional)

Within the QSECTION the label [cname1] must be a constraint name previously specified in the ROWS section. The label [cname1] denotes the constraint to which the quadratic terms belong. A record in the QSECTION has the form

```
[vname1]  [vname2]    [value1]    [vname3]  [value2]
```

where the requirements for each field are:

Field	Starting Position	Max Width	required	Description
[vname1]	5	8	Yes	Variable name
[vname2]	15	8	Yes	Variable name
[value1]	25	12	Yes	Numerical value
[vname3]	40	8	No	Variable name
[value2]	50	12	No	Numerical value

A record specifies one or two elements in the lower triangular part of the Q^i matrix where [cname1] specifies the i . Hence, if the names [vname1] and [vname2] have been assigned to the k-th and j-th variable, then Q^i_{kj} is assigned the value given by [value1] An optional second element is specified in the same way by the fields [vname1], [vname3], and [value2].

The example

$$\begin{array}{lcll} \text{minimize} & -x_2 + \frac{1}{2}(2x_1^2 - 2x_1x_3 + 0.2x_2^2 + 2x_3^2) & & \\ \text{subject to} & x_1 + x_2 + x_3 & \geq & 1, \\ & x \geq 0 & & \end{array}$$

has the following MPS file representation

```
* File: qo1.mps
NAME          qo1
ROWS
 N  obj
 G  c1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
QSECTION      obj
    x1        x1        2.0
    x1        x3        -1.0
    x2        x2        0.2
    x3        x3        2.0
ENDATA
```

Regarding the QSECTIONs please note that:

- Only one QSECTION is allowed for each constraint.
- The QSECTIONs can appear in an arbitrary order after the COLUMNS section.
- All variable names occurring in the QSECTION must already be specified in the COLUMNS section.
- All entries specified in a QSECTION are assumed to belong to the lower triangular part of the quadratic term of Q .

QMATRIX/QUADOBJ (optional)

The QMATRIX and QUADOBJ sections allow to define the quadratic term of the objective function. They differ in how the quadratic term of the objective function is stored:

- QMATRIX stores all the nonzeros coefficients, without taking advantage of the symmetry of the Q matrix.
- QUADOBJ stores the upper diagonal nonzero elements of the Q matrix.

A record in both sections has the form:

```
[vname1]   [vname2]     [value1]
```

where the requirements for each field are:

Field	Starting Position	Max Width	required	Description
[vname1]	5	8	Yes	Variable name
[vname2]	15	8	Yes	Variable name
[value1]	25	12	Yes	Numerical value

A record specifies one elements of the Q matrix in the objective function . Hence, if the names [vname1] and [vname2] have been assigned to the k-th and j-th variable, then Q_{kj} is assigned the value given by [value1]. Note that a line must appear for each off-diagonal coefficient if using a QMATRIX section, while only one entry is required in a QUADOBJ section. The quadratic part of the objective function will be evaluated as $1/2x^TQx$.

The example

$$\begin{array}{lcll} \text{minimize} & -x_2 + \frac{1}{2}(2x_1^2 - 2x_1x_3 + 0.2x_2^2 + 2x_3^2) & & \\ \text{subject to} & x_1 + x_2 + x_3 & \geq & 1, \\ & x \geq 0 & & \end{array}$$

has the following MPS file representation using QMATRIX

```
* File: qo1_matrix.mps
NAME          qo1_qmatrix
ROWS
 N  obj
 G  c1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
QMATRIX
    x1        x1        2.0
    x1        x3        -1.0
    x3        x1        -1.0
    x2        x2        0.2
    x3        x3        2.0
ENDATA
```

or the following using `QUADOBJ`

```
* File: qo1_quadobj.mps
NAME          qo1_quadobj
ROWS
 N  obj
 G  c1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
QUADOBJ
    x1        x1        2.0
    x1        x3        -1.0
    x2        x2        0.2
    x3        x3        2.0
ENDATA
```

Please also note that:

- A `QMATRIX`/`QUADOBJ` section can appear in an arbitrary order after the `COLUMNS` section.
- All variable names occurring in the `QMATRIX`/`QUADOBJ` section must already be specified in the `COLUMNS` section.

`QCMATRIX` (optional)

A `QCMATRIX` section allows to specify the quadratic part of a given constraint. Within the `QCMATRIX` the label `[cname1]` must be a constraint name previously specified in the `ROWS` section. The label `[cname1]` denotes the constraint to which the quadratic term belongs. A record in the `QSECTION` has the form

```
[vname1]  [vname2]    [value1]
```

where the requirements for each field are:

Field	Starting Position	Max Width	required	Description
[vname1]	5	8	Yes	Variable name
[vname2]	15	8	Yes	Variable name
[value1]	25	12	Yes	Numerical value

A record specifies an entry of the Q^i matrix where [cname1] specifies the i. Hence, if the names [vname1] and [vname2] have been assigned to the k-th and j-th variable, then Q^i_{kj} is assigned the value given by [value1]. Moreover, the quadratic term is represented as $1/2x^TQx$.

The example

$$\begin{array}{lcll} \text{minimize} & x_2 & & \\ \text{subject to} & x_1 + x_2 + x_3 & \geq & 1, \\ & \frac{1}{2}(-2x_1x_3 + 0.2x_2^2 + 2x_3^2) & \leq & 10, \\ & x \geq 0 & & \end{array}$$

has the following MPS file representation

```
* File: qo1.mps
NAME          qo1
ROWS
 N  obj
 G  c1
 L  q1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
    rhs       q1        10.0
QCMATRIX      q1
    x1        x1        2.0
    x1        x3        -1.0
    x3        x1        -1.0
    x2        x2        0.2
    x3        x3        2.0
ENDATA
```

Regarding the QCMATRIXs please note that:

- Only one QCMATRIX is allowed for each constraint.
- The QCMATRIXs can appear in an arbitrary order after the COLUMNS section.
- All variable names occurring in the QSECTION must already be specified in the COLUMNS section.
- QCMATRIX does not exploit the symmetry of Q: an off-diagonal entry (i, j) should appear twice.

BOUNDS (optional)

In the BOUNDS section changes to the default bounds vectors l^x and u^x are specified. The default bounds vectors are $l^x = 0$ and $u^x = \infty$. Moreover, it is possible to specify several sets of bound vectors. A record in this section has the form

```
?? [name]     [vname1]     [value1]
```

where the requirements for each field are:

Field	Starting Position	Max Width	Required	Description
??	2	2	Yes	Bound key
[name]	5	8	Yes	Name of the BOUNDS vector
[vname1]	15	8	Yes	Variable name
[value1]	25	12	No	Numerical value

Hence, a record in the BOUNDS section has the following interpretation: [name] is the name of the bound vector and [vname1] is the name of the variable for which the bounds are modified by the record. ?? and [value1] are used to modify the bound vectors according to the following table:

??	l_j^x	u_j^x	Made integer (added to $\mathcal{J}$)
FR	$-\infty$	∞	No
FX	v_1	v_1	No
LO	v_1	unchanged	No
MI	$-\infty$	unchanged	No
PL	unchanged	∞	No
UP	unchanged	v_1	No
BV	0	1	Yes
LI	$\lceil v_1 \rceil$	unchanged	Yes
UI	unchanged	$\lfloor v_1 \rfloor$	Yes

Here v_1 is the value specified by [value1].

CSECTION (optional)

The purpose of the CSECTION is to specify the conic constraint

$$x \in \mathcal{K}$$

in (16.1). It is assumed that $\mathcal{K}$ satisfies the following requirements. Let

$$x^t \in \mathbb{R}^{n^t}, \quad t = 1, \ldots, k$$

be vectors comprised of parts of the decision variables x so that each decision variable is a member of exactly **one** vector x^t, for example

$$x^1 = \begin{bmatrix} x_1 \\ x_4 \\ x_7 \end{bmatrix} \quad \text{and} \quad x^2 = \begin{bmatrix} x_6 \\ x_5 \\ x_3 \\ x_2 \end{bmatrix}.$$

Next define

$$\mathcal{K} := \left\{ x \in \mathbb{R}^n : \quad x^t \in \mathcal{K}_t, \quad t = 1, \ldots, k \right\}$$

where $\mathcal{K}_t$ must have one of the following forms:

- $\mathbb{R}$ set:

$$\mathcal{K}_t = \mathbb{R}^{n^t}.$$

- Zero cone:

$$\mathcal{K}_t = \{0\} \subseteq \mathbb{R}^{n^t}. \tag{16.2}$$

- Quadratic cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : x_1 \geq \sqrt{\sum_{j=2}^{n^t} x_j^2} \right\}. \tag{16.3}$$

- Rotated quadratic cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : 2x_1 x_2 \geq \sum_{j=3}^{n^t} x_j^2, \quad x_1, x_2 \geq 0 \right\}. \tag{16.4}$$

- Primal exponential cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^3 : x_1 \geq x_2 \exp(x_3/x_2), \quad x_1, x_2 \geq 0 \right\}. \tag{16.5}$$

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

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : x_1^{\alpha} x_2^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n^t} x_j^2}, \quad x_1, x_2 \geq 0 \right\}. \tag{16.6}$$

- Dual exponential cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^3 : x_1 \geq -x_3 e^{-1} \exp(x_2/x_3), \quad x_3 \leq 0, x_1 \geq 0 \right\}. \tag{16.7}$$

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

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : \left(\frac{x_1}{\alpha}\right)^{\alpha} \left(\frac{x_2}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n^t} x_j^2}, \quad x_1, x_2 \geq 0 \right\}. \tag{16.8}$$

In general, membership in the $\mathbb{R}$ set is not specified. If a variable is not a member of any other cone then it is assumed to be a member of the $\mathbb{R}$ cone.

Next, let us study an example. Assume that the power cone

$$x_4^{1/3} x_5^{2/3} \geq |x_8|$$

and the rotated quadratic cone

$$2x_3 x_7 \geq x_1^2 + x_0^2, \quad x_3, x_7 \geq 0,$$

should be specified in the MPS file. One `CSECTION` is required for each cone and they are specified as follows:

```
*        1         2         3         4         5         6
*2345678901234567890123456789012345678901234567890123456789012345678901234567890
CSECTION      konea     3e-1            PPOW
x4
x5
x8
CSECTION      koneb     0.0             RQUAD
x7
x3
x1
x0
```

In general, a `CSECTION` header has the format

```
CSECTION      [kname1]     [value1]     [ktype]
```

where the requirements for each field are as follows:

Field	Starting Position	Max Width	Required	Description
`[kname1]`	15	8	Yes	Name of the cone
`[value1]`	25	12	No	Cone parameter
`[ktype]`	40		Yes	Type of the cone.

The possible cone type keys are:

[ktype]	Members	[value1]	Interpretation.
ZERO	≥ 0	unused	Zero cone (16.2).
QUAD	≥ 1	unused	Quadratic cone (16.3).
RQUAD	≥ 2	unused	Rotated quadratic cone (16.4).
PEXP	3	unused	Primal exponential cone (16.5).
PPOW	≥ 2	α	Primal power cone (16.6).
DEXP	3	unused	Dual exponential cone (16.7).
DPOW	≥ 2	α	Dual power cone (16.8).

A record in the CSECTION has the format

```
[vname1]
```

where the requirements for each field are

Field	Starting Position	Max Width	required	Description
[vname1]	5	8	Yes	A valid variable name

A variable must occur in at most one CSECTION.

ENDATA

This keyword denotes the end of the MPS file.

16.2.2 Integer Variables

Using special bound keys in the BOUNDS section it is possible to specify that some or all of the variables should be integer-constrained i.e. be members of $\mathcal{J}$. However, an alternative method is available. This method is available only for backward compatibility and we recommend that it is not used. This method requires that markers are placed in the COLUMNS section as in the example:

```
COLUMNS
x1        obj       -10.0          c1        0.7
x1        c2        0.5            c3        1.0
x1        c4        0.1
* Start of integer-constrained variables.
MARK000   'MARKER'                 'INTORG'
x2        obj       -9.0           c1        1.0
x2        c2        0.8333333333   c3        0.66666667
x2        c4        0.25
x3        obj       1.0            c6        2.0
MARK001   'MARKER'                 'INTEND'
* End of integer-constrained variables.
```

Please note that special marker lines are used to indicate the start and the end of the integer variables. Furthermore be aware of the following

- All variables between the markers are assigned a default lower bound of 0 and a default upper bound of 1. **This may not be what is intended.** If it is not intended, the correct bounds should be defined in the BOUNDS section of the MPS formatted file.
- **MOSEK** ignores field 1, i.e. MARK0001 and MARK001, however, other optimization systems require them.
- Field 2, i.e. MARKER, must be specified including the single quotes. This implies that no row can be assigned the name MARKER.
- Field 3 is ignored and should be left blank.

- Field 4, i.e. `INTORG` and `INTEND`, must be specified.
- It is possible to specify several such integer marker sections within the `COLUMNS` section.

16.2.3 General Limitations

- An MPS file should be an ASCII file.

16.2.4 Interpretation of the MPS Format

Several issues related to the MPS format are not well-defined by the industry standard. However, **MOSEK** uses the following interpretation:

- If a matrix element in the `COLUMNS` section is specified multiple times, then the multiple entries are added together.
- If a matrix element in a `QSECTION` section is specified multiple times, then the multiple entries are added together.

16.2.5 The Free MPS Format

MOSEK supports a free format variation of the MPS format. The free format is similar to the MPS file format but less restrictive, e.g. it allows longer names. However, a name must not contain any blanks.

Moreover, by default a line in the MPS file must not contain more than 1024 characters. By modifying the parameter *MSK_IPAR_READ_MPS_WIDTH* an arbitrary large line width will be accepted.

The free MPS format is default. To change to the strict and other formats use the parameter *MSK_IPAR_READ_MPS_FORMAT*.

> **Warning:** This file format is to a large extent deprecated. While it can still be used for linear and quadratic problems, for conic problems the Sec. 16.5 is recommended.

16.3 The OPF Format

The *Optimization Problem Format (OPF)* is an alternative to LP and MPS files for specifying optimization problems. It is row-oriented, inspired by the CPLEX LP format.

Apart from containing objective, constraints, bounds etc. it may contain complete or partial solutions, comments and extra information relevant for solving the problem. It is designed to be easily read and modified by hand and to be forward compatible with possible future extensions.

Intended use

The OPF file format is meant to replace several other files:

- The LP file format: Any problem that can be written as an LP file can be written as an OPF file too; furthermore it naturally accommodates ranged constraints and variables as well as arbitrary characters in names, fixed expressions in the objective, empty constraints, and conic constraints.
- Parameter files: It is possible to specify integer, double and string parameters along with the problem (or in a separate OPF file).
- Solution files: It is possible to store a full or a partial solution in an OPF file and later reload it.

16.3.1 The File Format

The format uses tags to structure data. A simple example with the basic sections may look like this:

```
[comment]
This is a comment. You may write almost anything here...
[/comment]

# This is a single-line comment.

[objective min 'myobj']
x + 3 y + x^2 + 3 y^2 + z + 1
[/objective]

[constraints]
[con 'con01'] 4 <= x + y  [/con]
[/constraints]

[bounds]
[b] -10 <= x,y <= 10  [/b]

[cone quad] x,y,z [/cone]
[/bounds]
```

A scope is opened by a tag of the form `[tag]` and closed by a tag of the form `[/tag]`. An opening tag may accept a list of unnamed and named arguments, for examples:

```
[tag value] tag with one unnamed argument [/tag]
[tag arg=value] tag with one named argument [/tag]
```

Unnamed arguments are identified by their order, while named arguments may appear in any order, but never before an unnamed argument. The `value` can be a quoted, single-quoted or double-quoted text string, i.e.

```
[tag 'value']     single-quoted value [/tag]
[tag arg='value'] single-quoted value [/tag]
[tag "value"]     double-quoted value [/tag]
[tag arg="value"] double-quoted value [/tag]
```

16.3.2 Sections

The recognized tags are

[comment]

A comment section. This can contain *almost* any text: Between single quotes (') or double quotes (") any text may appear. Outside quotes the markup characters ([and]) must be prefixed by backslashes. Both single and double quotes may appear alone or inside a pair of quotes if it is prefixed by a backslash.

[objective]

The objective function: This accepts one or two parameters, where the first one (in the above example `min`) is either `min` or `max` (regardless of case) and defines the objective sense, and the second one (above `myobj`), if present, is the objective name. The section may contain linear and quadratic expressions.

If several objectives are specified, all but the last are ignored.

[constraints]

This does not directly contain any data, but may contain subsections `con` defining a linear constraint.

[con]

Defines a single constraint; if an argument is present (`[con NAME]`) this is used as the name of the constraint, otherwise it is given a null-name. The section contains a constraint definition written as linear and quadratic expressions with a lower bound, an upper bound, with both or with an equality. Examples:

```
[constraints]
[con 'con1'] 0 <= x + y       [/con]
[con 'con2'] 0 >= x + y       [/con]
[con 'con3'] 0 <= x + y <= 10 [/con]
[con 'con4']      x + y  = 10 [/con]
[/constraints]
```

Constraint names are unique. If a constraint is specified which has the same name as a previously defined constraint, the new constraint replaces the existing one.

[bounds]

This does not directly contain any data, but may contain subsections `b` (linear bounds on variables) and `cone` (cones).

[b]

Bound definition on one or several variables separated by comma (`,`). An upper or lower bound on a variable replaces any earlier defined bound on that variable. If only one bound (upper or lower) is given only this bound is replaced. This means that upper and lower bounds can be specified separately. So the OPF bound definition:

```
[b]  x,y >= -10  [/b]
[b]  x,y <= 10   [/b]
```

results in the bound $-10 \leq x, y \leq 10$.

`[cone]`

Specifies a cone. A cone is defined as a sequence of variables which belong to a single unique cone. The supported cone types are:

- `quad`: a quadratic cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1^2 \geq \sum_{i=2}^{n} x_i^2, \quad x_1 \geq 0.$$

- `rquad`: a rotated quadratic cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$2x_1 x_2 \geq \sum_{i=3}^{n} x_i^2, \quad x_1, x_2 \geq 0.$$

- `pexp`: primal exponential cone of 3 variables x_1, x_2, x_3 defines a constraint of the form

$$x_1 \geq x_2 \exp(x_3/x_2), \quad x_1, x_2 \geq 0.$$

- `ppow` with parameter $0 < \alpha < 1$: primal power cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1^{\alpha} x_2^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n} x_j^2}, \quad x_1, x_2 \geq 0.$$

- `dexp`: dual exponential cone of 3 variables x_1, x_2, x_3 defines a constraint of the form

$$x_1 \geq -x_3 e^{-1} \exp(x_2/x_3), \quad x_3 \leq 0, x_1 \geq 0.$$

- `dpow` with parameter $0 < \alpha < 1$: dual power cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$\left(\frac{x_1}{\alpha}\right)^{\alpha} \left(\frac{x_2}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n^t} x_j^2}, \quad x_1, x_2 \geq 0.$$

- `zero`: zero cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1 = \cdots = x_n = 0$$

A `[bounds]`-section example:

```
[bounds]
[b]  0 <= x,y <= 10  [/b] # ranged bound
[b] 10 >= x,y >=  0  [/b] # ranged bound
[b]  0 <= x,y <= inf [/b] # using inf
[b]       x,y free   [/b] # free variables
# Let (x,y,z,w) belong to the cone K
[cone rquad] x,y,z,w  [/cone] # rotated quadratic cone
[cone ppow '3e-01' 'a'] x1, x2, x3 [/cone] # power cone with alpha=1/3 and name 'a'
[/bounds]
```

By default all variables are free.

[variables]

This defines an ordering of variables as they should appear in the problem. This is simply a space-separated list of variable names.

[integer]

This contains a space-separated list of variables and defines the constraint that the listed variables must be integer-valued.

[hints]

This may contain only non-essential data; for example estimates of the number of variables, constraints and non-zeros. Placed before all other sections containing data this may reduce the time spent reading the file.

In the `hints` section, any subsection which is not recognized by **MOSEK** is simply ignored. In this section a hint is defined as follows:

```
[hint ITEM] value [/hint]
```

The hints recognized by **MOSEK** are:

- `numvar` (number of variables),
- `numcon` (number of linear/quadratic constraints),
- `numanz` (number of linear non-zeros in constraints),
- `numqnz` (number of quadratic non-zeros in constraints).

[solutions]

This section can contain a set of full or partial solutions to a problem. Each solution must be specified using a `[solution]`-section, i.e.

```
[solutions]
[solution]...[/solution] #solution 1
[solution]...[/solution] #solution 2
#other solutions....
[solution]...[/solution] #solution n
[/solutions]
```

The syntax of a `[solution]`-section is the following:

```
[solution SOLTYPE status=STATUS]...[/solution]
```

where `SOLTYPE` is one of the strings

- `interior`, a non-basic solution,
- `basic`, a basic solution,
- `integer`, an integer solution,

and `STATUS` is one of the strings

- `UNKNOWN`,
- `OPTIMAL`,
- `INTEGER_OPTIMAL`,
- `PRIM_FEAS`,
- `DUAL_FEAS`,
- `PRIM_AND_DUAL_FEAS`,

- NEAR_OPTIMAL,
- NEAR_PRIM_FEAS,
- NEAR_DUAL_FEAS,
- NEAR_PRIM_AND_DUAL_FEAS,
- PRIM_INFEAS_CER,
- DUAL_INFEAS_CER,
- NEAR_PRIM_INFEAS_CER,
- NEAR_DUAL_INFEAS_CER,
- NEAR_INTEGER_OPTIMAL.

Most of these values are irrelevant for input solutions; when constructing a solution for simplex hot-start or an initial solution for a mixed integer problem the safe setting is UNKNOWN.

A [solution]-section contains [con] and [var] sections. Each [con] and [var] section defines solution information for a single variable or constraint, specified as list of KEYWORD/value pairs, in any order, written as

```
KEYWORD=value
```

Allowed keywords are as follows:

- sk. The status of the item, where the value is one of the following strings:
  - LOW, the item is on its lower bound.
  - UPR, the item is on its upper bound.
  - FIX, it is a fixed item.
  - BAS, the item is in the basis.
  - SUPBAS, the item is super basic.
  - UNK, the status is unknown.
  - INF, the item is outside its bounds (infeasible).
- lvl Defines the level of the item.
- sl Defines the level of the dual variable associated with its lower bound.
- su Defines the level of the dual variable associated with its upper bound.
- sn Defines the level of the variable associated with its cone.
- y Defines the level of the corresponding dual variable (for constraints only).

A [var] section should always contain the items sk, lvl, sl and su. Items sl and su are not required for integer solutions.

A [con] section should always contain sk, lvl, sl, su and y.

An example of a solution section

```
[solution basic status=UNKNOWN]
[var x0] sk=LOW    lvl=5.0      [/var]
[var x1] sk=UPR    lvl=10.0     [/var]
[var x2] sk=SUPBAS lvl=2.0  sl=1.5 su=0.0 [/var]

[con c0] sk=LOW    lvl=3.0 y=0.0 [/con]
[con c0] sk=UPR    lvl=0.0 y=5.0 [/con]
[/solution]
```

- `[vendor]` This contains solver/vendor specific data. It accepts one argument, which is a vendor ID – for **MOSEK** the ID is simply `mosek` – and the section contains the subsection `parameters` defining solver parameters. When reading a vendor section, any unknown vendor can be safely ignored. This is described later.

Comments using the `#` may appear anywhere in the file. Between the `#` and the following line-break any text may be written, including markup characters.

16.3.3 Numbers

Numbers, when used for parameter values or coefficients, are written in the usual way by the `printf` function. That is, they may be prefixed by a sign (`+` or `-`) and may contain an integer part, decimal part and an exponent. The decimal point is always `.` (a dot). Some examples are

```
1
1.0
.0
1.
1e10
1e+10
1e-10
```

Some *invalid* examples are

```
e10   # invalid, must contain either integer or decimal part
.     # invalid
.e10  # invalid
```

More formally, the following standard regular expression describes numbers as used:

```
[+|-]?([0-9]+[.][0-9]*|[.][0-9]+)([eE][+|-]?[0-9]+)?
```

16.3.4 Names

Variable names, constraint names and objective name may contain arbitrary characters, which in some cases must be enclosed by quotes (single or double) that in turn must be preceded by a backslash. Unquoted names must begin with a letter (`a-z` or `A-Z`) and contain only the following characters: the letters `a-z` and `A-Z`, the digits `0-9`, braces (`{` and `}`) and underscore (`_`).

Some examples of legal names:

```
an_unquoted_name
another_name{123}
'single quoted name'
"double quoted name"
"name with \\"quote\\" in it"
"name with []s in it"
```

16.3.5 Parameters Section

In the `vendor` section solver parameters are defined inside the `parameters` subsection. Each parameter is written as

```
[p PARAMETER_NAME] value [/p]
```

where `PARAMETER_NAME` is replaced by a **MOSEK** parameter name, usually of the form `MSK_IPAR_...`, `MSK_DPAR_...` or `MSK_SPAR_...`, and the `value` is replaced by the value of that parameter; both integer values and named values may be used. Some simple examples are

```
[vendor mosek]
[parameters]
[p MSK_IPAR_OPF_MAX_TERMS_PER_LINE] 10     [/p]
[p MSK_IPAR_OPF_WRITE_PARAMETERS]   MSK_ON [/p]
[p MSK_DPAR_DATA_TOL_BOUND_INF]     1.0e18 [/p]
[/parameters]
[/vendor]
```

16.3.6 Writing OPF Files from MOSEK

To write an OPF file then make sure the file extension is .opf.

Then modify the following parameters to define what the file should contain:

MSK_IPAR_OPF_WRITE_SOL_BAS	Include basic solution, if defined.
MSK_IPAR_OPF_WRITE_SOL_ITG	Include integer solution, if defined.
MSK_IPAR_OPF_WRITE_SOL_ITR	Include interior solution, if defined.
MSK_IPAR_OPF_WRITE_SOLUTION	Include solutions if they are defined. If this is off, no solutions are included.
MSK_IPAR_OPF_WRITE_HEADER	Include a small header with comments.
MSK_IPAR_OPF_WRITE_PROBLEM	Include the problem itself — objective, constraints and bounds.
MSK_IPAR_OPF_WRITE_PARAMETE	Include all parameter settings.
MSK_IPAR_OPF_WRITE_HINTS	Include hints about the size of the problem.

16.3.7 Examples

This section contains a set of small examples written in OPF and describing how to formulate linear, quadratic and conic problems.

Linear Example `lo1.opf`

Consider the example:

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

having the bounds

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

In the `OPF` format the example is displayed as shown in Listing 16.1.

Listing 16.1: Example of an OPF file for a linear problem.

```
[comment]
  The lo1 example in OPF format
[/comment]

[hints]
  [hint NUMVAR] 4 [/hint]
  [hint NUMCON] 3 [/hint]
  [hint NUMANZ] 9 [/hint]
[/hints]
```

(continues on next page)

(continued from previous page)

```
[variables disallow_new_variables]
  x1 x2 x3 x4
[/variables]

[objective maximize 'obj']
   3 x1 + x2 + 5 x3 + x4
[/objective]

[constraints]
  [con 'c1'] 3 x1 +   x2 + 2 x3         = 30 [/con]
  [con 'c2'] 2 x1 +   x2 + 3 x3 +   x4 >= 15 [/con]
  [con 'c3']        2 x2        + 3 x4 <= 25 [/con]
[/constraints]

[bounds]
  [b] 0 <= * [/b]
  [b] 0 <= x2 <= 10 [/b]
[/bounds]
```

Quadratic Example `qo1.opf`

An example of a quadratic optimization problem is

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

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]
```

(continues on next page)

(continued from previous page)

```
[bounds]
  [b] 0 <= * [/b]
[/bounds]
```

Conic Quadratic Example `cqo1.opf`

Consider the example:

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

Please note that the type of the cones is defined by the parameter to `[cone ...]`; the content of the `cone`-section is the names of variables that belong to the cone. The resulting OPF file is in Listing 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: x4 >= sqrt( x1^2 + x2^2 )
  [cone quad 'k1'] x4, x1, x2 [/cone]

  # Define rotated quadratic cone: 2 x5 x6 >= x3^2
  [cone rquad 'k2'] x5, x6, x3 [/cone]
[/bounds]
```

Mixed Integer Example `milo1.opf`

Consider the mixed integer problem:

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

This can be implemented in `OPF` with the file in Listing 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]

[variables disallow_new_variables]
  x1 x2
[/variables]

[objective maximize 'obj']
   x1 + 6.4e-1 x2
[/objective]

[constraints]
  [con 'c1'] 5e+1 x1 + 3.1e+1 x2 <= 2.5e+2 [/con]
  [con 'c2'] -4 <= 3 x1 - 2 x2 [/con]
[/constraints]

[bounds]
  [b] 0 <= * [/b]
[/bounds]

[integer]
  x1 x2
[/integer]
```

16.4 The CBF Format

This document constitutes the technical reference manual of the *Conic Benchmark Format* with file extension: `.cbf` or `.CBF`. It unifies linear, second-order cone (also known as conic quadratic), exponential cone, power cone and semidefinite optimization with mixed-integer variables. The format has been designed with benchmark libraries in mind, and therefore focuses on compact and easily parsable representations. The CBF format separates problem structure from the problem data.

16.4.1 How Instances Are Specified

This section defines the spectrum of conic optimization problems that can be formulated in terms of the keywords of the CBF format.

In the CBF format, conic optimization problems are considered in the following form:

$$\begin{array}{lll} \min / \max & g^{obj} & \\ & g_i \in \mathcal{K}_i, & i \in \mathcal{I}, \\ \text{s.t.} & G_i \in \mathcal{K}_i, & i \in \mathcal{I}^{PSD}, \\ & x_j \in \mathcal{K}_j, & j \in \mathcal{J}, \\ & \overline{X}_j \in \mathcal{K}_j, & j \in \mathcal{J}^{PSD}. \end{array} \tag{16.9}$$

- **Variables** are either scalar variables, x_j for $j \in \mathcal{J}$, or matrix variables, $\overline{X}_j$ for $j \in \mathcal{J}^{PSD}$. Scalar variables can also be declared as integer.
- **Constraints** are affine expressions of the variables, either scalar-valued g_i for $i \in \mathcal{I}$, or matrix-valued G_i for $i \in \mathcal{I}^{PSD}$

$$g_i = \sum_{j \in \mathcal{J}^{PSD}} \langle F_{ij}, X_j \rangle + \sum_{j \in \mathcal{J}} a_{ij} x_j + b_i,$$

$$G_i = \sum_{j \in \mathcal{J}} x_j H_{ij} + D_i.$$

- The **objective function** is a scalar-valued affine expression of the variables, either to be minimized or maximized. We refer to this expression as g^{obj}

$$g^{obj} = \sum_{j \in \mathcal{J}^{PSD}} \langle F_j^{obj}, X_j \rangle + \sum_{j \in \mathcal{J}} a_j^{obj} x_j + b^{obj}.$$

As of version 4 of the format, CBF files can represent the following non-parametric cones $\mathcal{K}$:

- **Free domain** - A cone in the linear family defined by

$$\{x \in \mathbb{R}^n\}, \text{ for } n \geq 1.$$

- **Positive orthant** - A cone in the linear family defined by

$$\{x \in \mathbb{R}^n \mid x_j \geq 0 \text{ for } j = 1, \ldots, n\}, \text{ for } n \geq 1.$$

- **Negative orthant** - A cone in the linear family defined by

$$\{x \in \mathbb{R}^n \mid x_j \leq 0 \text{ for } j = 1, \ldots, n\}, \text{ for } n \geq 1.$$

- **Fixpoint zero** - A cone in the linear family defined by

$$\{x \in \mathbb{R}^n \mid x_j = 0 \text{ for } j = 1, \ldots, n\}, \text{ for } n \geq 1.$$

- **Quadratic cone** - A cone in the second-order cone family defined by

$$\left\{ \begin{pmatrix} p \\ x \end{pmatrix} \in \mathbb{R} \times \mathbb{R}^{n-1},\ p^2 \geq x^T x,\ p \geq 0 \right\},\ \text{for } n \geq 2.$$

- **Rotated quadratic cone** - A cone in the second-order cone family defined by

$$\left\{ \begin{pmatrix} p \\ q \\ x \end{pmatrix} \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}^{n-2},\ 2pq \geq x^T x,\ p \geq 0,\ q \geq 0 \right\},\ \text{for } n \geq 3.$$

- **Exponential cone** - A cone in the exponential cone family defined by

$$\mathrm{cl}(S_1) = S_1 \cup S_2$$

where,

$$S_1 = \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3,\ t \geq se^{\frac{r}{s}},\ s \geq 0 \right\}.$$

and,

$$S_2 = \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3,\ t \geq 0,\ r \leq 0,\ s = 0 \right\}.$$

- **Dual Exponential cone** - A cone in the exponential cone family defined by

$$\mathrm{cl}(S_1) = S_1 \cup S_2$$

where,

$$S_1 = \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3,\ et \geq (-r)e^{\frac{s}{r}},\ -r \geq 0 \right\}.$$

and,

$$S_2 = \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3,\ et \geq 0,\ s \geq 0,\ r = 0 \right\}.$$

- **Radial geometric mean cone** - A cone in the power cone family defined by

$$\left\{ \begin{pmatrix} p \\ x \end{pmatrix} \in \mathbb{R}^k_+ \times \mathbb{R}^1,\ \left(\prod_{j=1}^{k} p_j \right)^{\frac{1}{k}} \geq |x| \right\},\ \text{for } n = k+1 \geq 2.$$

- **Dual radial geometric mean cone** - A cone in the power cone family defined by

$$\left\{ \begin{pmatrix} p \\ x \end{pmatrix} \in \mathbb{R}^k_+ \times \mathbb{R}^1,\ \left(\prod_{j=1}^{k} kp_j \right)^{\frac{1}{k}} \geq |x| \right\},\ \text{for } n = k+1 \geq 2.$$

and, the following parametric cones:

- **Radial power cone** - A cone in the power cone family defined by

$$\left\{ \begin{pmatrix} p \\ x \end{pmatrix} \in \mathbb{R}^k_+ \times \mathbb{R}^{n-k}, \ \left(\prod_{j=1}^{k} p_j^{\alpha_j} \right)^{\frac{1}{\sigma}} \geq \|x\|_2 \right\}, \text{ for } n \geq k \geq 1.$$

  where, $\sigma = \sum_{j=1}^{k} \alpha_j$ and $\alpha = \mathbb{R}^k_{++}$.

- **Dual radial power cone** - A cone in the power cone family defined by

$$\left\{ \begin{pmatrix} p \\ x \end{pmatrix} \in \mathbb{R}^k_+ \times \mathbb{R}^{n-k}, \ \left(\prod_{j=1}^{k} \left(\frac{\sigma p_j}{\alpha_j} \right)^{\alpha_j} \right)^{\frac{1}{\sigma}} \geq \|x\|_2 \right\}, \text{ for } n \geq k \geq 1.$$

  where, $\sigma = \sum_{j=1}^{k} \alpha_j$ and $\alpha = \mathbb{R}^k_{++}$.

16.4.2 The Structure of CBF Files

This section defines how information is written in the CBF format, without being specific about the type of information being communicated.

All information items belong to exactly one of the three groups of information. These information groups, and the order they must appear in, are:

1. File format.
2. Problem structure.
3. Problem data.

The first group, file format, provides information on how to interpret the file. The second group, problem structure, provides the information needed to deduce the type and size of the problem instance. Finally, the third group, problem data, specifies the coefficients and constants of the problem instance.

Information items

The format is composed as a list of information items. The first line of an information item is the `KEYWORD`, revealing the type of information provided. The second line - of some keywords only - is the `HEADER`, typically revealing the size of information that follows. The remaining lines are the `BODY` holding the actual information to be specified.

```
KEYWORD
BODY

KEYWORD
HEADER
BODY
```

The `KEYWORD` determines how each line in the `HEADER` and `BODY` is structured. Moreover, the number of lines in the `BODY` follows either from the `KEYWORD`, the `HEADER`, or from another information item required to precede it.

File encoding and line width restrictions

The format is based on the US-ASCII printable character set with two extensions as listed below. Note, by definition, that none of these extensions can be misinterpreted as printable US-ASCII characters:

- A line feed marks the end of a line, carriage returns are ignored.
- Comment-lines may contain unicode characters in UTF-8 encoding.

The line width is restricted to 512 bytes, with 3 bytes reserved for the potential carriage return, line feed and null-terminator.

Integers and floating point numbers must follow the ISO C decimal string representation in the standard C locale. The format does not impose restrictions on the magnitude of, or number of significant digits in numeric data, but the use of 64-bit integers and 64-bit IEEE 754 floating point numbers should be sufficient to avoid loss of precision.

Comment-line and whitespace rules

The format allows single-line comments respecting the following rule:

- Lines having first byte equal to '#' (US-ASCII 35) are comments, and should be ignored. Comments are only allowed between information items.

Given that a line is not a comment-line, whitespace characters should be handled according to the following rules:

- Leading and trailing whitespace characters should be ignored.
  - The seperator between multiple pieces of information on one line, is either one or more whitespace characters.
- Lines containing only whitespace characters are empty, and should be ignored. Empty lines are only allowed between information items.

16.4.3 Problem Specification

The problem structure

The problem structure defines the objective sense, whether it is minimization and maximization. It also defines the index sets, $\mathcal{J}$, $\mathcal{J}^{PSD}$, $\mathcal{I}$ and $\mathcal{I}^{PSD}$, which are all numbered from zero, $\{0, 1, \ldots\}$, and empty until explicitly constructed.

- **Scalar variables** are constructed in vectors restricted to a conic domain, such as $(x_0, x_1) \in \mathbb{R}^2_+$, $(x_2, x_3, x_4) \in \mathcal{Q}^3$, etc. In terms of the Cartesian product, this generalizes to

$$x \in \mathcal{K}_1^{n_1} \times \mathcal{K}_2^{n_2} \times \cdots \times \mathcal{K}_k^{n_k}$$

  which in the CBF format becomes:

```
VAR
n k
K1 n1
K2 n2
...
Kk nk
```

  where $\sum_i n_i = n$ is the total number of scalar variables. The list of supported cones is found in Table 16.3. Integrality of scalar variables can be specified afterwards.

- **PSD variables** are constructed one-by-one. That is, $X_j \succeq \mathbf{0}^{n_j \times n_j}$ for $j \in \mathcal{J}^{PSD}$, constructs a matrix-valued variable of size $n_j \times n_j$ restricted to be symmetric positive semidefinite. In the CBF format, this list of constructions becomes:

```
PSDVAR
N
n1
n2
...
nN
```

where N is the total number of PSD variables.

- **Scalar constraints** are constructed in vectors restricted to a conic domain, such as $(g_0, g_1) \in \mathbb{R}^2_+$, $(g_2, g_3, g_4) \in \mathcal{Q}^3$, etc. In terms of the Cartesian product, this generalizes to

$$g \in \mathcal{K}_1^{m_1} \times \mathcal{K}_2^{m_2} \times \cdots \times \mathcal{K}_k^{m_k}$$

which in the CBF format becomes:

```
CON
m k
K1 m1
K2 m2
..
Kk mk
```

where $\sum_i m_i = m$ is the total number of scalar constraints. The list of supported cones is found in Table 16.3.

- **PSD constraints** are constructed one-by-one. That is, $G_i \succeq \mathbf{0}^{m_i \times m_i}$ for $i \in \mathcal{I}^{PSD}$, constructs a matrix-valued affine expressions of size $m_i \times m_i$ restricted to be symmetric positive semidefinite. In the CBF format, this list of constructions becomes

```
PSDCON
M
m1
m2
..
mM
```

where M is the total number of PSD constraints.

With the objective sense, variables (with integer indications) and constraints, the definitions of the many affine expressions follow in problem data.

Problem data

The problem data defines the coefficients and constants of the affine expressions of the problem instance. These are considered zero until explicitly defined, implying that instances with no keywords from this information group are, in fact, valid. Duplicating or conflicting information is a failure to comply with the standard. Consequently, two coefficients written to the same position in a matrix (or to transposed positions in a symmetric matrix) is an error.

The affine expressions of the objective, g^{obj}, of the scalar constraints, g_i, and of the PSD constraints, G_i, are defined separately. The following notation uses the standard trace inner product for matrices, $\langle X, Y \rangle = \sum_{i,j} X_{ij} Y_{ij}$.

- The affine expression of the objective is defined as

$$g^{obj} = \sum_{j \in \mathcal{J}^{PSD}} \langle F_j^{obj}, X_j \rangle + \sum_{j \in \mathcal{J}} a_j^{obj} x_j + b^{obj},$$

in terms of the symmetric matrices, F_j^{obj}, and scalars, a_j^{obj} and b^{obj}.

- The affine expressions of the scalar constraints are defined, for $i \in \mathcal{I}$, as

$$g_i = \sum_{j \in \mathcal{J}^{PSD}} \langle F_{ij}, X_j \rangle + \sum_{j \in \mathcal{J}} a_{ij} x_j + b_i,$$

  in terms of the symmetric matrices, F_{ij}, and scalars, a_{ij} and b_i.

- The affine expressions of the PSD constraints are defined, for $i \in \mathcal{I}^{PSD}$, as

$$G_i = \sum_{j \in \mathcal{J}} x_j H_{ij} + D_i,$$

  in terms of the symmetric matrices, H_{ij} and D_i.

List of cones

The format uses an explicit syntax for symmetric positive semidefinite cones as shown above. For scalar variables and constraints, constructed in vectors, the supported conic domains and their sizes are given as follows.

Table 16.3: Cones available in the CBF format

Name	CBF keyword	Cone family	Cone size
Free domain	`F`	linear	$n \geq 1$
Positive orthant	`L+`	linear	$n \geq 1$
Negative orthant	`L-`	linear	$n \geq 1$
Fixpoint zero	`L=`	linear	$n \geq 1$
Quadratic cone	`Q`	second-order	$n \geq 1$
Rotated quadratic cone	`QR`	second-order	$n \geq 2$
Exponential cone	`EXP`	exponential	$n = 3$
Dual exponential cone	`EXP*`	exponential	$n = 3$
Radial geometric mean cone	`GMEANABS`	power	$n = k + 1 \geq 2$
Dual radial geometric mean cone	`GMEANABS*`	power	$n = k + 1 \geq 2$
Radial power cone (parametric)	`POW`	power	$n \geq k \geq 1$
Dual radial power cone (parametric)	`POW*`	power	$n \geq k \geq 1$

16.4.4 File Format Keywords

VER

Description: The version of the Conic Benchmark Format used to write the file.
`HEADER`: None
`BODY`: One line formatted as:

```
INT
```

This is the version number.
Must appear exactly once in a file, as the first keyword.

POWCONES

Description: Define a lookup table for power cone domains.
`HEADER`: One line formatted as:

```
INT INT
```

This is the number of cones to be specified and the combined length of their dense parameter vectors.

`BODY`: A list of chunks each specifying the dense parameter vector of a power cone.
`CHUNKHEADER`: One line formatted as:

```
INT
```

This is the parameter vector length.

`CHUNKBODY`: A list of lines formatted as:

```
REAL
```

This is the parameter vector values. The number of lines should match the number stated in the chunk header.

The cone specified at index k (with 0-based indexing) is registered under the CBF name @k:POW.

POW*CONES

Description: Define a lookup table for dual power cone domains.
`HEADER`: One line formatted as:

```
INT INT
```

This is the number of cones to be specified and the combined length of their dense parameter vectors.

`BODY`: A list of chunks each specifying the dense parameter vector of a dual power cone.
`CHUNKHEADER`: One line formatted as:

```
INT
```

This is the parameter vector length.

`CHUNKBODY`: A list of lines formatted as:

```
REAL
```

This is the parameter vector values. The number of lines should match the number stated in the chunk header.

The cone specified at index k (with 0-based indexing) is registered under the CBF name @k:POW*.

OBJSENSE

Description: Define the objective sense.
`HEADER`: None
`BODY`: One line formatted as:

```
STR
```

having `MIN` indicates minimize, and `MAX` indicates maximize. Upper-case letters are required. Must appear exactly once in a file.

PSDVAR

Description: Construct the PSD variables.
`HEADER`: One line formatted as:

```
INT
```

This is the number of PSD variables in the problem.
`BODY`: A list of lines formatted as:

```
INT
```

This indicates the number of rows (equal to the number of columns) in the matrix-valued PSD variable. The number of lines should match the number stated in the header.

VAR

Description: Construct the scalar variables.

`HEADER`: One line formatted as:

```
INT INT
```

This is the number of scalar variables, followed by the number of conic domains they are restricted to.

`BODY`: A list of lines formatted as:

```
STR INT
```

This indicates the cone name (see Table 16.3), and the number of scalar variables restricted to this cone. These numbers should add up to the number of scalar variables stated first in the header. The number of lines should match the second number stated in the header.

INT

Description: Declare integer requirements on a selected subset of scalar variables.

`HEADER`: one line formatted as:

```
INT
```

This is the number of integer scalar variables in the problem.

`BODY`: a list of lines formatted as:

```
INT
```

This indicates the scalar variable index $j \in \mathcal{J}$. The number of lines should match the number stated in the header.

Can only be used after the keyword `VAR`.

PSDCON

Description: Construct the PSD constraints.

`HEADER`: One line formatted as:

```
INT
```

This is the number of PSD constraints in the problem.

`BODY`: A list of lines formatted as:

```
INT
```

This indicates the number of rows (equal to the number of columns) in the matrix-valued affine expression of the PSD constraint. The number of lines should match the number stated in the header.

Can only be used after these keywords: `PSDVAR`, `VAR`.

CON

Description: Construct the scalar constraints.

`HEADER`: One line formatted as:

```
INT INT
```

This is the number of scalar constraints, followed by the number of conic domains they restrict to.

`BODY`: A list of lines formatted as:

```
STR INT
```

This indicates the cone name (see Table 16.3), and the number of affine expressions restricted to this cone. These numbers should add up to the number of scalar constraints stated first in the header. The number of lines should match the second number stated in the header.

Can only be used after these keywords: `PSDVAR`, `VAR`

OBJFCOORD

Description: Input sparse coordinates (quadruplets) to define the symmetric matrices F_j^{obj}, as used in the objective.

`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.

`BODY`: A list of lines formatted as:

```
INT INT INT REAL
```

This indicates the PSD variable index $j \in \mathcal{J}^{PSD}$, the row index, the column index and the coefficient value. The number of lines should match the number stated in the header.

OBJACOORD

Description: Input sparse coordinates (pairs) to define the scalars, a_j^{obj}, as used in the objective.

`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.

`BODY`: A list of lines formatted as:

```
INT REAL
```

This indicates the scalar variable index $j \in \mathcal{J}$ and the coefficient value. The number of lines should match the number stated in the header.

OBJBCOORD

Description: Input the scalar, b^{obj}, as used in the objective.

`HEADER`: None.

`BODY`: One line formatted as:

```
REAL
```

This indicates the coefficient value.

FCOORD

Description: Input sparse coordinates (quintuplets) to define the symmetric matrices, F_{ij}, as used in the scalar constraints.

`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.

`BODY`: A list of lines formatted as:

```
INT INT INT INT REAL
```

This indicates the scalar constraint index $i \in \mathcal{I}$, the PSD variable index $j \in \mathcal{J}^{PSD}$, the row index, the column index and the coefficient value. The number of lines should match the number stated in the header.

ACOORD

Description: Input sparse coordinates (triplets) to define the scalars, a_{ij}, as used in the scalar constraints.
`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.
`BODY`: A list of lines formatted as:

```
INT INT REAL
```

This indicates the scalar constraint index $i \in \mathcal{I}$, the scalar variable index $j \in \mathcal{J}$ and the coefficient value. The number of lines should match the number stated in the header.

BCOORD

Description: Input sparse coordinates (pairs) to define the scalars, b_i, as used in the scalar constraints.
`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.
`BODY`: A list of lines formatted as:

```
INT REAL
```

This indicates the scalar constraint index $i \in \mathcal{I}$ and the coefficient value. The number of lines should match the number stated in the header.

HCOORD

Description: Input sparse coordinates (quintuplets) to define the symmetric matrices, H_{ij}, as used in the PSD constraints.
`HEADER`: One line formatted as:

```
INT
```

This is the number of coordinates to be specified.
`BODY`: A list of lines formatted as

```
INT INT INT INT REAL
```

This indicates the PSD constraint index $i \in \mathcal{I}^{PSD}$, the scalar variable index $j \in \mathcal{J}$, the row index, the column index and the coefficient value. The number of lines should match the number stated in the header.

DCOORD

Description: Input sparse coordinates (quadruplets) to define the symmetric matrices, D_i, as used in the PSD constraints.
`HEADER`: One line formatted as

```
INT
```

This is the number of coordinates to be specified.
`BODY`: A list of lines formatted as:

```
INT INT INT REAL
```

This indicates the PSD constraint index $i \in \mathcal{I}^{PSD}$, the row index, the column index and the coefficient value. The number of lines should match the number stated in the header.

16.4.5 CBF Format Examples

Minimal Working Example

The conic optimization problem (16.10) , has three variables in a quadratic cone - first one is integer - and an affine expression in domain 0 (equality constraint).

$$\begin{array}{ll} \text{minimize} & 5.1\,x_0 \\ \text{subject to} & 6.2\,x_1 + 7.3\,x_2 - 8.4 \in \{0\} \\ & x \in \mathcal{Q}^3,\ x_0 \in \mathbb{Z}. \end{array} \tag{16.10}$$

Its formulation in the Conic Benchmark Format begins with the version of the CBF format used, to safeguard against later revisions.

```
VER
4
```

Next follows the problem structure, consisting of the objective sense, the number and domain of variables, the indices of integer variables, and the number and domain of scalar-valued affine expressions (i.e., the equality constraint).

```
OBJSENSE
MIN

VAR
3 1
Q 3

INT
1
0

CON
1 1
L= 1
```

Finally follows the problem data, consisting of the coefficients of the objective, the coefficients of the constraints, and the constant terms of the constraints. All data is specified on a sparse coordinate form.

```
OBJACOORD
1
0 5.1

ACOORD
2
0 1 6.2
0 2 7.3

BCOORD
1
0 -8.4
```

This concludes the example.

Mixing Linear, Second-order and Semidefinite Cones

The conic optimization problem (16.11), has a semidefinite cone, a quadratic cone over unordered subindices, and two equality constraints.

$$\begin{array}{lll} \text{minimize} & \left\langle \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, X_1 \right\rangle + x_1 & \\ \text{subject to} & \left\langle \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, X_1 \right\rangle + x_1 & = \quad 1.0\,, \\ & \left\langle \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, X_1 \right\rangle + x_0 + x_2 & = \quad 0.5\,, \\ & x_1 \geq \sqrt{x_0^2 + x_2^2}\,, & \\ & X_1 \succeq \mathbf{0}\,. & \end{array} \tag{16.11}$$

The equality constraints are easily rewritten to the conic form, $(g_0, g_1) \in \{0\}^2$, by moving constants such that the right-hand-side becomes zero. The quadratic cone does not fit under the `VAR` keyword in this variable permutation. Instead, it takes a scalar constraint $(g_2, g_3, g_4) = (x_1, x_0, x_2) \in \mathcal{Q}^3$, with scalar variables constructed as $(x_0, x_1, x_2) \in \mathbb{R}^3$. Its formulation in the CBF format is reported in the following list

```
# File written using this version of the Conic Benchmark Format:
#     | Version 4.
VER
4

# The sense of the objective is:
#     | Minimize.
OBJSENSE
MIN

# One PSD variable of this size:
#     | Three times three.
PSDVAR
1
3

# Three scalar variables in this one conic domain:
#     | Three are free.
VAR
3 1
F 3

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

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

(continues on next page)

(continued from previous page)

```
0 0 0 2.0
0 1 0 1.0
0 1 1 2.0
0 2 1 1.0
0 2 2 2.0

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

# Nine coordinates in F_ij coefficients:
#     | F[0,0][0,0] = 1.0
#     | F[0,0][1,1] = 1.0
#     | and more...
FCOORD
9
0 0 0 0 1.0
0 0 1 1 1.0
0 0 2 2 1.0
1 0 0 0 1.0
1 0 1 0 1.0
1 0 2 0 1.0
1 0 1 1 1.0
1 0 2 1 1.0
1 0 2 2 1.0

# Six coordinates in a_ij coefficients:
#     | a[0,1] = 1.0
#     | a[1,0] = 1.0
#     | and more...
ACOORD
6
0 1 1.0
1 0 1.0
1 2 1.0
2 1 1.0
3 0 1.0
4 2 1.0

# Two coordinates in b_i coefficients:
#     | b[0] = -1.0
#     | b[1] = -0.5
BCOORD
2
0 -1.0
1 -0.5
```

Mixing Semidefinite Variables and Linear Matrix Inequalities

The standard forms in semidefinite optimization are usually based either on semidefinite variables or linear matrix inequalities. In the CBF format, both forms are supported and can even be mixed as shown.

$$\begin{array}{lll} \text{minimize} & \left\langle \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, X_1 \right\rangle + x_1 + x_2 + 1 & \\ \text{subject to} & \left\langle \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, X_1 \right\rangle - x_1 - x_2 & \geq \quad 0.0\,, \\ & x_1 \begin{bmatrix} 0 & 1 \\ 1 & 3 \end{bmatrix} + x_2 \begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \succeq \quad \mathbf{0}\,, \\ & X_1 \succeq \mathbf{0}\,. & \end{array} \tag{16.12}$$

Its formulation in the CBF format is written in what follows

```
# File written using this version of the Conic Benchmark Format:
#     | Version 4.
VER
4

# The sense of the objective is:
#     | Minimize.
OBJSENSE
MIN

# One PSD variable of this size:
#     | Two times two.
PSDVAR
1
2

# Two scalar variables in this one conic domain:
#     | Two are free.
VAR
2 1
F 2

# One PSD constraint of this size:
#     | Two times two.
PSDCON
1
2

# One scalar constraint with an affine expression in this one conic domain:
#     | One is greater than or equal to zero.
CON
1 1
L+ 1

# Two coordinates in F^{obj}_j coefficients:
#     | F^{obj}[0][0,0] = 1.0
#     | F^{obj}[0][1,1] = 1.0
OBJFCOORD
2
0 0 0 1.0
0 1 1 1.0

# Two coordinates in a^{obj}_j coefficients:
```

(continues on next page)

(continued from previous page)

```
#     | a^{obj}[0] = 1.0
#     | a^{obj}[1] = 1.0
OBJACOORD
2
0 1.0
1 1.0

# One coordinate in b^{obj} coefficient:
#     | b^{obj} = 1.0
OBJBCOORD
1.0

# One coordinate in F_ij coefficients:
#     | F[0,0][1,0] = 1.0
FCOORD
1
0 0 1 0 1.0

# Two coordinates in a_ij coefficients:
#     | a[0,0] = -1.0
#     | a[0,1] = -1.0
ACOORD
2
0 0 -1.0
0 1 -1.0

# Four coordinates in H_ij coefficients:
#     | H[0,0][1,0] = 1.0
#     | H[0,0][1,1] = 3.0
#     | and more...
HCOORD
4
0 0 1 0 1.0
0 0 1 1 3.0
0 1 0 0 3.0
0 1 1 0 1.0

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

The exponential cone

The conic optimization problem (16.13), has one equality constraint, one quadratic cone constraint and an exponential cone constraint.

$$\begin{array}{ll} \text{minimize} & x_0 - x_3 \\ \text{subject to} & x_0 + 2\,x_1 - x_2 \in \{0\} \\ & (5.0, x_0, x_1) \in \mathcal{Q}^3 \\ & (x_2, 1.0, x_3) \in EXP. \end{array} \tag{16.13}$$

The nonlinear conic constraints enforce $\sqrt{x_0^2 + x_1^2} \leq 0.5$ and $x_3 \leq \log(x_2)$.

```
# File written using this version of the Conic Benchmark Format:
#      | Version 3.
VER
3

# The sense of the objective is:
#      | Minimize.
OBJSENSE
MIN

# Four scalar variables in this one conic domain:
#      | Four are free.
VAR
4 1
F 4

# Seven scalar constraints with affine expressions in three conic domains:
#      | One is fixed to zero.
#      | Three are in conic quadratic domain.
#      | Three are in exponential cone domain.
CON
7 3
L= 1
Q 3
EXP 3

# Two coordinates in a^{obj}_j coefficients:
#      | a^{obj}[0] = 1.0
#      | a^{obj}[3] = -1.0
OBJACOORD
2
0 1.0
3 -1.0

# Seven coordinates in a_ij coefficients:
#      | a[0,0] = 1.0
#      | a[0,1] = 2.0
#      | and more...
ACOORD
7
0 0 1.0
0 1 2.0
0 2 -1.0
2 0 1.0
3 1 1.0
4 2 1.0
6 3 1.0
```

(continues on next page)

(continued from previous page)

```
# Two coordinates in b_i coefficients:
#      | b[1] = 5.0
#      | b[5] = 1.0
BCOORD
2
1 5.0
5 1.0
```

Parametric cones

The problem (16.14), has three variables in a power cone with parameter $\alpha_1 = (1, 1)$ and two power cone constraints each with parameter $\alpha_0 = (8, 1)$.

$$\begin{array}{ll} \text{minimize} & x_3 \\ \text{subject to} & (1.0, x_1, x_1 + x_2) \in POW_{\alpha_0} \\ & (1.0, x_2, x_1 + x_2) \in POW_{\alpha_0} \\ & x \in POW_{\alpha_1}. \end{array} \tag{16.14}$$

The nonlinear conic constraints enforce $x_3 \leq x_1 x_2$ and $x_1 + x_2 \leq \min(x_1^{\frac{1}{9}}, x_2^{\frac{1}{9}})$.

```
# File written using this version of the Conic Benchmark Format:
#      | Version 3.
VER
3

# Two power cone domains defined in a total of four parameters:
#      | @0:POW (specification 0) has two parameters:
#      | alpha[0] = 8.0.
#      | alpha[1] = 1.0.
#      | @1:POW (specification 1) has two parameters:
#      | alpha[0] = 1.0.
#      | alpha[1] = 1.0.
POWCONES
2 4
2
8.0
1.0
2
1.0
1.0

# The sense of the objective is:
#      | Maximize.
OBJSENSE
MAX

# Three scalar variable in this one conic domain:
#      | Three are in power cone domain (specification 1).
VAR
3 1
@1:POW 3

# Six scalar constraints with affine expressions in two conic domains:
#      | Three are in power cone domain (specification 0).
#      | Three are in power cone domain (specification 0).
```

(continues on next page)

(continued from previous page)

```
CON
6 2
@0:POW 3
@0:POW 3

# One coordinate in a^{obj}_j coefficients:
#      | a^{obj}[2] = 1.0
OBJACOORD
1
2 1.0

# Six coordinates in a_ij coefficients:
#      | a[1,0] = 1.0
#      | a[2,0] = 1.0
#      | and more...
ACOORD
6
1 0 1.0
2 0 1.0
2 1 1.0
4 1 1.0
5 0 1.0
5 1 1.0

# Two coordinates in b_i coefficients:
#      | b[0] = 1.0
#      | b[3] = 1.0
BCOORD
2
0 1.0
3 1.0
```

16.5 The PTF Format

The PTF format is a human-readable, natural text format that supports all linear, conic and mixed-integer features.

16.5.1 The overall format

The format is indentation based, where each section is started by a head line and followed by a section body with deeper indentation that the head line. For example:

```
Header line
   Body line 1
   Body line 1
   Body line 1
```

Section can also be nested:

```
Header line A
   Body line in A
   Header line A.1
      Body line in A.1
      Body line in A.1
   Body line in A
```

The indentation of blank lines is ignored, so a subsection can contain a blank line with no indentation. The character # defines a line comment and anything between the # character and the end of the line is ignored.

In a PTF file, the first section must be a `Task` section. The order of the remaining section is arbitrary, and sections may occur multiple times or not at all.

MOSEK will ignore any top-level section it does not recognize.

Names

In the description of the format we use following definitions for name strings:

```
NAME: PLAIN_NAME | QUOTED_NAME
PLAIN_NAME: [a-zA-Z_] [a-zA-Z0-9_-.!|]
QUOTED_NAME: "'" ( [^'\\\r\n] | "\\" ( [\\rn] | "x" [0-9a-fA-F] [0-9a-fA-F] ) )* "'"
```

Expressions

An expression is a sum of terms. A term is either a linear term (a coefficient and a variable name, where the coefficient can be left out if it is 1.0), or a matrix inner product.

An expression:

```
EXPR: EMPTY | ( [+-] TERM )*
TERM: LINEAR_TERM | MATRIX_TERM
```

A linear term

```
LINEAR_TERM: FLOAT? NAME
```

A matrix term

```
MATRIX_TERM: "<" ( [+-] FLOAT? NAME)* ";" NAME ">"
```

Here the right-hand name is the name of a (semidefinite) matrix variable, and the left-hand side is a sum of symmetric matrices. The actual matrices are defined in a separate section.

Expressions can span multiple lines by giving subsequent lines a deeper indentation.

For example following two section are equivalent:

```
# Everything on one line:
+ x1 + x2 + x3 + x4

# Split into multiple lines:
+ x1
  + x2
  + x3
  + x4
```

16.5.2 `Task` section

The first section of the file must be a `Task`. The text in this section is not used and may contain comments, or meta-information from the writer or about the content.

Format:

```
Task NAME
   Anything goes here...
```

`NAME` is a the task name.

16.5.3 `Objective` section

The `Objective` section defines the objective name, sense and function. The format:

```
"Objective" NAME?
   ( "Minimize" | "Maximize" ) EXPR
```

For example:

```
Objective 'obj'
   Minimize + x1 + 0.2 x2 + < M1 ; X1 >
```

16.5.4 `Constraints` section

The constraints section defines a series of constraints. A constraint defines a term $A \cdot x + b \in K$. For linear constraints `A` is just one row, while for conic constraints it can be multiple rows. If a constraint spans multiple rows these can either be written inline separated by semi-colons, or each expression in a separete sub-section.

Simple linear constraints:

```
"Constraints"
  NAME? "[" [-+] (FLOAT | "inf") (";" [-+] (FLOAT | "inf") )? "]" EXPR
```

If the brackets contain two values, they are used as upper and lower bounds. It they contain one value the constraint is an equality.

For example:

```
Constraints
  # Ranged constraint
  'c1' [0;10] + x1 + x2 + x3
  # Fixed constraint, expression equals to 0
  [0] + x1 + x2 + x3
  # Nonnegative constraint
  [0;+inf] + x1 + x2 + x3
```

Constraint blocks put the expression either in a subsection or inline. The cone type (domain) is written in the brackets, and **MOSEK** currently supports following types:

- **Major (primal) cones:**
  - `QUAD(N)` or `SOC(N)`: Second order cone of dimension `N`.
  - `RQUAD(N)` or `RSOC(N)`: Rotated second order cone of dimension `N`.
  - `PEXP`: Primal exponential cone of dimension 3.
  - `PPOW(N,P)`: Primal power cone of dimension `N` with parameter `P` (float between 0 and 1).
  - `PPOW(N;ALPHA)`: Primal power cone of dimension `N` with exponent sequence `ALPHA` (comma-separated list of floats).
  - `PGEOMEAN(N)`: Primal geometric mean cone of domension `N`.
  - `SVECPSD(N)`: Vectorized symmetric positive semidefinite cone of dimension `N` (`N` must be of the form `D*(D+1)/2`).
- **Dual cones:**
  - `DEXP`: Dual exponential cone of dimension 3.
  - `DPOW(N,P)`: Dual power cone of dimension `N` with parameter `P` (float between 0 and 1).
  - `DPOW(N;ALPHA)`: Dual power cone of dimension `N` with exponent sequence `ALPHA` (comma-separated list of floats).
  - `DGEOMEAN(N)`: Dual geometric mean cone of domension `N`.
- **Linear cones:**
  - `FREE(N)` The free (unbounded) cone of dimension `N`.
  - `POSITIVE(N)` The non-negative cone of dimension `N`.

- `NEGATIVE(N)` The non-positive cone of dimension `N`.
- `ZERO(N)` The zero-cone of dimension `N`.

See Sec. 15.8 for definitions of the parameters.

```
"Constraints"
  NAME? "[" DOMAIN "]" EXPR_LIST
```

For example:

```
Constraints
   'K1' [PPOW(5;3,1)]
      + x1 + x2
      + x2 + x3
      + 1.0
      + x1
      + x3
   'K2' [RQUAD(3)]
      + x1 + x2
      + x2 + x3
      + x3 + x1
```

16.5.5 `Variables` section

Any variable used in an expression must be defined in a variable section. The variable section defines each variable domain.

```
"Variables"
   NAME "[" [-+] (FLOAT | "inf") (";" [-+] (FLOAT | "inf") )? "]"
   NAME "[" "PSD" (INT) "]"
```

For example, a linear variable

```
Variables
   # Nonnegative variable
   x1 [0;inf]
   # Ranged variable
   x2 [0;1]
   # Fixed variable
   x3 [5.0]
   # 5-dimensional symmetric matrix variable
   X [PSD(5)]
```

16.5.6 `Integer` section

This section contains a list of variables that are integral. For example:

```
Integer
   x1 x2 x3
```

16.5.7 `SymmetricMatrixes` section

This section defines the symmetric matrixes used for matrix coefficients in matrix inner product terms. The section lists named matrixes, each with a size and a number of non-zeros. Only non-zeros in the lower triangular part should be defined.

```
"SymmetricMatrixes"
   NAME "SYMMAT" "(" INT ")"  ( "(" INT "," INT "," FLOAT ")" )*
   ...
```

For example:

```
SymmetricMatrixes
   M1 SYMMAT(3) (0,0,1.0) (1,1,2.0) (2,1,0.5)
   M2 SYMMAT(3)
      (0,0,1.0)
      (1,1,2.0)
      (2,1,0.5)
```

16.5.8 `Solutions` section

Each subsection defines a solution. A solution defines for each constraint and for each variable exactly one primal value and either one (for conic domains) or two (for linear domains) dual values. The values follow the same logic as in the **MOSEK** C API. A primal and a dual solution status defines the meaning of the values primal and dual (solution, certificate, unknown, etc.)

The format is this:

```
"Solutions"
   "Solution" WHICHSOL
      "ProblemStatus" PROSTA PROSTA?
   "SolutionStatus" SOLSTA SOLSTA?
   "Objective" FLOAT FLOAT_OR_NONE
   "Variables"
      # Linear variable status: level, slx, sux
      NAME "[" STATUS "]" FLOAT FLOAT_OR_NONE FLOAT_OR_NONE
   "Constraints"
      # Linear variable status: level, slx, sux
      NAME "[" STATUS "]" FLOAT FLOAT_OR_NONE FLOAT_OR_NONE
      # Conic constraint status: level, doty
      NAME
         "[" STATUS "]" FLOAT FLOAT_OR_NONE
```

Nonexistent values (for example, dual values for an integer solution) are replaced with a single dot (`.`):

```
FLOAT_OR_NONE = FLOAT | .
```

Following values for `WHICHSOL` are supported:

- `interior` Interior solution, the result of an interior-point solver.
- `basic` Basic solution, as produced by a simplex solver.
- `integer` Integer solution, the solution to a mixed-integer problem. This does not define a dual solution.

Following values for `PROSTA` are supported:

- `unknown` The problem status is unknown
- `feasible` The problem has been proven feasible
- `infeasible` The problem has been proven infeasible
- `illposed` The problem has been proved to be ill posed
- `infeasible_or_unbounded` The problem is infeasible or unbounded

Following values for `SOLSTA` are supported:

- `unknown` The solution status is unknown
- `feasible` The solution is feasible
- `optimal` The solution is optimal
- `infeas_cert` The solution is a certificate of infeasibility
- `illposed_cert` The solution is a certificate of illposedness

Following values for `STATUS` are supported:

- `unknown` The value is unknown
- `super_basic` The value is super basic
- `at_lower` The value is basic and at its lower bound
- `at_upper` The value is basic and at its upper bound
- `fixed` The value is basic fixed
- `infinite` The value is at infinity

16.5.9 Examples

Linear example `lo1.ptf`

```
Task ''
    # Written by MOSEK v10.0.13
    # problemtype: Linear Problem
    # number of linear variables: 4
    # number of linear constraints: 3
    # number of old-style A nonzeros: 9
Objective obj
    Maximize + 3 x1 + x2 + 5 x3 + x4
Constraints
    c1 [3e+1] + 3 x1 + x2 + 2 x3
    c2 [1.5e+1;+inf] + 2 x1 + x2 + 3 x3 + x4
    c3 [-inf;2.5e+1] + 2 x2 + 3 x4
Variables
    x1 [0;+inf]
    x2 [0;1e+1]
    x3 [0;+inf]
    x4 [0;+inf]
```

Conic quadratic example `cqo1.ptf`

```
Task ''
    # Written by MOSEK v10.0.17
    # problemtype: Conic Problem
    # number of linear variables: 6
    # number of linear constraints: 1
    # number of  old-style cones: 0
    # number of positive semidefinite variables: 0
    # number of positive semidefinite matrixes: 0
    # number of affine conic constraints: 2
    # number of disjunctive constraints: 0
    # number scalar affine expressions/nonzeros : 6/6
    # number of old-style A nonzeros: 3
Objective obj
    Minimize + x4 + x5 + x6
Constraints
    c1 [1] + x1 + x2 + 2 x3
    k1 [QUAD(3)]
        @ac1: + x4
        @ac2: + x1
        @ac3: + x2
    k2 [RQUAD(3)]
        @ac4: + x5
        @ac5: + x6
        @ac6: + x3
Variables
    x4
    x1 [0;+inf]
    x2 [0;+inf]
    x5
    x6
    x3 [0;+inf]
```

Power cone example `cqo1.ptf`

```
Task ''
Objective ''
    Maximize - x0 + x3 + x4
Constraints
    c0 [2] + x0 + x1 + 5e-1 x2
    C1 [PPOW(3,2e-1)]
        + x0
        + x1
        + x3
    C2 [PPOW(3;4.0,6.0)]
        + x2
        + x5
        + x4
Variables
    x0
    x1
    x2
    x3
    x4
    x5 [1.0]
```

Disjunctive example djc1.ptf

```
Task djc1
Objective ''
    Minimize + 2 'x[0]' + 'x[1]' + 3 'x[2]' + 'x[3]'
Constraints
    @c0 [-10;+inf] + 'x[0]' + 'x[1]' + 'x[2]' + 'x[3]'
    @D0 [OR]
        [AND]
            [NEGATIVE(1)]
                 + 'x[0]' - 2 'x[1]' + 1
            [ZERO(2)]
                 + 'x[2]'
                 + 'x[3]'
        [AND]
            [NEGATIVE(1)]
                 + 'x[2]' - 3 'x[3]' + 2
            [ZERO(2)]
                 + 'x[0]'
                 + 'x[1]'
    @D1 [OR]
        [ZERO(1)]
             + 'x[0]' - 2.5
        [ZERO(1)]
             + 'x[1]' - 2.5
        [ZERO(1)]
             + 'x[2]' - 2.5
        [ZERO(1)]
             + 'x[3]' - 2.5
Variables
    'x[0]'
    'x[1]'
    'x[2]'
    'x[3]'
```

Semidefinite example sdo1.ptf

```
Task ''
    # Written by MOSEK v10.0.17
    # problemtype: Conic Problem
    # number of linear variables: 3
    # number of linear constraints: 0
    # number of  old-style cones: 0
    # number of positive semidefinite variables: 1
    # number of positive semidefinite matrixes: 3
    # number of affine conic constraints: 2
    # number of disjunctive constraints: 0
    # number scalar affine expressions/nonzeros : 5/6
    # number of old-style A nonzeros: 0
Objective ''
    Minimize + @x0 + <M0;@X0>
Constraints
    @C0 [ZERO(2)]
        @ac0: + @x0 + < + M1;@X0> - 1
        @ac1: + @x1 + @x2 + < + M2;@X0> - 0.5
    @C1 [QUAD(3)]
```

(continues on next page)

(continued from previous page)

```
        @ac2: + @x0
        @ac3: + @x1
        @ac4: + @x2
Variables
    @x0
    @x1
    @x2
    @X0 [PSD(3)]
SymmetricMatrixes
    M0 SYMMAT(3) (0,0,2) (1,0,1) (1,1,2) (2,1,1) (2,2,2)
    M1 SYMMAT(3) (0,0,1) (1,1,1) (2,2,1)
    M2 SYMMAT(3) (0,0,1) (1,0,1) (1,1,1) (2,0,1) (2,1,1) (2,2,1)
```

16.6 The Task Format

The Task format is **MOSEK**'s native binary format. It contains a complete image of a **MOSEK** task, i.e.

- Problem data: Linear, conic, semidefinite and quadratic data
- Problem item names: Variable names, constraints names, cone names etc.
- Parameter settings
- Solutions

There are a few things to be aware of:

- Status of a solution read from a file will *always* be unknown.
- Parameter settings in a task file *always override* any parameters set on the command line or in a parameter file.

The format is based on the *TAR* (USTar) file format. This means that the individual pieces of data in a `.task` file can be examined by unpacking it as a *TAR* file. Please note that the inverse may not work: Creating a file using *TAR* will most probably not create a valid **MOSEK** Task file since the order of the entries is important.

16.7 The JSON Format

MOSEK provides the possibility to read/write problems and solutions in JSON format. The official JSON website http://www.json.org provides plenty of information along with the format definition. JSON is an industry standard for data exchange and JSON files can be easily written and read in most programming languages using dedicated libraries.

MOSEK uses two JSON-based formats:

- **JTASK**, for storing problem instances together with solutions and parameters. The JTASK format contains the same information as a native **MOSEK** task *task format*, that is a very close representation of the internal data storage in the task object.

  You can write a JTASK file specifying the extension `.jtask`. When the parameter *MSK_IPAR_WRITE_JSON_INDENTATION* is set the JTASK file will be indented to slightly improve readability.

- **JSOL**, for storing solutions and information items.

  You can write a JSOL solution file using *writejsonsol*. When the parameter *MSK_IPAR_WRITE_JSON_INDENTATION* is set the JSOL file will be indented to slightly improve readability.

  You can read a JSOL solution into an existing task file using *readjsonsol*. Only the `Task/solutions` section of the data will be taken into consideration.

16.7.1 JTASK Specification

The JTASK is a dictionary containing the following sections. All sections are optional and can be omitted if irrelevant for the problem.

- `$schema`: JSON schema.
- `Task/name`: The name of the task (string).
- `Task/INFO`: Information about problem data dimensions and similar. These are treated as hints when reading the file.
  - `numvar`: number of variables (int32).
  - `numcon`: number of constraints (int32).
  - `numcone`: number of cones (int32, deprecated).
  - `numbarvar`: number of symmetric matrix variables (int32).
  - `numanz`: number of nonzeros in A (int64).
  - `numsymmat`: number of matrices in the symmetric matrix storage E (int64).
  - `numafe`: number of affine expressions in AFE storage (int64).
  - `numfnz`: number of nonzeros in F (int64).
  - `numacc`: number of affine conic constraints (ACCs) (int64).
  - `numdjc`: number of disjunctive constraints (DJCs) (int64).
  - `numdom`: number of domains (int64).
  - `mosekver`: MOSEK version (list(int32)).
- `Task/data`: Numerical and structural data of the problem.
  - `var`: Information about variables. All fields present must have the same length as `bk`. All or none of `bk`, `bl`, and `bu` must appear.
    - `name`: Variable names (list(string)).
    - `bk`: Bound keys (list(string)).
    - `bl`: Lower bounds (list(double)).
    - `bu`: Upper bounds (list(double)).
    - `type`: Variable types (list(string)).
  - `con`: Information about linear constraints. All fields present must have the same length as `bk`. All or none of `bk`, `bl`, and `bu` must appear.
    - `name`: Constraint names (list(string)).
    - `bk`: Bound keys (list(string)).
    - `bl`: Lower bounds (list(double)).
    - `bu`: Upper bounds (list(double)).
  - `barvar`: Information about symmetric matrix variables. All fields present must have the same length as `dim`.
    - `name`: Barvar names (list(string)).
    - `dim`: Dimensions (list(int32)).
  - `objective`: Information about the objective.
    - `name`: Objective name (string).
    - `sense`: Objective sense (string).
    - `c`: The linear part c of the objective as a sparse vector. Both arrays must have the same length.
      - `subj`: indices of nonzeros (list(int32)).
      - `val`: values of nonzeros (list(double)).
    - `cfix`: Constant term in the objective (double).

    * `Q`: The quadratic part Q^o of the objective as a sparse matrix, only lower-triangular part included. All arrays must have the same length.
      · `subi`: row indices of nonzeros (list(int32)).
      · `subj`: column indices of nonzeros (list(int32)).
      · `val`: values of nonzeros (list(double)).
    * `barc`: The semidefinite part $\overline{C}$ of the objective (list). Each element of the list is a list describing one entry $\overline{C}_j$ using three fields:
      · index j (int32).
      · weights of the matrices from the storage E forming $\overline{C}_j$ (list(double)).
      · indices of the matrices from the storage E forming $\overline{C}_j$ (list(int64)).
- `A`: The linear constraint matrix A as a sparse matrix. All arrays must have the same length.
    * `subi`: row indices of nonzeros (list(int32)).
    * `subj`: column indices of nonzeros (list(int32)).
    * `val`: values of nonzeros (list(double)).
- `bara`: The semidefinite part $\overline{A}$ of the constraints (list). Each element of the list is a list describing one entry $\overline{A}_{ij}$ using four fields:
    * index i (int32).
    * index j (int32).
    * weights of the matrices from the storage E forming $\overline{A}_{ij}$ (list(double)).
    * indices of the matrices from the storage E forming $\overline{A}_{ij}$ (list(int64)).
- `AFE`: The affine expression storage.
    * `numafe`: number of rows in the storage (int64).
    * `F`: The matrix F as a sparse matrix. All arrays must have the same length.
      · `subi`: row indices of nonzeros (list(int64)).
      · `subj`: column indices of nonzeros (list(int32)).
      · `val`: values of nonzeros (list(double)).
    * `g`: The vector g of constant terms as a sparse vector. Both arrays must have the same length.
      · `subi`: indices of nonzeros (list(int64)).
      · `val`: values of nonzeros (list(double)).
    * `barf`: The semidefinite part $\overline{F}$ of the expressions in AFE storage (list). Each element of the list is a list describing one entry $\overline{F}_{ij}$ using four fields:
      · index i (int64).
      · index j (int32).
      · weights of the matrices from the storage E forming $\overline{F}_{ij}$ (list(double)).
      · indices of the matrices from the storage E forming $\overline{F}_{ij}$ (list(int64)).
- `domains`: Information about domains. All fields present must have the same length as `type`.
    * `name`: Domain names (list(string)).
    * `type`: Description of the type of each domain (list). Each element of the list is a list describing one domain using at least one field:
      · domain type (string).
      · (except `pexp`, `dexp`) dimension (int64).
      · (only `ppow`, `dpow`) weights (list(double)).
- `ACC`: Information about affine conic constraints (ACC). All fields present must have the same length as `domain`.
    * `name`: ACC names (list(string)).
    * `domain`: Domains (list(int64)).
    * `afeidx`: AFE indices, grouped by ACC (list(list(int64))).
    * `b`: constant vectors b, grouped by ACC (list(list(double))).

- `DJC`: Information about disjunctive constraints (DJC). All fields present must have the same length as `termsize`.
  - `name`: DJC names (list(string)).
  - `termsize`: Term sizes, grouped by DJC (list(list(int64))).
  - `domain`: Domains, grouped by DJC (list(list(int64))).
  - `afeidx`: AFE indices, grouped by DJC (list(list(int64))).
  - `b`: constant vectors b, grouped by DJC (list(list(double))).
- `MatrixStore`: The symmetric matrix storage E (list). Each element of the list is a list describing one entry E using four fields in sparse matrix format, lower-triangular part only:
  - dimension (int32).
  - row indices of nonzeros (list(int32)).
  - column indices of nonzeros (list(int32)).
  - values of nonzeros (list(double)).
- `Q`: The quadratic part Q^c of the constraints (list). Each element of the list is a list describing one entry Q_i^c using four fields in sparse matrix format, lower-triangular part only:
  - the row index i (int32).
  - row indices of nonzeros (list(int32)).
  - column indices of nonzeros (list(int32)).
  - values of nonzeros (list(double)).
- `qcone` (deprecated). The description of cones. All fields present must have the same length as `type`.
  - `name`: Cone names (list(string)).
  - `type`: Cone types (list(string)).
  - `par`: Additional cone parameters (list(double)).
  - `members`: Members, grouped by cone (list(list(int32))).

- `Task/solutions`: Solutions. This section can contain up to three subsections called:
  - `interior`
  - `basic`
  - `integer`

  corresponding to the three solution types in MOSEK. Each of these sections has the same structure:
  - `prosta`: problem status (string).
  - `solsta`: solution status (string).
  - `xx`, `xc`, `y`, `slc`, `suc`, `slx`, `sux`, `snx`: one for each component of the solution of the same name (list(double)).
  - `skx`, `skc`, `skn`: status keys (list(string)).
  - `doty`: the dual $\dot{y}$ solution, grouped by ACC (list(list(double))).
  - `barx`, `bars`: the primal/dual semidefinite solution, grouped by matrix variable (list(list(double))).

- `Task/parameters`: Parameters.
  - `iparam`: Integer parameters (dictionary). A dictionary with entries of the form `name:value`, where `name` is a shortened parameter name (without leading `MSK_IPAR_`) and `value` is either an integer or string if the parameter takes values from an enum.
  - `dparam`: Double parameters (dictionary). A dictionary with entries of the form `name:value`, where `name` is a shortened parameter name (without leading `MSK_DPAR_`) and `value` is a double.
  - `sparam`: String parameters (dictionary). A dictionary with entries of the form `name:value`, where `name` is a shortened parameter name (without leading `MSK_SPAR_`) and `value` is a string. Note that this section is allowed but MOSEK ignores it both when writing and reading JTASK files.

16.7.2 JSOL Specification

The JSOL is a dictionary containing the following sections. All sections are optional and can be omitted if irrelevant for the problem.

- `$schema`: JSON schema.
- `Task/name`: The name of the task (string).
- `Task/solutions`: Solutions. This section can contain up to three subsections called:
  - `interior`
  - `basic`
  - `integer`

  corresponding to the three solution types in MOSEK. Each of these section has the same structure:
  - `prosta`: problem status (string).
  - `solsta`: solution status (string).
  - `xx`, `xc`, `y`, `slc`, `suc`, `slx`, `sux`, `snx`: one for each component of the solution of the same name (list(double)).
  - `skx`, `skc`, `skn`: status keys (list(string)).
  - `doty`: the dual $\dot{y}$ solution, grouped by ACC (list(list(double))).
  - `barx`, `bars`: the primal/dual semidefinite solution, grouped by matrix variable (list(list(double))).
- `Task/information`: Information items from the optimizer.
  - `int32`: int32 information items (dictionary). A dictionary with entries of the form `name: value`.
  - `int64`: int64 information items (dictionary). A dictionary with entries of the form `name: value`.
  - `double`: double information items (dictionary). A dictionary with entries of the form `name: value`.

16.7.3 A `jtask` example

Listing 16.5: A formatted `jtask` file for a simple portfolio optimization problem.

```
{
   "$schema":"http://mosek.com/json/schema#",
   "Task/name":"Markowitz portfolio with market impact",
   "Task/INFO":{"numvar":7,"numcon":1,"numcone":0,"numbarvar":0,"numanz":6,"numsymmat
↪":0,"numafe":13,"numfnz":12,"numacc":4,"numdjc":0,"numdom":3,"mosekver":[10,0,0,3]},
   "Task/data":{
      "var":{
         "name":["1.0","x[0]","x[1]","x[2]","t[0]","t[1]","t[2]"],
         "bk":["fx","lo","lo","lo","fr","fr","fr"],
         "bl":[1,0.0,0.0,0.0,-1e+30,-1e+30,-1e+30],
         "bu":[1,1e+30,1e+30,1e+30,1e+30,1e+30,1e+30],
         "type":["cont","cont","cont","cont","cont","cont","cont"]
      },
      "con":{
         "name":["budget[]"],
         "bk":["fx"],
         "bl":[1],
```

(continues on next page)

(continued from previous page)


```
         "bu":[1]
      },
      "objective":{
         "sense":"max",
         "name":"obj",
         "c":{
            "subj":[1,2,3],
            "val":[0.1073,0.0737,0.0627]
         },
         "cfix":0.0
      },
      "A":{
         "subi":[0,0,0,0,0,0],
         "subj":[1,2,3,4,5,6],
         "val":[1,1,1,0.01,0.01,0.01]
      },
      "AFE":{
         "numafe":13,
         "F":{
            "subi":[1,1,1,2,2,3,4,6,7,9,10,12],
            "subj":[1,2,3,2,3,3,4,1,5,2,6,3],
            "val":[0.166673333200005,0.0232190712557243,0.0012599496030238,0.
↪102863378954911,-0.00222873156550421,0.0338148677744977,1,1,1,1,1,1]
         },
         "g":{
            "subi":[0,5,8,11],
            "val":[0.035,1,1,1]
         }
      },
      "domains":{
         "type":[["r",0],
                 ["quad",4],
                 ["ppow",3,[0.6666666666666666,0.3333333333333337]]]
      },
      "ACC":{
         "name":["risk[]","tz[0]","tz[1]","tz[2]"],
         "domain":[1,2,2,2],
         "afeidx":[[0,1,2,3],
                   [4,5,6],
                   [7,8,9],
                   [10,11,12]]
      }
   },
   "Task/solutions":{
      "interior":{
         "prosta":"unknown",
         "solsta":"unknown",
         "skx":["fix","supbas","supbas","supbas","supbas","supbas","supbas"],
         "skc":["fix"],
         "xx":[1,0.10331580274282556,0.11673185566457132,0.7724326587076371,0.
↪033208600335718846,0.03988270849469869,0.6788769587942524],
         "xc":[1],
         "slx":[0.0,-5.585840467641202e-10,-8.945844685006369e-10,-7.815248786428623e-
↪11,0.0,0.0,0.0],
         "sux":[0.0,0.0,0.0,0.0,0.0,0.0,0.0],
         "snx":[0.0,0.0,0.0,0.0,0.0,0.0,0.0],
```


(continues on next page)

(continued from previous page)

```
         "slc":[0.0],
         "suc":[-0.046725814048521205],
         "y":[0.046725814048521205],
         "doty":[[-0.6062603164682975,0.3620818321879349,0.17817754087278295,0.
↪4524390346223723],
                 [-4.6725842015519993e-4,-7.708781121860897e-6,2.24800624747081e-4],
                 [-4.6725842015519993e-4,-9.268264309496919e-6,2.390390600079771e-4],
                 [-4.6725842015519993e-4,-1.5854982159992136e-4,6.159249331148646e-4]]
      }
   },
   "Task/parameters":{
      "iparam":{
         "LICENSE_DEBUG":"ON",
         "MIO_SEED":422
      },
      "dparam":{
         "MIO_MAX_TIME":100
      },
      "sparam":{
      }
   }
}
```

16.8 The Solution File Format

MOSEK can output solutions to a text file:

- *basis solution file* (extension `.bas`) if the problem is optimized using the simplex optimizer or basis identification is performed,
- *interior solution file* (extension `.sol`) if a problem is optimized using the interior-point optimizer and no basis identification is required,
- *integer solution file* (extension `.int`) if the problem is solved with the mixed-integer optimizer.

All solution files have the format:

```
NAME                : <problem name>
PROBLEM STATUS      : <status of the problem>
SOLUTION STATUS     : <status of the solution>
OBJECTIVE NAME      : <name of the objective function>
PRIMAL OBJECTIVE    : <primal objective value corresponding to the solution>
DUAL OBJECTIVE      : <dual objective value corresponding to the solution>

CONSTRAINTS
INDEX  NAME     AT ACTIVITY    LOWER LIMIT   UPPER LIMIT   DUAL LOWER   DUAL UPPER
?      <name>   ?? <a value>   <a value>     <a value>     <a value>    <a value>

AFFINE CONIC CONSTRAINTS
INDEX  NAME     I          ACTIVITY    DUAL
?      <name>   <a value>  <a value>   <a value>

VARIABLES
INDEX  NAME     AT ACTIVITY    LOWER LIMIT   UPPER LIMIT   DUAL LOWER   DUAL UPPER  ␣
↪[CONIC DUAL]
?      <name>   ?? <a value>   <a value>     <a value>     <a value>    <a value>   ␣
↪[<a value>]
```

(continues on next page)

(continued from previous page)

```
SYMMETRIC MATRIX VARIABLES
INDEX  NAME     I           J             PRIMAL        DUAL
?      <name>   <a value>   <a value>     <a value>     <a value>
```

The fields `?`, `??` and `<>` will be filled with problem and solution specific information as described below. The solution contains sections corresponding to parts of the input. Empty sections may be omitted and fields in `[]` are optional, depending on what type of problem is solved. The notation below follows the **MOSEK** naming convention for parts of the solution as defined in the problem specifications in Sec. 12.

- `HEADER`
  In this section, first the name of the problem is listed and afterwards the problem and solution status are shown. Next the primal and dual objective values are displayed.
- `CONSTRAINTS`
  - `INDEX`: A sequential index assigned to the constraint by **MOSEK**
  - `NAME`: The name of the constraint assigned by the user or autogenerated.
  - `AT`: The status key `bkc` of the constraint as in Table 16.4.
  - `ACTIVITY`: the activity `xc` of the constraint expression.
  - `LOWER LIMIT`: the lower bound `blc` of the constraint.
  - `UPPER LIMIT`: the upper bound `buc` of the constraint.
  - `DUAL LOWER`: the dual multiplier `slc` corresponding to the lower limit on the constraint.
  - `DUAL UPPER`: the dual multiplier `suc` corresponding to the upper limit on the constraint.
- `AFFINE CONIC CONSTRAINTS`
  - `INDEX`: A sequential index assigned to the affine expressions by **MOSEK**
  - `NAME`: The name of the affine conic constraint assigned by the user or autogenerated.
  - `I`: The sequential index of the affine expression in the affine conic constraint.
  - `ACTIVITY`: the activity of the `I`-th affine expression in the affine conic constraint.
  - `DUAL`: the dual multiplier `doty` for the `I`-th entry in the affine conic constraint.
- `VARIABLES`
  - `INDEX`: A sequential index assigned to the variable by **MOSEK**
  - `NAME`: The name of the variable assigned by the user or autogenerated.
  - `AT`: The status key `bkx` of the variable as in Table 16.4.
  - `ACTIVITY`: the value `xx` of the variable.
  - `LOWER LIMIT`: the lower bound `blx` of the variable.
  - `UPPER LIMIT`: the upper bound `bux` of the variable.
  - `DUAL LOWER`: the dual multiplier `slx` corresponding to the lower limit on the variable.
  - `DUAL UPPER`: the dual multiplier `sux` corresponding to the upper limit on the variable.
  - `CONIC DUAL`: the dual multiplier `skx` corresponding to a conic variable (deprecated).
- `SYMMETRIC MATRIX VARIABLES`
  - `INDEX`: A sequential index assigned to each symmetric matrix entry by **MOSEK**
  - `NAME`: The name of the symmetric matrix variable assigned by the user or autogenerated.
  - `I`: The row index in the symmetric matrix variable.
  - `J`: The column index in the symmetric matrix variable.
  - `PRIMAL`: the value of `barx` for the (`I`, `J`)-th entry in the symmetric matrix variable.

– `DUAL`: the dual multiplier `bars` for the (`I`, `J`)-th entry in the symmetric matrix variable.

Table 16.4: Status keys.

Status key	Interpretation
`UN`	Unknown status
`BS`	Is basic
`SB`	Is superbasic
`LL`	Is at the lower limit (bound)
`UL`	Is at the upper limit (bound)
`EQ`	Lower limit is identical to upper limit
`**`	Is infeasible i.e. the lower limit is greater than the upper limit.

Example.

Below is an example of a solution file.

Listing 16.6: An example of `.sol` file.

```
NAME                 :
PROBLEM STATUS       : PRIMAL_AND_DUAL_FEASIBLE
SOLUTION STATUS      : OPTIMAL
OBJECTIVE NAME       : OBJ
PRIMAL OBJECTIVE     : 0.70571049347734
DUAL OBJECTIVE       : 0.70571048919757

CONSTRAINTS
INDEX      NAME           AT ACTIVITY                LOWER LIMIT        UPPER LIMIT ␣
↪      DUAL LOWER               DUAL UPPER

AFFINE CONIC CONSTRAINTS
INDEX      NAME           I          ACTIVITY                DUAL
0          A1             0          1.0000000009656         0.54475821296644
1          A1             1          0.50000000152223        0.32190455246225
2          A2             0          0.25439922724695        0.4552417870329
3          A2             1          0.17988741850378        -0.32190455246178
4          A2             2          0.17988741850378        -0.32190455246178

VARIABLES
INDEX      NAME           AT ACTIVITY                LOWER LIMIT        UPPER LIMIT ␣
↪      DUAL LOWER               DUAL UPPER
0          X1             SB 0.25439922724695        NONE               NONE        ␣
↪      0                        0
1          X2             SB 0.17988741850378        NONE               NONE        ␣
↪      0                        0
2          X3             SB 0.17988741850378        NONE               NONE        ␣
↪      0                        0

SYMMETRIC MATRIX VARIABLES
INDEX      NAME           I          J          PRIMAL                  DUAL
0          BARX1          0          0          0.21725733689874        1.1333372337141
1          BARX1          1          0          -0.25997257078534       0.
↪67809544651396
2          BARX1          2          0          0.21725733648507        -0.
↪3219045527104
3          BARX1          1          1          0.31108610088839        1.1333372332693
4          BARX1          2          1          -0.25997257078534       0.
```

(continues on next page)

(continued from previous page)

```
↪67809544651435
5           BARX1           2           2           0.21725733689874          1.1333372337145
6           BARX2           0           0           4.8362272828127e-10       0.
↪54475821339698
7           BARX2           1           0           0                         0
8           BARX2           1           1           4.8362272828127e-10       0.
↪54475821339698
```

Chapter 17

List of examples

List of examples shipped in the distribution of Optimizer API for Julia:

Table 17.1: List of distributed examples

File	Description
acc1.jl	A simple problem with one affine conic constraint (ACC)
acc2.jl	A simple problem with two affine conic constraints (ACC)
callback.jl	An example of data/progress callback
ceo1.jl	A simple conic exponential problem
concurrent1.jl	Implementation of a concurrent optimizer for linear and mixed-integer problems
cqo1.jl	A simple conic quadratic problem
djc1.jl	A simple problem with disjunctive constraints (DJC)
feasrepairex1.jl	A simple example of how to repair an infeasible problem
gp1.jl	A simple geometric program (GP) in conic form
helloworld.jl	A Hello World example
lo1.jl	A simple linear problem
lo2.jl	A simple linear problem
logistic.jl	Implements logistic regression and simple log-sum-exp (CEO)
mico1.jl	A simple mixed-integer conic problem
milo1.jl	A simple mixed-integer linear problem
mioinitsol.jl	A simple mixed-integer linear problem with an initial guess
opt_server_async.jl	Uses **MOSEK** OptServer to solve an optimization problem asynchronously
opt_server_sync.jl	Uses **MOSEK** OptServer to solve an optimization problem synchronously
parallel.jl	Demonstrates parallel optimization using a batch method in MOSEK
parameters.jl	Shows how to set optimizer parameters and read information items
pinfeas.jl	Shows how to obtain and analyze a primal infeasibility certificate
portfolio_1_basi.jl	Portfolio optimization - basic Markowitz model
portfolio_2_fron.jl	Portfolio optimization - efficient frontier
portfolio_3_impa.jl	Portfolio optimization - market impact costs
portfolio_4_tran.jl	Portfolio optimization - transaction costs
portfolio_5_card.jl	Portfolio optimization - cardinality constraints
portfolio_6_fact.jl	Portfolio optimization - factor model
pow1.jl	A simple power cone problem
qcqo1.jl	A simple quadratically constrained quadratic problem

continues on next page

Table 17.1 – continued from previous page

File	Description
qo1.jl	A simple quadratic problem
reoptimization.jl	Demonstrate how to modify and re-optimize a linear problem
response.jl	Demonstrates proper response handling
sdo1.jl	A simple semidefinite problem with one matrix variable and a quadratic cone
sdo2.jl	A simple semidefinite problem with two matrix variables
sdo_lmi.jl	A simple semidefinite problem with an LMI using the SVEC domain.
sensitivity.jl	Sensitivity analysis performed on a small linear problem
simple.jl	A simple I/O example: read problem from a file, solve and write solutions
solutionquality.jl	Demonstrates how to examine the quality of a solution
solvebasis.jl	Demonstrates solving a linear system with the basis matrix
solvelinear.jl	Demonstrates solving a general linear system
sparsecholesky.jl	Shows how to find a Cholesky factorization of a sparse matrix

Additional examples can be found on the **MOSEK** website and in other **MOSEK** publications.

Chapter 18

Interface changes

The section shows interface-specific changes to the **MOSEK** Optimizer API for Julia in version 11.2 compared to version 10. See the release notes for general changes and new features of the **MOSEK** Optimization Suite.

18.1 Important changes compared to version 10

- **Parameters.** Users who set parameters to tune the performance and numerical properties of the solver (termination criteria, tolerances, solving primal or dual, presolve etc.) are recommended to reevaluate such tuning. It may be that other, or default, parameter settings will be more beneficial in the current version. The hints in Sec. 8 may be useful for some cases.

18.2 Changes compared to version 10

18.2.1 Functions compared to version 10

Added

- *appenddualpowerconedomainseq*
- *appendprimalpowerconedomainseq*
- *getdualproblem*
- *getlintparam*
- *putlintparam*
- *resetdouparam*
- *resetintparam*
- *resetparameters*
- *resetstrparam*

Removed

- `Task.setdefaults`

18.2.2 Parameters compared to version 10

Added

- *MSK_DPAR_FOLDING_TOL_EQ*
- *MSK_DPAR_MIO_CLIQUE_TABLE_SIZE_FACTOR*
- *MSK_DPAR_SIM_PRECISION_SCALING_EXTENDED*
- *MSK_DPAR_SIM_PRECISION_SCALING_NORMAL*
- *MSK_IPAR_FOLDING_USE*
- *MSK_IPAR_GETDUAL_CONVERT_LMIS*
- *MSK_IPAR_HEARTBEAT_SIM_FREQ_TICKS*
- *MSK_IPAR_LOG_SIM_FREQ_GIGA_TICKS*
- *MSK_IPAR_MIO_CONFLICT_ANALYSIS_LEVEL*
- *MSK_IPAR_MIO_CROSSOVER_MAX_NODES*
- *MSK_IPAR_MIO_INDEPENDENT_BLOCK_LEVEL*
- *MSK_IPAR_MIO_OPT_FACE_MAX_NODES*
- *MSK_IPAR_MIO_RENS_MAX_NODES*
- *MSK_IPAR_PTF_WRITE_SINGLE_PSD_TERMS*
- *MSK_IPAR_READ_ASYNC*
- *MSK_IPAR_SIM_PRECISION*
- *MSK_IPAR_SIM_PRECISION_BOOST*
- *MSK_IPAR_WRITE_ASYNC*

Removed

- dparam.check_convexity_rel_tol
- dparam.presolve_tol_aij
- iparam.infeas_prefer_primal
- iparam.intpnt_max_num_refinement_steps
- iparam.intpnt_purify
- iparam.log_response
- iparam.log_sim_minor
- iparam.mio_root_repeat_presolve_level
- iparam.presolve_level
- iparam.sensitivity_optimizer
- iparam.sim_stability_priority
- iparam.sol_filter_keep_ranged
- iparam.solution_callback
- iparam.write_data_param
- iparam.write_generic_names_io
- iparam.write_task_inc_sol
- iparam.write_xml_mode
- sparam.write_lp_gen_var_name

18.2.3 Constants compared to version 10

Added

- *MSK_CALLBACK_BEGIN_FOLDING*
- *MSK_CALLBACK_BEGIN_FOLDING_BI*
- *MSK_CALLBACK_BEGIN_FOLDING_BI_DUAL*
- *MSK_CALLBACK_BEGIN_FOLDING_BI_INITIALIZE*
- *MSK_CALLBACK_BEGIN_FOLDING_BI_OPTIMIZER*
- *MSK_CALLBACK_BEGIN_FOLDING_BI_PRIMAL*
- *MSK_CALLBACK_BEGIN_INITIALIZE_BI*
- *MSK_CALLBACK_BEGIN_OPTIMIZE_BI*
- *MSK_CALLBACK_DECOMP_MIO*
- *MSK_CALLBACK_END_FOLDING*
- *MSK_CALLBACK_END_FOLDING_BI*
- *MSK_CALLBACK_END_FOLDING_BI_DUAL*
- *MSK_CALLBACK_END_FOLDING_BI_INITIALIZE*
- *MSK_CALLBACK_END_FOLDING_BI_OPTIMIZER*
- *MSK_CALLBACK_END_FOLDING_BI_PRIMAL*
- *MSK_CALLBACK_END_INITIALIZE_BI*
- *MSK_CALLBACK_END_OPTIMIZE_BI*
- *MSK_CALLBACK_FOLDING_BI_DUAL*
- *MSK_CALLBACK_FOLDING_BI_OPTIMIZER*
- *MSK_CALLBACK_FOLDING_BI_PRIMAL*
- *MSK_CALLBACK_HEARTBEAT*
- *MSK_CALLBACK_OPTIMIZE_BI*
- *MSK_CALLBACK_QO_REFORMULATE*
- *MSK_DINF_FOLDING_BI_OPTIMIZE_TIME*
- *MSK_DINF_FOLDING_BI_UNFOLD_DUAL_TIME*
- *MSK_DINF_FOLDING_BI_UNFOLD_INITIALIZE_TIME*
- *MSK_DINF_FOLDING_BI_UNFOLD_PRIMAL_TIME*
- *MSK_DINF_FOLDING_BI_UNFOLD_TIME*
- *MSK_DINF_FOLDING_FACTOR*
- *MSK_DINF_FOLDING_TIME*
- *MSK_IINF_FOLDING_APPLIED*

- *MSK_IINF_MIO_FINAL_NUMBIN*
- *MSK_IINF_MIO_FINAL_NUMBINCONEVAR*
- *MSK_IINF_MIO_FINAL_NUMCON*
- *MSK_IINF_MIO_FINAL_NUMCONE*
- *MSK_IINF_MIO_FINAL_NUMCONEVAR*
- *MSK_IINF_MIO_FINAL_NUMCONT*
- *MSK_IINF_MIO_FINAL_NUMCONTCONEVAR*
- *MSK_IINF_MIO_FINAL_NUMDEXPCONES*
- *MSK_IINF_MIO_FINAL_NUMDJC*
- *MSK_IINF_MIO_FINAL_NUMDPOWCONES*
- *MSK_IINF_MIO_FINAL_NUMINT*
- *MSK_IINF_MIO_FINAL_NUMINTCONEVAR*
- *MSK_IINF_MIO_FINAL_NUMPEXPCONES*
- *MSK_IINF_MIO_FINAL_NUMPPOWCONES*
- *MSK_IINF_MIO_FINAL_NUMQCONES*
- *MSK_IINF_MIO_FINAL_NUMRQCONES*
- *MSK_IINF_MIO_FINAL_NUMVAR*
- *MSK_IINF_MIO_NUM_BLOCKS_SOLVED_IN_BB*
- *MSK_IINF_MIO_NUM_BLOCKS_SOLVED_IN_PRESOLVE*
- *MSK_LIINF_BI_CLEAN_ITER*
- *MSK_LIINF_FOLDING_BI_DUAL_ITER*
- *MSK_LIINF_FOLDING_BI_OPTIMIZER_ITER*
- *MSK_LIINF_FOLDING_BI_PRIMAL_ITER*
- *MSK_LIINF_MIO_FINAL_ANZ*
- *MSK_OPTIMIZER_NEW_DUAL_SIMPLEX*
- *MSK_OPTIMIZER_NEW_PRIMAL_SIMPLEX*

Removed

- `constant.callbackcode.begin_simplex_bi`
- `constant.callbackcode.im_bi`
- `constant.callbackcode.im_conic`
- `constant.callbackcode.im_dual_bi`
- `constant.callbackcode.im_intpnt`
- `constant.callbackcode.im_presolve`

- constant.callbackcode.im_primal_bi
- constant.callbackcode.im_qo_reformulate
- constant.callbackcode.im_simplex_bi
- constant.dinfitem.bi_clean_dual_time
- constant.dinfitem.bi_clean_primal_time
- constant.liinfitem.bi_clean_dual_deg_iter
- constant.liinfitem.bi_clean_dual_iter
- constant.liinfitem.bi_clean_primal_deg_iter
- constant.liinfitem.bi_clean_primal_iter

18.2.4 Response Codes compared to version 10

Added

- *MSK_RES_ERR_GETDUAL_NOT_AVAILABLE*
- *MSK_RES_ERR_READ_ASYNC*
- *MSK_RES_ERR_READ_PREMATURE_EOF*
- *MSK_RES_ERR_READ_WRITE*
- *MSK_RES_ERR_SERVER_HARD_TIMEOUT*
- *MSK_RES_ERR_TASK_PREMATURE_EOF*
- *MSK_RES_ERR_WRITE_ASYNC*
- *MSK_RES_ERR_WRITE_LP_DUPLICATE_CON_NAMES*
- *MSK_RES_ERR_WRITE_LP_DUPLICATE_VAR_NAMES*
- *MSK_RES_ERR_WRITE_LP_INVALID_CON_NAMES*
- *MSK_RES_ERR_WRITE_LP_INVALID_VAR_NAMES*
- *MSK_RES_TRM_SERVER_MAX_MEMORY*
- *MSK_RES_TRM_SERVER_MAX_TIME*
- *MSK_RES_WRN_GETDUAL_IGNORES_INTEGRALITY*
- *MSK_RES_WRN_PRESOLVE_PRIMAL_PERTURBATIONS*
- *MSK_RES_WRN_PTF_UNKNOWN_SECTION*

Removed

- rescode.err_invalid_ampl_stub
- rescode.err_size_license_numcores
- rescode.err_xml_invalid_problem_type
- rescode.wrn_presolve_primal_pertubations
- rescode.wrn_write_lp_duplicate_con_names
- rescode.wrn_write_lp_duplicate_var_names
- rescode.wrn_write_lp_invalid_con_names
- rescode.wrn_write_lp_invalid_var_names

Bibliography

[AA95] E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Math. Programming*, 71(2):221–245, 1995.

[AGMeszarosX96] E. D. Andersen, J. Gondzio, Cs. Mészáros, and X. Xu. Implementation of interior point methods for large scale linear programming. In T. Terlaky, editor, *Interior-point methods of mathematical programming*, pages 189–252. Kluwer Academic Publishers, 1996.

[ART03] E. D. Andersen, C. Roos, and T. Terlaky. On implementing a primal-dual interior-point method for conic quadratic optimization. *Math. Programming*, February 2003.

[AY96] E. D. Andersen and Y. Ye. Combining interior-point and pivoting algorithms. *Management Sci.*, 42(12):1719–1731, December 1996.

[And09] Erling D. Andersen. The homogeneous and self-dual model and algorithm for linear optimization. Technical Report TR-1-2009, MOSEK ApS, 2009. URL: http://docs.mosek.com/whitepapers/homolo.pdf.

[And13] Erling D. Andersen. On formulating quadratic functions in optimization models. Technical Report TR-1-2013, MOSEK ApS, 2013. Last revised 23-feb-2016. URL: http://docs.mosek.com/whitepapers/qmodel.pdf.

[BKVH07] S. Boyd, S.J. Kim, L. Vandenberghe, and A. Hassibi. A Tutorial on Geometric Programming. *Optimization and Engineering*, 8(1):67–127, 2007. Available at http://www.stanford.edu/\protect\unhbox\voidb@x\penalty\@M\boyd/gp_tutorial.html.

[Chvatal83] V. Chvátal. *Linear programming*. W.H. Freeman and Company, 1983.

[CCornuejolsZ14] M. Conforti, G. Cornu/'ejols, and G. Zambelli. *Integer programming*. Springer, 2014.

[GK00] Richard C. Grinold and Ronald N. Kahn. *Active portfolio management*. McGraw-Hill, New York, 2 edition, 2000.

[Naz87] J. L. Nazareth. *Computer Solution of Linear Programs*. Oxford University Press, New York, 1987.

[RTV97] C. Roos, T. Terlaky, and J. -Ph. Vial. *Theory and algorithms for linear optimization: an interior point approach*. John Wiley and Sons, New York, 1997.

[Ste98] G. W. Stewart. *Matrix Algorithms. Volume 1: Basic decompositions*. SIAM, 1998.

[Wal00] S. W. Wallace. Decision making under uncertainty: is sensitivity of any use. *Oper. Res.*, 48(1):20–25, January 2000.

[Wol98] L. A. Wolsey. *Integer programming*. John Wiley and Sons, 1998.

Symbol Index

Enumerations

basindtype, 492
MSK_BI_RESERVERED, 492
MSK_BI_NO_ERROR, 492
MSK_BI_NEVER, 492
MSK_BI_IF_FEASIBLE, 492
MSK_BI_ALWAYS, 492
boundkey, 492
MSK_BK_UP, 492
MSK_BK_RA, 492
MSK_BK_LO, 492
MSK_BK_FX, 492
MSK_BK_FR, 492
branchdir, 515
MSK_BRANCH_DIR_UP, 515
MSK_BRANCH_DIR_ROOT_LP, 515
MSK_BRANCH_DIR_PSEUDOCOST, 516
MSK_BRANCH_DIR_NEAR, 515
MSK_BRANCH_DIR_GUIDED, 516
MSK_BRANCH_DIR_FREE, 515
MSK_BRANCH_DIR_FAR, 515
MSK_BRANCH_DIR_DOWN, 515
callbackcode, 494
MSK_CALLBACK_WRITE_OPF, 499
MSK_CALLBACK_UPDATE_SIMPLEX, 499
MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX_BI, 499
MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX, 499
MSK_CALLBACK_UPDATE_PRIMAL_BI, 499
MSK_CALLBACK_UPDATE_PRESOLVE, 499
MSK_CALLBACK_UPDATE_DUAL_SIMPLEX_BI, 499
MSK_CALLBACK_UPDATE_DUAL_SIMPLEX, 499
MSK_CALLBACK_UPDATE_DUAL_BI, 498
MSK_CALLBACK_SOLVING_REMOTE, 498
MSK_CALLBACK_RESTART_MIO, 498
MSK_CALLBACK_READ_OPF_SECTION, 498
MSK_CALLBACK_READ_OPF, 498
MSK_CALLBACK_QO_REFORMULATE, 498
MSK_CALLBACK_PRIMAL_SIMPLEX, 498
MSK_CALLBACK_OPTIMIZE_BI, 498
MSK_CALLBACK_NEW_INT_MIO, 498
MSK_CALLBACK_INTPNT, 498
MSK_CALLBACK_IM_SIMPLEX, 498
MSK_CALLBACK_IM_ROOT_CUTGEN, 498
MSK_CALLBACK_IM_READ, 498
MSK_CALLBACK_IM_PRIMAL_SIMPLEX, 498
MSK_CALLBACK_IM_PRIMAL_SENSIVITY, 498
MSK_CALLBACK_IM_ORDER, 498
MSK_CALLBACK_IM_MIO_PRIMAL_SIMPLEX, 498
MSK_CALLBACK_IM_MIO_INTPNT, 498
MSK_CALLBACK_IM_MIO_DUAL_SIMPLEX, 498
MSK_CALLBACK_IM_MIO, 498
MSK_CALLBACK_IM_LU, 497
MSK_CALLBACK_IM_LICENSE_WAIT, 497
MSK_CALLBACK_IM_DUAL_SIMPLEX, 497
MSK_CALLBACK_IM_DUAL_SENSIVITY, 497
MSK_CALLBACK_HEARTBEAT, 497
MSK_CALLBACK_FOLDING_BI_PRIMAL, 497
MSK_CALLBACK_FOLDING_BI_OPTIMIZER, 497
MSK_CALLBACK_FOLDING_BI_DUAL, 497
MSK_CALLBACK_END_WRITE, 497
MSK_CALLBACK_END_TO_CONIC, 497
MSK_CALLBACK_END_SOLVE_ROOT_RELAX, 497
MSK_CALLBACK_END_SIMPLEX_BI, 497
MSK_CALLBACK_END_SIMPLEX, 497
MSK_CALLBACK_END_ROOT_CUTGEN, 497
MSK_CALLBACK_END_READ, 497
MSK_CALLBACK_END_QCQO_REFORMULATE, 497
MSK_CALLBACK_END_PRIMAL_SIMPLEX_BI, 497
MSK_CALLBACK_END_PRIMAL_SIMPLEX, 497
MSK_CALLBACK_END_PRIMAL_SETUP_BI, 497
MSK_CALLBACK_END_PRIMAL_SENSITIVITY, 497
MSK_CALLBACK_END_PRIMAL_REPAIR, 497
MSK_CALLBACK_END_PRIMAL_BI, 496
MSK_CALLBACK_END_PRESOLVE, 496
MSK_CALLBACK_END_OPTIMIZER, 496
MSK_CALLBACK_END_OPTIMIZE_BI, 496
MSK_CALLBACK_END_MIO, 496
MSK_CALLBACK_END_LICENSE_WAIT, 496
MSK_CALLBACK_END_INTPNT, 496
MSK_CALLBACK_END_INITIALIZE_BI, 496
MSK_CALLBACK_END_INFEAS_ANA, 496
MSK_CALLBACK_END_FOLDING_BI_PRIMAL, 496
MSK_CALLBACK_END_FOLDING_BI_OPTIMIZER, 496
MSK_CALLBACK_END_FOLDING_BI_INITIALIZE, 496
MSK_CALLBACK_END_FOLDING_BI_DUAL, 496
MSK_CALLBACK_END_FOLDING_BI, 496
MSK_CALLBACK_END_FOLDING, 496
MSK_CALLBACK_END_DUAL_SIMPLEX_BI, 496
MSK_CALLBACK_END_DUAL_SIMPLEX, 496
MSK_CALLBACK_END_DUAL_SETUP_BI, 496
MSK_CALLBACK_END_DUAL_SENSITIVITY, 496
MSK_CALLBACK_END_DUAL_BI, 496
MSK_CALLBACK_END_CONIC, 496
MSK_CALLBACK_END_BI, 495
MSK_CALLBACK_DUAL_SIMPLEX, 495
MSK_CALLBACK_DECOMP_MIO, 495
MSK_CALLBACK_CONIC, 495

MSK_CALLBACK_BEGIN_WRITE, 495
MSK_CALLBACK_BEGIN_TO_CONIC, 495
MSK_CALLBACK_BEGIN_SOLVE_ROOT_RELAX, 495
MSK_CALLBACK_BEGIN_SIMPLEX, 495
MSK_CALLBACK_BEGIN_ROOT_CUTGEN, 495
MSK_CALLBACK_BEGIN_READ, 495
MSK_CALLBACK_BEGIN_QCQO_REFORMULATE, 495
MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX_BI, 495
MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX, 495
MSK_CALLBACK_BEGIN_PRIMAL_SETUP_BI, 495
MSK_CALLBACK_BEGIN_PRIMAL_SENSITIVITY, 495
MSK_CALLBACK_BEGIN_PRIMAL_REPAIR, 495
MSK_CALLBACK_BEGIN_PRIMAL_BI, 495
MSK_CALLBACK_BEGIN_PRESOLVE, 495
MSK_CALLBACK_BEGIN_OPTIMIZER, 495
MSK_CALLBACK_BEGIN_OPTIMIZE_BI, 495
MSK_CALLBACK_BEGIN_MIO, 495
MSK_CALLBACK_BEGIN_LICENSE_WAIT, 494
MSK_CALLBACK_BEGIN_INTPNT, 494
MSK_CALLBACK_BEGIN_INITIALIZE_BI, 494
MSK_CALLBACK_BEGIN_INFEAS_ANA, 494
MSK_CALLBACK_BEGIN_FOLDING_BI_PRIMAL, 494
MSK_CALLBACK_BEGIN_FOLDING_BI_OPTIMIZER, 494
MSK_CALLBACK_BEGIN_FOLDING_BI_INITIALIZE, 494
MSK_CALLBACK_BEGIN_FOLDING_BI_DUAL, 494
MSK_CALLBACK_BEGIN_FOLDING_BI, 494
MSK_CALLBACK_BEGIN_FOLDING, 494
MSK_CALLBACK_BEGIN_DUAL_SIMPLEX_BI, 494
MSK_CALLBACK_BEGIN_DUAL_SIMPLEX, 494
MSK_CALLBACK_BEGIN_DUAL_SETUP_BI, 494
MSK_CALLBACK_BEGIN_DUAL_SENSITIVITY, 494
MSK_CALLBACK_BEGIN_DUAL_BI, 494
MSK_CALLBACK_BEGIN_CONIC, 494
MSK_CALLBACK_BEGIN_BI, 494
compresstype, 499
MSK_COMPRESS_ZSTD, 499
MSK_COMPRESS_NONE, 499
MSK_COMPRESS_GZIP, 499
MSK_COMPRESS_FREE, 499
conetype, 499
MSK_CT_ZERO, 500
MSK_CT_RQUAD, 499
MSK_CT_QUAD, 499
MSK_CT_PPOW, 499
MSK_CT_PEXP, 499
MSK_CT_DPOW, 499
MSK_CT_DEXP, 499
dataformat, 500
MSK_DATA_FORMAT_TASK, 501
MSK_DATA_FORMAT_PTF, 501
MSK_DATA_FORMAT_OP, 501
MSK_DATA_FORMAT_MPS, 501
MSK_DATA_FORMAT_LP, 501
MSK_DATA_FORMAT_JSON_TASK, 501
MSK_DATA_FORMAT_FREE_MPS, 501
MSK_DATA_FORMAT_EXTENSION, 500
MSK_DATA_FORMAT_CB, 501
dinfitem, 501
MSK_DINF_WRITE_DATA_TIME, 507
MSK_DINF_TO_CONIC_TIME, 507
MSK_DINF_SOL_ITR_PVIOLVAR, 507
MSK_DINF_SOL_ITR_PVIOLCONES, 507
MSK_DINF_SOL_ITR_PVIOLCON, 507
MSK_DINF_SOL_ITR_PVIOLBARVAR, 507
MSK_DINF_SOL_ITR_PVIOLACC, 507
MSK_DINF_SOL_ITR_PRIMAL_OBJ, 507
MSK_DINF_SOL_ITR_NRM_Y, 507
MSK_DINF_SOL_ITR_NRM_XX, 507
MSK_DINF_SOL_ITR_NRM_XC, 507
MSK_DINF_SOL_ITR_NRM_SUX, 507
MSK_DINF_SOL_ITR_NRM_SUC, 507
MSK_DINF_SOL_ITR_NRM_SNX, 506
MSK_DINF_SOL_ITR_NRM_SLX, 506
MSK_DINF_SOL_ITR_NRM_SLC, 506
MSK_DINF_SOL_ITR_NRM_BARX, 506
MSK_DINF_SOL_ITR_NRM_BARS, 506
MSK_DINF_SOL_ITR_DVIOLVAR, 506
MSK_DINF_SOL_ITR_DVIOLCONES, 506
MSK_DINF_SOL_ITR_DVIOLCON, 506
MSK_DINF_SOL_ITR_DVIOLBARVAR, 506
MSK_DINF_SOL_ITR_DVIOLACC, 506
MSK_DINF_SOL_ITR_DUAL_OBJ, 506
MSK_DINF_SOL_ITG_PVIOLVAR, 506
MSK_DINF_SOL_ITG_PVIOLITG, 506
MSK_DINF_SOL_ITG_PVIOLDJC, 506
MSK_DINF_SOL_ITG_PVIOLCONES, 506
MSK_DINF_SOL_ITG_PVIOLCON, 506
MSK_DINF_SOL_ITG_PVIOLBARVAR, 506
MSK_DINF_SOL_ITG_PVIOLACC, 505
MSK_DINF_SOL_ITG_PRIMAL_OBJ, 505
MSK_DINF_SOL_ITG_NRM_XX, 505
MSK_DINF_SOL_ITG_NRM_XC, 505
MSK_DINF_SOL_ITG_NRM_BARX, 505
MSK_DINF_SOL_BAS_PVIOLVAR, 505
MSK_DINF_SOL_BAS_PVIOLCON, 505
MSK_DINF_SOL_BAS_PRIMAL_OBJ, 505
MSK_DINF_SOL_BAS_NRM_Y, 505
MSK_DINF_SOL_BAS_NRM_XX, 505
MSK_DINF_SOL_BAS_NRM_XC, 505
MSK_DINF_SOL_BAS_NRM_SUX, 505
MSK_DINF_SOL_BAS_NRM_SUC, 505
MSK_DINF_SOL_BAS_NRM_SLX, 505
MSK_DINF_SOL_BAS_NRM_SLC, 505
MSK_DINF_SOL_BAS_NRM_BARX, 505
MSK_DINF_SOL_BAS_DVIOLVAR, 505
MSK_DINF_SOL_BAS_DVIOLCON, 505
MSK_DINF_SOL_BAS_DUAL_OBJ, 505
MSK_DINF_SIM_TIME, 505
MSK_DINF_SIM_PRIMAL_TIME, 504
MSK_DINF_SIM_OBJ, 504
MSK_DINF_SIM_FEAS, 504
MSK_DINF_SIM_DUAL_TIME, 504
MSK_DINF_REMOTE_TIME, 504
MSK_DINF_READ_DATA_TIME, 504

MSK_DINF_QCQO_REFORMULATE_WORST_CHOLESKY_DIAG_SCALING, 504
MSK_DINF_QCQO_REFORMULATE_WORST_CHOLESKY_COLUMN_SCALING, 504
MSK_DINF_QCQO_REFORMULATE_TIME, 504
MSK_DINF_QCQO_REFORMULATE_MAX_PERTURBATION, 504
MSK_DINF_PRIMAL_REPAIR_PENALTY_OBJ, 504
MSK_DINF_PRESOLVE_TOTAL_PRIMAL_PERTURBATION, 504
MSK_DINF_PRESOLVE_TIME, 504
MSK_DINF_PRESOLVE_LINDEP_TIME, 504
MSK_DINF_PRESOLVE_ELI_TIME, 504
MSK_DINF_OPTIMIZER_TIME, 504
MSK_DINF_OPTIMIZER_TICKS, 504
MSK_DINF_MIO_USER_OBJ_CUT, 504
MSK_DINF_MIO_TIME, 504
MSK_DINF_MIO_SYMMETRY_FACTOR, 504
MSK_DINF_MIO_SYMMETRY_DETECTION_TIME, 504
MSK_DINF_MIO_ROOT_TIME, 504
MSK_DINF_MIO_ROOT_PRESOLVE_TIME, 503
MSK_DINF_MIO_ROOT_OPTIMIZER_TIME, 503
MSK_DINF_MIO_ROOT_CUT_SEPARATION_TIME, 503
MSK_DINF_MIO_ROOT_CUT_SELECTION_TIME, 503
MSK_DINF_MIO_PROBING_TIME, 503
MSK_DINF_MIO_OBJ_REL_GAP, 503
MSK_DINF_MIO_OBJ_INT, 503
MSK_DINF_MIO_OBJ_BOUND, 503
MSK_DINF_MIO_OBJ_ABS_GAP, 503
MSK_DINF_MIO_LIPRO_SEPARATION_TIME, 503
MSK_DINF_MIO_LIPRO_SELECTION_TIME, 503
MSK_DINF_MIO_KNAPSACK_COVER_SEPARATION_TIME, 503
MSK_DINF_MIO_KNAPSACK_COVER_SELECTION_TIME, 503
MSK_DINF_MIO_INITIAL_FEASIBLE_SOLUTION_OBJ, 503
MSK_DINF_MIO_IMPLIED_BOUND_SEPARATION_TIME, 503
MSK_DINF_MIO_IMPLIED_BOUND_SELECTION_TIME, 503
MSK_DINF_MIO_GMI_SEPARATION_TIME, 502
MSK_DINF_MIO_GMI_SELECTION_TIME, 502
MSK_DINF_MIO_DUAL_BOUND_AFTER_PRESOLVE, 502
MSK_DINF_MIO_CONSTRUCT_SOLUTION_OBJ, 502
MSK_DINF_MIO_CMIR_SEPARATION_TIME, 502
MSK_DINF_MIO_CMIR_SELECTION_TIME, 502
MSK_DINF_MIO_CLIQUE_SEPARATION_TIME, 502
MSK_DINF_MIO_CLIQUE_SELECTION_TIME, 502
MSK_DINF_INTPNT_TIME, 502
MSK_DINF_INTPNT_PRIMAL_OBJ, 502
MSK_DINF_INTPNT_PRIMAL_FEAS, 502
MSK_DINF_INTPNT_ORDER_TIME, 502
MSK_DINF_INTPNT_OPT_STATUS, 502
MSK_DINF_INTPNT_FACTOR_NUM_FLOPS, 502
MSK_DINF_INTPNT_DUAL_OBJ, 502
MSK_DINF_INTPNT_DUAL_FEAS, 502
MSK_DINF_FOLDING_TIME, 502
MSK_DINF_FOLDING_FACTOR, 502
MSK_DINF_FOLDING_BI_UNFOLD_TIME, 502
MSK_DINF_FOLDING_BI_UNFOLD_PRIMAL_TIME, 502
MSK_DINF_FOLDING_BI_UNFOLD_INITIALIZE_TIME, 501
MSK_DINF_FOLDING_BI_UNFOLD_DUAL_TIME, 501
MSK_DINF_FOLDING_BI_OPTIMIZE_TIME, 501
MSK_DINF_BI_TIME, 501
MSK_DINF_BI_PRIMAL_TIME, 501
MSK_DINF_BI_DUAL_TIME, 501
MSK_DINF_BI_CLEAN_TIME, 501
MSK_DINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_DENSITY, 501
domaintype, 500
MSK_DOMAIN_SVEC_PSD_CONE, 500
MSK_DOMAIN_RZERO, 500
MSK_DOMAIN_RQUADRATIC_CONE, 500
MSK_DOMAIN_RPLUS, 500
MSK_DOMAIN_RMINUS, 500
MSK_DOMAIN_R, 500
MSK_DOMAIN_QUADRATIC_CONE, 500
MSK_DOMAIN_PRIMAL_POWER_CONE, 500
MSK_DOMAIN_PRIMAL_GEO_MEAN_CONE, 500
MSK_DOMAIN_PRIMAL_EXP_CONE, 500
MSK_DOMAIN_DUAL_POWER_CONE, 500
MSK_DOMAIN_DUAL_GEO_MEAN_CONE, 500
MSK_DOMAIN_DUAL_EXP_CONE, 500
dparam, 406
feature, 507
MSK_FEATURE_PTS, 507
MSK_FEATURE_PTON, 507
foldingmode, 518
MSK_FOLDING_MODE_OFF, 518
MSK_FOLDING_MODE_FREE_UNLESS_BASIC, 518
MSK_FOLDING_MODE_FREE, 518
MSK_FOLDING_MODE_FORCE, 519
iinfitem, 508
MSK_IINF_STO_NUM_A_REALLOC, 515
MSK_IINF_SOL_ITR_SOLSTA, 515
MSK_IINF_SOL_ITR_PROSTA, 515
MSK_IINF_SOL_ITG_SOLSTA, 515
MSK_IINF_SOL_ITG_PROSTA, 515
MSK_IINF_SOL_BAS_SOLSTA, 515
MSK_IINF_SOL_BAS_PROSTA, 515
MSK_IINF_SIM_SOLVE_DUAL, 514
MSK_IINF_SIM_PRIMAL_ITER, 514
MSK_IINF_SIM_PRIMAL_INF_ITER, 514
MSK_IINF_SIM_PRIMAL_HOTSTART_LU, 514
MSK_IINF_SIM_PRIMAL_HOTSTART, 514
MSK_IINF_SIM_PRIMAL_DEG_ITER, 514
MSK_IINF_SIM_NUMVAR, 514
MSK_IINF_SIM_NUMCON, 514
MSK_IINF_SIM_DUAL_ITER, 514
MSK_IINF_SIM_DUAL_INF_ITER, 514
MSK_IINF_SIM_DUAL_HOTSTART_LU, 514
MSK_IINF_SIM_DUAL_HOTSTART, 514
MSK_IINF_SIM_DUAL_DEG_ITER, 514
MSK_IINF_RD_PROTYPE, 514

MSK_IINF_RD_NUMVAR, 514
MSK_IINF_RD_NUMQ, 514
MSK_IINF_RD_NUMINTVAR, 514
MSK_IINF_RD_NUMCONE, 514
MSK_IINF_RD_NUMCON, 514
MSK_IINF_RD_NUMBARVAR, 514
MSK_IINF_PURIFY_PRIMAL_SUCCESS, 514
MSK_IINF_PURIFY_DUAL_SUCCESS, 514
MSK_IINF_PRESOLVE_NUM_PRIMAL_PERTURBATIONS, 513
MSK_IINF_OPTIMIZE_RESPONSE, 513
MSK_IINF_OPT_NUMVAR, 513
MSK_IINF_OPT_NUMCON, 513
MSK_IINF_MIO_USER_OBJ_CUT, 513
MSK_IINF_MIO_TOTAL_NUM_SEPARATED_CUTS, 513
MSK_IINF_MIO_TOTAL_NUM_SELECTED_CUTS, 513
MSK_IINF_MIO_RELGAP_SATISFIED, 513
MSK_IINF_MIO_PRESOLVED_NUMVAR, 513
MSK_IINF_MIO_PRESOLVED_NUMRQCONES, 513
MSK_IINF_MIO_PRESOLVED_NUMQCONES, 513
MSK_IINF_MIO_PRESOLVED_NUMPPOWCONES, 513
MSK_IINF_MIO_PRESOLVED_NUMPEXPCONES, 513
MSK_IINF_MIO_PRESOLVED_NUMINTCONEVAR, 513
MSK_IINF_MIO_PRESOLVED_NUMINT, 513
MSK_IINF_MIO_PRESOLVED_NUMDPOWCONES, 513
MSK_IINF_MIO_PRESOLVED_NUMDJC, 513
MSK_IINF_MIO_PRESOLVED_NUMDEXPCONES, 513
MSK_IINF_MIO_PRESOLVED_NUMCONTCONEVAR, 513
MSK_IINF_MIO_PRESOLVED_NUMCONT, 513
MSK_IINF_MIO_PRESOLVED_NUMCONEVAR, 513
MSK_IINF_MIO_PRESOLVED_NUMCONE, 513
MSK_IINF_MIO_PRESOLVED_NUMCON, 512
MSK_IINF_MIO_PRESOLVED_NUMBINCONEVAR, 512
MSK_IINF_MIO_PRESOLVED_NUMBIN, 512
MSK_IINF_MIO_OBJ_BOUND_DEFINED, 512
MSK_IINF_MIO_NUMVAR, 512
MSK_IINF_MIO_NUMRQCONES, 512
MSK_IINF_MIO_NUMQCONES, 512
MSK_IINF_MIO_NUMPPOWCONES, 512
MSK_IINF_MIO_NUMPEXPCONES, 512
MSK_IINF_MIO_NUMINTCONEVAR, 512
MSK_IINF_MIO_NUMINT, 512
MSK_IINF_MIO_NUMDPOWCONES, 512
MSK_IINF_MIO_NUMDJC, 512
MSK_IINF_MIO_NUMDEXPCONES, 512
MSK_IINF_MIO_NUMCONTCONEVAR, 512
MSK_IINF_MIO_NUMCONT, 512
MSK_IINF_MIO_NUMCONEVAR, 512
MSK_IINF_MIO_NUMCONE, 512
MSK_IINF_MIO_NUMCON, 512
MSK_IINF_MIO_NUMBINCONEVAR, 512
MSK_IINF_MIO_NUMBIN, 512
MSK_IINF_MIO_NUM_SOLVED_NODES, 512
MSK_IINF_MIO_NUM_SEPARATED_LIPRO_CUTS, 511
MSK_IINF_MIO_NUM_SEPARATED_KNAPSACK_COVER_CUTS, 511
MSK_IINF_MIO_NUM_SEPARATED_IMPLIED_BOUND_CUTS, 511
MSK_IINF_MIO_NUM_SEPARATED_GOMORY_CUTS, 511
MSK_IINF_MIO_NUM_SEPARATED_CMIR_CUTS, 511
MSK_IINF_MIO_NUM_SEPARATED_CLIQUE_CUTS, 511
MSK_IINF_MIO_NUM_SELECTED_LIPRO_CUTS, 511
MSK_IINF_MIO_NUM_SELECTED_KNAPSACK_COVER_CUTS, 511
MSK_IINF_MIO_NUM_SELECTED_IMPLIED_BOUND_CUTS, 511
MSK_IINF_MIO_NUM_SELECTED_GOMORY_CUTS, 511
MSK_IINF_MIO_NUM_SELECTED_CMIR_CUTS, 511
MSK_IINF_MIO_NUM_SELECTED_CLIQUE_CUTS, 511
MSK_IINF_MIO_NUM_ROOT_CUT_ROUNDS, 511
MSK_IINF_MIO_NUM_RESTARTS, 511
MSK_IINF_MIO_NUM_REPEATED_PRESOLVE, 511
MSK_IINF_MIO_NUM_RELAX, 511
MSK_IINF_MIO_NUM_INT_SOLUTIONS, 511
MSK_IINF_MIO_NUM_BRANCH, 511
MSK_IINF_MIO_NUM_BLOCKS_SOLVED_IN_PRESOLVE, 511
MSK_IINF_MIO_NUM_BLOCKS_SOLVED_IN_BB, 511
MSK_IINF_MIO_NUM_ACTIVE_ROOT_CUTS, 511
MSK_IINF_MIO_NUM_ACTIVE_NODES, 511
MSK_IINF_MIO_NODE_DEPTH, 510
MSK_IINF_MIO_INITIAL_FEASIBLE_SOLUTION, 510
MSK_IINF_MIO_FINAL_NUMVAR, 510
MSK_IINF_MIO_FINAL_NUMRQCONES, 510
MSK_IINF_MIO_FINAL_NUMQCONES, 510
MSK_IINF_MIO_FINAL_NUMPPOWCONES, 510
MSK_IINF_MIO_FINAL_NUMPEXPCONES, 510
MSK_IINF_MIO_FINAL_NUMINTCONEVAR, 510
MSK_IINF_MIO_FINAL_NUMINT, 510
MSK_IINF_MIO_FINAL_NUMDPOWCONES, 510
MSK_IINF_MIO_FINAL_NUMDJC, 510
MSK_IINF_MIO_FINAL_NUMDEXPCONES, 510
MSK_IINF_MIO_FINAL_NUMCONTCONEVAR, 510
MSK_IINF_MIO_FINAL_NUMCONT, 510
MSK_IINF_MIO_FINAL_NUMCONEVAR, 510
MSK_IINF_MIO_FINAL_NUMCONE, 510
MSK_IINF_MIO_FINAL_NUMCON, 510
MSK_IINF_MIO_FINAL_NUMBINCONEVAR, 510
MSK_IINF_MIO_FINAL_NUMBIN, 510
MSK_IINF_MIO_CONSTRUCT_SOLUTION, 510
MSK_IINF_MIO_CLIQUE_TABLE_SIZE, 509
MSK_IINF_MIO_ABSGAP_SATISFIED, 509
MSK_IINF_INTPNT_SOLVE_DUAL, 509
MSK_IINF_INTPNT_NUM_THREADS, 509
MSK_IINF_INTPNT_ITER, 509
MSK_IINF_INTPNT_FACTOR_DIM_DENSE, 509
MSK_IINF_FOLDING_APPLIED, 509
MSK_IINF_ANA_PRO_NUM_VAR_UP, 509
MSK_IINF_ANA_PRO_NUM_VAR_RA, 509
MSK_IINF_ANA_PRO_NUM_VAR_LO, 509
MSK_IINF_ANA_PRO_NUM_VAR_INT, 509
MSK_IINF_ANA_PRO_NUM_VAR_FR, 509
MSK_IINF_ANA_PRO_NUM_VAR_EQ, 509
MSK_IINF_ANA_PRO_NUM_VAR_CONT, 509
MSK_IINF_ANA_PRO_NUM_VAR_BIN, 509
MSK_IINF_ANA_PRO_NUM_VAR, 509

MSK_IINF_ANA_PRO_NUM_CON_UP, 509
MSK_IINF_ANA_PRO_NUM_CON_RA, 509
MSK_IINF_ANA_PRO_NUM_CON_LO, 509
MSK_IINF_ANA_PRO_NUM_CON_FR, 509
MSK_IINF_ANA_PRO_NUM_CON_EQ, 509
MSK_IINF_ANA_PRO_NUM_CON, 508
inftype, 515
MSK_INF_LINT_TYPE, 515
MSK_INF_INT_TYPE, 515
MSK_INF_DOU_TYPE, 515
intpnthotstart, 493
MSK_INTPNT_HOTSTART_PRIMAL_DUAL, 494
MSK_INTPNT_HOTSTART_PRIMAL, 494
MSK_INTPNT_HOTSTART_NONE, 493
MSK_INTPNT_HOTSTART_DUAL, 494
iomode, 515
MSK_IOMODE_WRITE, 515
MSK_IOMODE_READWRITE, 515
MSK_IOMODE_READ, 515
iparam, 420
liinfitem, 507
MSK_LIINF_SIMPLEX_ITER, 508
MSK_LIINF_RD_NUMQNZ, 508
MSK_LIINF_RD_NUMDJC, 508
MSK_LIINF_RD_NUMANZ, 508
MSK_LIINF_RD_NUMACC, 508
MSK_LIINF_MIO_SIMPLEX_ITER, 508
MSK_LIINF_MIO_PRESOLVED_ANZ, 508
MSK_LIINF_MIO_NUM_PRIM_ILLPOSED_CER, 508
MSK_LIINF_MIO_NUM_DUAL_ILLPOSED_CER, 508
MSK_LIINF_MIO_INTPNT_ITER, 508
MSK_LIINF_MIO_FINAL_ANZ, 508
MSK_LIINF_MIO_ANZ, 508
MSK_LIINF_INTPNT_FACTOR_NUM_NZ, 508
MSK_LIINF_FOLDING_BI_PRIMAL_ITER, 508
MSK_LIINF_FOLDING_BI_OPTIMIZER_ITER, 508
MSK_LIINF_FOLDING_BI_DUAL_ITER, 508
MSK_LIINF_BI_PRIMAL_ITER, 508
MSK_LIINF_BI_DUAL_ITER, 508
MSK_LIINF_BI_CLEAN_ITER, 508
MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_ROWS, 507
MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_NZ, 507
MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_COLUMNS, 507
mark, 492
MSK_MARK_UP, 492
MSK_MARK_LO, 492
miocontsoltype, 516
MSK_MIO_CONT_SOL_ROOT, 516
MSK_MIO_CONT_SOL_NONE, 516
MSK_MIO_CONT_SOL_ITG_REL, 516
MSK_MIO_CONT_SOL_ITG, 516
miodatapermmethod, 516
MSK_MIO_DATA_PERMUTATION_METHOD_RANDOM, 516
MSK_MIO_DATA_PERMUTATION_METHOD_NONE, 516
MSK_MIO_DATA_PERMUTATION_METHOD_CYCLIC_SHIFT, 516
miomode, 516
MSK_MIO_MODE_SATISFIED, 516
MSK_MIO_MODE_IGNORED, 516
mionodeseltype, 517
MSK_MIO_NODE_SELECTION_PSEUDO, 517
MSK_MIO_NODE_SELECTION_FREE, 517
MSK_MIO_NODE_SELECTION_FIRST, 517
MSK_MIO_NODE_SELECTION_BEST, 517
miovarseltype, 517
MSK_MIO_VAR_SELECTION_STRONG, 517
MSK_MIO_VAR_SELECTION_PSEUDOCOST, 517
MSK_MIO_VAR_SELECTION_FREE, 517
miqcqoreformmethod, 516
MSK_MIO_QCQO_REFORMULATION_METHOD_RELAX_SDP, 516
MSK_MIO_QCQO_REFORMULATION_METHOD_NONE, 516
MSK_MIO_QCQO_REFORMULATION_METHOD_LINEARIZATION, 516
MSK_MIO_QCQO_REFORMULATION_METHOD_FREE, 516
MSK_MIO_QCQO_REFORMULATION_METHOD_EIGEN_VAL_METHOD, 516
MSK_MIO_QCQO_REFORMULATION_METHOD_DIAG_SDP, 516
mpsformat, 517
MSK_MPS_FORMAT_STRICT, 517
MSK_MPS_FORMAT_RELAXED, 517
MSK_MPS_FORMAT_FREE, 517
MSK_MPS_FORMAT_CPLEX, 517
nametype, 500
MSK_NAME_TYPE_MPS, 500
MSK_NAME_TYPE_LP, 500
MSK_NAME_TYPE_GEN, 500
objsense, 517
MSK_OBJECTIVE_SENSE_MINIMIZE, 517
MSK_OBJECTIVE_SENSE_MAXIMIZE, 517
onoffkey, 517
MSK_ON, 517
MSK_OFF, 517
optimizertype, 517
MSK_OPTIMIZER_PRIMAL_SIMPLEX, 518
MSK_OPTIMIZER_NEW_PRIMAL_SIMPLEX, 518
MSK_OPTIMIZER_NEW_DUAL_SIMPLEX, 518
MSK_OPTIMIZER_MIXED_INT, 518
MSK_OPTIMIZER_INTPNT, 518
MSK_OPTIMIZER_FREE_SIMPLEX, 518
MSK_OPTIMIZER_FREE, 518
MSK_OPTIMIZER_DUAL_SIMPLEX, 517
MSK_OPTIMIZER_CONIC, 517
orderingtype, 518
MSK_ORDER_METHOD_TRY_GRAPHPAR, 518
MSK_ORDER_METHOD_NONE, 518
MSK_ORDER_METHOD_FREE, 518
MSK_ORDER_METHOD_FORCE_GRAPHPAR, 518
MSK_ORDER_METHOD_EXPERIMENTAL, 518
MSK_ORDER_METHOD_APPMINLOC, 518
parametertype, 519

MSK_PAR_STR_TYPE, 519
MSK_PAR_INVALID_TYPE, 519
MSK_PAR_INT_TYPE, 519
MSK_PAR_DOU_TYPE, 519
presolvemode, 518
MSK_PRESOLVE_MODE_ON, 518
MSK_PRESOLVE_MODE_OFF, 518
MSK_PRESOLVE_MODE_FREE, 518
problemitem, 519
MSK_PI_VAR, 519
MSK_PI_CONE, 519
MSK_PI_CON, 519
problemtype, 519
MSK_PROBTYPE_QO, 519
MSK_PROBTYPE_QCQO, 519
MSK_PROBTYPE_MIXED, 519
MSK_PROBTYPE_LO, 519
MSK_PROBTYPE_CONIC, 519
prosta, 519
MSK_PRO_STA_UNKNOWN, 519
MSK_PRO_STA_PRIM_INFEAS_OR_UNBOUNDED, 520
MSK_PRO_STA_PRIM_INFEAS, 519
MSK_PRO_STA_PRIM_FEAS, 519
MSK_PRO_STA_PRIM_AND_DUAL_INFEAS, 520
MSK_PRO_STA_PRIM_AND_DUAL_FEAS, 519
MSK_PRO_STA_ILL_POSED, 520
MSK_PRO_STA_DUAL_INFEAS, 520
MSK_PRO_STA_DUAL_FEAS, 519
rescode, 468
rescodetype, 520
MSK_RESPONSE_WRN, 520
MSK_RESPONSE_UNK, 520
MSK_RESPONSE_TRM, 520
MSK_RESPONSE_OK, 520
MSK_RESPONSE_ERR, 520
scalingmethod, 520
MSK_SCALING_METHOD_POW2, 520
MSK_SCALING_METHOD_FREE, 520
scalingtype, 520
MSK_SCALING_NONE, 520
MSK_SCALING_FREE, 520
sensitivitytype, 520
MSK_SENSITIVITY_TYPE_BASIS, 520
simdegen, 492
MSK_SIM_DEGEN_NONE, 492
MSK_SIM_DEGEN_MODERATE, 493
MSK_SIM_DEGEN_MINIMUM, 493
MSK_SIM_DEGEN_FREE, 492
MSK_SIM_DEGEN_AGGRESSIVE, 492
simdupvec, 493
MSK_SIM_EXPLOIT_DUPVEC_ON, 493
MSK_SIM_EXPLOIT_DUPVEC_OFF, 493
MSK_SIM_EXPLOIT_DUPVEC_FREE, 493
simhotstart, 493
MSK_SIM_HOTSTART_STATUS_KEYS, 493
MSK_SIM_HOTSTART_NONE, 493
MSK_SIM_HOTSTART_FREE, 493
simprecision, 492
MSK_SIM_PRECISION_NORMAL, 492
MSK_SIM_PRECISION_EXTENDED, 492
simreform, 493
MSK_SIM_REFORMULATION_ON, 493
MSK_SIM_REFORMULATION_OFF, 493
MSK_SIM_REFORMULATION_FREE, 493
MSK_SIM_REFORMULATION_AGGRESSIVE, 493
simseltype, 520
MSK_SIM_SELECTION_SE, 521
MSK_SIM_SELECTION_PARTIAL, 521
MSK_SIM_SELECTION_FULL, 520
MSK_SIM_SELECTION_FREE, 520
MSK_SIM_SELECTION_DEVEX, 521
MSK_SIM_SELECTION_ASE, 520
solformat, 501
MSK_SOL_FORMAT_TASK, 501
MSK_SOL_FORMAT_JSON_TASK, 501
MSK_SOL_FORMAT_EXTENSION, 501
MSK_SOL_FORMAT_B, 501
solitem, 521
MSK_SOL_ITEM_Y, 521
MSK_SOL_ITEM_XX, 521
MSK_SOL_ITEM_XC, 521
MSK_SOL_ITEM_SUX, 521
MSK_SOL_ITEM_SUC, 521
MSK_SOL_ITEM_SNX, 521
MSK_SOL_ITEM_SLX, 521
MSK_SOL_ITEM_SLC, 521
solsta, 521
MSK_SOL_STA_UNKNOWN, 521
MSK_SOL_STA_PRIM_INFEAS_CER, 521
MSK_SOL_STA_PRIM_ILLPOSED_CER, 521
MSK_SOL_STA_PRIM_FEAS, 521
MSK_SOL_STA_PRIM_AND_DUAL_FEAS, 521
MSK_SOL_STA_OPTIMAL, 521
MSK_SOL_STA_INTEGER_OPTIMAL, 522
MSK_SOL_STA_DUAL_INFEAS_CER, 521
MSK_SOL_STA_DUAL_ILLPOSED_CER, 521
MSK_SOL_STA_DUAL_FEAS, 521
soltype, 522
MSK_SOL_ITR, 522
MSK_SOL_ITG, 522
MSK_SOL_BAS, 522
solveform, 522
MSK_SOLVE_PRIMAL, 522
MSK_SOLVE_FREE, 522
MSK_SOLVE_DUAL, 522
sparam, 464
stakey, 522
MSK_SK_UPR, 522
MSK_SK_UNK, 522
MSK_SK_SUPBAS, 522
MSK_SK_LOW, 522
MSK_SK_INF, 522
MSK_SK_FIX, 522
MSK_SK_BAS, 522
startpointtype, 522
MSK_STARTING_POINT_GUESS, 522

MSK_STARTING_POINT_FREE, 522
MSK_STARTING_POINT_CONSTANT, 522
streamtype, 522
MSK_STREAM_WRN, 523
MSK_STREAM_MSG, 523
MSK_STREAM_LOG, 522
MSK_STREAM_ERR, 523
symmattype, 500
MSK_SYMMAT_TYPE_SPARSE, 500
transpose, 493
MSK_TRANSPOSE_YES, 493
MSK_TRANSPOSE_NO, 493
uplo, 493
MSK_UPLO_UP, 493
MSK_UPLO_LO, 493
value, 523
MSK_MAX_STR_LEN, 523
MSK_LICENSE_BUFFER_LENGTH, 523
variabletype, 523
MSK_VAR_TYPE_INT, 523
MSK_VAR_TYPE_CONT, 523

Functions

analyzenames, 219
analyzeproblem, 219
analyzesolution, 219
appendacc, 219
appendaccs, 220
appendaccseq, 221
appendaccsseq, 221
appendafes, 222
appendbarvars, 222
appendcone, 222
appendconeseq, 224
appendconesseq, 224
appendcons, 225
appenddjcs, 225
appenddualexpconedomain, 226
appenddualgeomeanconedomain, 226
appenddualpowerconedomain, 226
appenddualpowerconedomainseq, 227
appendprimalexpconedomain, 227
appendprimalgeomeanconedomain, 227
appendprimalpowerconedomain, 228
appendprimalpowerconedomainseq, 228
appendquadraticconedomain, 229
appendrdomain, 229
appendrminusdomain, 230
appendrplusdomain, 230
appendrquadraticconedomain, 230
appendrzerodomain, 230
appendsparsesymmat, 231
appendsparsesymmatlist, 232
appendsvecpsdconedomain, 232
appendvars, 233
asyncgetlog, 233
asyncgetresult, 234
asyncoptimize, 234
asyncpoll, 234
asyncstop, 235
basiscond, 235
bktostr, 236
callbackcodetostr, 236
checkinall, 236
checkinlicense, 236
checkmem, 237
checkoutlicense, 237
chgconbound, 237
chgvarbound, 238
clearcallbackfunc, 239
clearstreamfunc, 239
commitchanges, 239
computesparsecholesky, 239
conetypetostr, 241
deletesolution, 242
dinfitemtostr, 242
dualsensitivity, 242
echointro, 243
emptyafebarfrow, 243
emptyafebarfrowlist, 243
emptyafefcol, 243
emptyafefcollist, 244
emptyafefrow, 244
emptyafefrowlist, 244
evaluateacc, 245
evaluateaccs, 245
expirylicenses, 245
getaccafeidxlist, 245
getaccb, 246
getaccbarfblocktriplet, 246
getaccbarfnumblocktriplets, 247
getaccdomain, 247
getaccdoty, 247
getaccdotys, 247
getaccfnumnz, 248
getaccftrip, 248
getaccgvector, 248
getaccn, 249
getaccname, 249
getaccnamelen, 249
getaccntot, 249
getaccs, 250
getacol, 250
getacolnumnz, 250
getacolslice, 251
getacolslicenumnz, 251
getacolslicetrip, 252
getafebarfblocktriplet, 252
getafebarfnumblocktriplets, 252
getafebarfnumrowentries, 253
getafebarfrow, 253
getafebarfrowinfo, 254
getafefnumnz, 254
getafefrow, 254
getafefrownumnz, 255
getafeftrip, 255

getafeg, 255
getafegslice, 256
getaij, 256
getapiecenumnz, 256
getarow, 257
getarownumnz, 257
getarowslice, 258
getarowslicenumnz, 258
getarowslicetrip, 259
getatrip, 259
getatruncatetol, 260
getbarablocktriplet, 260
getbaraidx, 260
getbaraidxij, 261
getbaraidxinfo, 261
getbarasparsity, 262
getbarcblocktriplet, 262
getbarcidx, 262
getbarcidxinfo, 263
getbarcidxj, 263
getbarcsparsity, 263
getbarsj, 264
getbarsslice, 264
getbarvarname, 265
getbarvarnameindex, 265
getbarvarnamelen, 265
getbarxj, 265
getbarxslice, 266
getc, 266
getcfix, 267
getcj, 267
getclist, 267
getcodedesc, 268
getconbound, 268
getconboundslice, 268
getcone, 269
getconeinfo, 269
getconename, 270
getconenameindex, 270
getconenamelen, 270
getconname, 271
getconnameindex, 271
getconnamelen, 271
getcslice, 272
getdimbarvarj, 272
getdjcafeidxlist, 272
getdjcb, 273
getdjcdomainidxlist, 273
getdjcname, 273
getdjcnamelen, 274
getdjcnumafe, 274
getdjcnumafetot, 274
getdjcnumdomain, 275
getdjcnumdomaintot, 275
getdjcnumterm, 275
getdjcnumtermtot, 276
getdjcs, 276
getdjctermsizelist, 276
getdomainn, 276
getdomainname, 277
getdomainnamelen, 277
getdomaintype, 277
getdouinf, 278
getdouparam, 278
getdualobj, 278
getdualproblem, 279
getdualsolutionnorms, 279
getdviolacc, 279
getdviolbarvar, 280
getdviolcon, 280
getdviolcones, 281
getdviolvar, 282
getinfeasiblesubproblem, 282
getinfname, 283
getintinf, 283
getintparam, 283
getlasterror, 283
getlenbarvarj, 284
getlintinf, 284
getlintparam, 284
getmaxnumanz, 285
getmaxnumbarvar, 285
getmaxnumcon, 285
getmaxnumcone, 285
getmaxnumqnz, 286
getmaxnumvar, 286
getmemusage, 286
getnadouinf, 286
getnadouparam, 287
getnaintinf, 287
getnaintparam, 287
getnastrparam, 287
getnumacc, 288
getnumafe, 288
getnumanz, 288
getnumbarablocktriplets, 289
getnumbaranz, 289
getnumbarcblocktriplets, 289
getnumbarcnz, 289
getnumbarvar, 289
getnumcon, 290
getnumcone, 290
getnumconemem, 290
getnumdjc, 290
getnumdomain, 291
getnumintvar, 291
getnumparam, 291
getnumqconknz, 291
getnumqobjnz, 292
getnumsymmat, 292
getnumvar, 292
getobjname, 292
getobjnamelen, 292
getobjsense, 293
getparamname, 293
getpowerdomainalpha, 293

getpowerdomaininfo, 294
getprimalobj, 294
getprimalsolutionnorms, 294
getprobtype, 295
getprosta, 295
getpviolacc, 295
getpviolbarvar, 295
getpviolcon, 296
getpviolcones, 297
getpvioldjc, 297
getpviolvar, 298
getqconk, 298
getqobj, 299
getqobjij, 299
getreducedcosts, 299
getskc, 300
getskcslice, 300
getskn, 300
getskx, 301
getskxslice, 301
getslc, 301
getslcslice, 302
getslx, 302
getslxslice, 302
getsnx, 303
getsnxslice, 303
getsolsta, 304
getsolution, 304
getsolutioninfo, 305
getsolutioninfonew, 306
getsolutionnew, 307
getsolutionslice, 307
getsparsesymmat, 308
getstrparam, 308
getstrparamlen, 309
getsuc, 309
getsucslice, 309
getsux, 310
getsuxslice, 310
getsymmatinfo, 310
gettaskname, 311
gettasknamelen, 311
getvarbound, 311
getvarboundslice, 312
getvarname, 312
getvarnameindex, 312
getvarnamelen, 313
getvartype, 313
getvartypelist, 313
getversion, 314
getxc, 314
getxcslice, 314
getxx, 315
getxxslice, 315
gety, 315
getyslice, 316
iinfitemtostr, 316
infeasibilityreport, 316
initbasissolve, 317
inputdata, 317
isdouparname, 319
isintparname, 319
isstrparname, 319
licensecleanup, 319
liinfitemtostr, 320
linkfiletostream, 320
linkfiletostream, 320
makeenv, 321
maketask, 321
onesolutionsummary, 321
optimize, 321
optimizebatch, 322
optimizermt, 323
optimizersummary, 323
primalrepair, 323
primalsensitivity, 324
printparam, 325
probtypetostr, 326
prostatostr, 326
putacc, 326
putaccb, 327
putaccbj, 327
putaccdoty, 327
putacclist, 328
putaccname, 328
putacol, 329
putacollist, 329
putacolslice, 330
putafebarfblocktriplet, 331
putafebarfentry, 331
putafebarfentrylist, 332
putafebarfrow, 333
putafefcol, 334
putafefentry, 334
putafefentrylist, 335
putafefrow, 335
putafefrowlist, 336
putafeg, 336
putafeglist, 337
putafegslice, 337
putaij, 338
putaijlist, 338
putarow, 339
putarowlist, 339
putarowslice, 340
putatruncatetol, 341
putbarablocktriplet, 341
putbaraij, 342
putbaraijlist, 342
putbararowlist, 343
putbarcblocktriplet, 344
putbarcj, 345
putbarsj, 345
putbarvarname, 345
putbarxj, 346
putcallbackfunc, 346

putcfix, 346
putcj, 347
putclist, 347
putconbound, 348
putconboundlist, 348
putconboundlistconst, 349
putconboundslice, 349
putconboundsliceconst, 350
putcone, 350
putconename, 351
putconname, 351
putconsolutioni, 351
putcslice, 352
putdjc, 352
putdjcname, 353
putdjcslice, 354
putdomainname, 355
putdouparam, 355
putintparam, 355
putlicensecode, 356
putlicensedebug, 356
putlicensepath, 356
putlicensewait, 356
putlintparam, 357
putmaxnumacc, 357
putmaxnumafe, 357
putmaxnumanz, 358
putmaxnumbarvar, 358
putmaxnumcon, 359
putmaxnumcone, 359
putmaxnumdjc, 359
putmaxnumdomain, 360
putmaxnumqnz, 360
putmaxnumvar, 361
putnadouparam, 361
putnaintparam, 361
putnastrparam, 362
putobjname, 362
putobjsense, 362
putoptserverhost, 362
putparam, 363
putqcon, 363
putqconk, 364
putqobj, 364
putqobjij, 365
putskc, 366
putskcslice, 366
putskx, 366
putskxslice, 367
putslc, 367
putslcslice, 367
putslx, 368
putslxslice, 368
putsnx, 369
putsnxslice, 369
putsolution, 369
putsolutionnew, 371
putsolutionyi, 372
putstreamfunc, 372
putstrparam, 372
putsuc, 373
putsucslice, 373
putsux, 374
putsuxslice, 374
puttaskname, 374
putvarbound, 375
putvarboundlist, 375
putvarboundlistconst, 376
putvarboundslice, 376
putvarboundsliceconst, 377
putvarname, 377
putvarsolutionj, 377
putvartype, 378
putvartypelist, 379
putxc, 379
putxcslice, 379
putxx, 380
putxxslice, 380
puty, 380
putyslice, 381
readbsolution, 381
readdata, 382
readdataformat, 382
readjsonsol, 382
readjsonstring, 382
readlpstring, 382
readopfstring, 383
readparamfile, 383
readptfstring, 383
readsolution, 384
readsolutionfile, 384
readsummary, 384
readtask, 384
removebarvars, 385
removecones, 385
removecons, 385
removevars, 386
rescodetostr, 386
resetdouparam, 386
resetexpirylicenses, 386
resetintparam, 386
resetparameters, 387
resetstrparam, 387
resizetask, 387
sensitivityreport, 388
solutiondef, 388
solutionsummary, 388
solvewithbasis, 388
sparsetriangularsolvedense, 389
strtoconetype, 391
strtosk, 391
updatesolutioninfo, 391
writebsolution, 391
writedata, 392
writedatastream, 392
writejsonsol, 392

writeparamfile, 393
writesolution, 393
writesolutionfile, 393
writetask, 393

Parameters

Double parameters, 406
MSK_DPAR_ANA_SOL_INFEAS_TOL, 406
MSK_DPAR_BASIS_REL_TOL_S, 406
MSK_DPAR_BASIS_TOL_S, 406
MSK_DPAR_BASIS_TOL_X, 406
MSK_DPAR_DATA_SYM_MAT_TOL, 407
MSK_DPAR_DATA_SYM_MAT_TOL_HUGE, 407
MSK_DPAR_DATA_SYM_MAT_TOL_LARGE, 407
MSK_DPAR_DATA_TOL_AIJ_HUGE, 407
MSK_DPAR_DATA_TOL_AIJ_LARGE, 407
MSK_DPAR_DATA_TOL_BOUND_INF, 408
MSK_DPAR_DATA_TOL_BOUND_WRN, 408
MSK_DPAR_DATA_TOL_C_HUGE, 408
MSK_DPAR_DATA_TOL_CJ_LARGE, 408
MSK_DPAR_DATA_TOL_QIJ, 408
MSK_DPAR_DATA_TOL_X, 409
MSK_DPAR_FOLDING_TOL_EQ, 409
MSK_DPAR_INTPNT_CO_TOL_DFEAS, 409
MSK_DPAR_INTPNT_CO_TOL_INFEAS, 409
MSK_DPAR_INTPNT_CO_TOL_MU_RED, 410
MSK_DPAR_INTPNT_CO_TOL_NEAR_REL, 410
MSK_DPAR_INTPNT_CO_TOL_PFEAS, 410
MSK_DPAR_INTPNT_CO_TOL_REL_GAP, 410
MSK_DPAR_INTPNT_QO_TOL_DFEAS, 411
MSK_DPAR_INTPNT_QO_TOL_INFEAS, 411
MSK_DPAR_INTPNT_QO_TOL_MU_RED, 411
MSK_DPAR_INTPNT_QO_TOL_NEAR_REL, 411
MSK_DPAR_INTPNT_QO_TOL_PFEAS, 411
MSK_DPAR_INTPNT_QO_TOL_REL_GAP, 412
MSK_DPAR_INTPNT_TOL_DFEAS, 412
MSK_DPAR_INTPNT_TOL_DSAFE, 412
MSK_DPAR_INTPNT_TOL_INFEAS, 412
MSK_DPAR_INTPNT_TOL_MU_RED, 413
MSK_DPAR_INTPNT_TOL_PATH, 413
MSK_DPAR_INTPNT_TOL_PFEAS, 413
MSK_DPAR_INTPNT_TOL_PSAFE, 413
MSK_DPAR_INTPNT_TOL_REL_GAP, 414
MSK_DPAR_INTPNT_TOL_REL_STEP, 414
MSK_DPAR_INTPNT_TOL_STEP_SIZE, 414
MSK_DPAR_LOWER_OBJ_CUT, 414
MSK_DPAR_LOWER_OBJ_CUT_FINITE_TRH, 414
MSK_DPAR_MIO_CLIQUE_TABLE_SIZE_FACTOR, 415
MSK_DPAR_MIO_DJC_MAX_BIGM, 415
MSK_DPAR_MIO_MAX_TIME, 415
MSK_DPAR_MIO_REL_GAP_CONST, 415
MSK_DPAR_MIO_TOL_ABS_GAP, 416
MSK_DPAR_MIO_TOL_ABS_RELAX_INT, 416
MSK_DPAR_MIO_TOL_FEAS, 416
MSK_DPAR_MIO_TOL_REL_DUAL_BOUND_IMPROVEMENT, 416
MSK_DPAR_MIO_TOL_REL_GAP, 416
MSK_DPAR_OPTIMIZER_MAX_TICKS, 417
MSK_DPAR_OPTIMIZER_MAX_TIME, 417
MSK_DPAR_PRESOLVE_TOL_ABS_LINDEP, 417
MSK_DPAR_PRESOLVE_TOL_PRIMAL_INFEAS_PERTURBATION, 417
MSK_DPAR_PRESOLVE_TOL_REL_LINDEP, 418
MSK_DPAR_PRESOLVE_TOL_S, 418
MSK_DPAR_PRESOLVE_TOL_X, 418
MSK_DPAR_QCQO_REFORMULATE_REL_DROP_TOL, 418
MSK_DPAR_SEMIDEFINITE_TOL_APPROX, 418
MSK_DPAR_SIM_LU_TOL_REL_PIV, 419
MSK_DPAR_SIM_PRECISION_SCALING_EXTENDED, 419
MSK_DPAR_SIM_PRECISION_SCALING_NORMAL, 419
MSK_DPAR_SIMPLEX_ABS_TOL_PIV, 419
MSK_DPAR_UPPER_OBJ_CUT, 420
MSK_DPAR_UPPER_OBJ_CUT_FINITE_TRH, 420
Integer parameters, 420
MSK_IPAR_ANA_SOL_BASIS, 420
MSK_IPAR_ANA_SOL_PRINT_VIOLATED, 420
MSK_IPAR_AUTO_SORT_A_BEFORE_OPT, 421
MSK_IPAR_AUTO_UPDATE_SOL_INFO, 421
MSK_IPAR_BASIS_SOLVE_USE_PLUS_ONE, 421
MSK_IPAR_BI_CLEAN_OPTIMIZER, 421
MSK_IPAR_BI_IGNORE_MAX_ITER, 422
MSK_IPAR_BI_IGNORE_NUM_ERROR, 422
MSK_IPAR_BI_MAX_ITERATIONS, 422
MSK_IPAR_CACHE_LICENSE, 422
MSK_IPAR_COMPRESS_STATFILE, 423
MSK_IPAR_FOLDING_USE, 423
MSK_IPAR_GETDUAL_CONVERT_LMIS, 423
MSK_IPAR_HEARTBEAT_SIM_FREQ_TICKS, 423
MSK_IPAR_INFEAS_GENERIC_NAMES, 423
MSK_IPAR_INFEAS_REPORT_AUTO, 424
MSK_IPAR_INFEAS_REPORT_LEVEL, 424
MSK_IPAR_INTPNT_BASIS, 424
MSK_IPAR_INTPNT_DIFF_STEP, 424
MSK_IPAR_INTPNT_HOTSTART, 425
MSK_IPAR_INTPNT_MAX_ITERATIONS, 425
MSK_IPAR_INTPNT_MAX_NUM_COR, 425
MSK_IPAR_INTPNT_OFF_COL_TRH, 425
MSK_IPAR_INTPNT_ORDER_GP_NUM_SEEDS, 426
MSK_IPAR_INTPNT_ORDER_METHOD, 426
MSK_IPAR_INTPNT_REGULARIZATION_USE, 426
MSK_IPAR_INTPNT_SCALING, 426
MSK_IPAR_INTPNT_SOLVE_FORM, 426
MSK_IPAR_INTPNT_STARTING_POINT, 427
MSK_IPAR_LICENSE_DEBUG, 427
MSK_IPAR_LICENSE_PAUSE_TIME, 427
MSK_IPAR_LICENSE_SUPPRESS_EXPIRE_WRNS, 427
MSK_IPAR_LICENSE_TRH_EXPIRY_WRN, 428
MSK_IPAR_LICENSE_WAIT, 428
MSK_IPAR_LOG, 428
MSK_IPAR_LOG_ANA_PRO, 428
MSK_IPAR_LOG_BI, 429
MSK_IPAR_LOG_BI_FREQ, 429
MSK_IPAR_LOG_CUT_SECOND_OPT, 429
MSK_IPAR_LOG_EXPAND, 429
MSK_IPAR_LOG_FEAS_REPAIR, 429

MSK_IPAR_LOG_FILE, 430
MSK_IPAR_LOG_INCLUDE_SUMMARY, 430
MSK_IPAR_LOG_INFEAS_ANA, 430
MSK_IPAR_LOG_INTPNT, 430
MSK_IPAR_LOG_LOCAL_INFO, 431
MSK_IPAR_LOG_MIO, 431
MSK_IPAR_LOG_MIO_FREQ, 431
MSK_IPAR_LOG_ORDER, 431
MSK_IPAR_LOG_PRESOLVE, 431
MSK_IPAR_LOG_SENSITIVITY, 432
MSK_IPAR_LOG_SENSITIVITY_OPT, 432
MSK_IPAR_LOG_SIM, 432
MSK_IPAR_LOG_SIM_FREQ, 432
MSK_IPAR_LOG_SIM_FREQ_GIGA_TICKS, 433
MSK_IPAR_LOG_STORAGE, 433
MSK_IPAR_MAX_NUM_WARNINGS, 433
MSK_IPAR_MIO_BRANCH_DIR, 433
MSK_IPAR_MIO_CONFLICT_ANALYSIS_LEVEL, 434
MSK_IPAR_MIO_CONIC_OUTER_APPROXIMATION, 434
MSK_IPAR_MIO_CONSTRUCT_SOL, 434
MSK_IPAR_MIO_CROSSOVER_MAX_NODES, 434
MSK_IPAR_MIO_CUT_CLIQUE, 435
MSK_IPAR_MIO_CUT_CMIR, 435
MSK_IPAR_MIO_CUT_GMI, 435
MSK_IPAR_MIO_CUT_IMPLIED_BOUND, 435
MSK_IPAR_MIO_CUT_KNAPSACK_COVER, 436
MSK_IPAR_MIO_CUT_LIPRO, 436
MSK_IPAR_MIO_CUT_SELECTION_LEVEL, 436
MSK_IPAR_MIO_DATA_PERMUTATION_METHOD, 436
MSK_IPAR_MIO_DUAL_RAY_ANALYSIS_LEVEL, 436
MSK_IPAR_MIO_FEASPUMP_LEVEL, 437
MSK_IPAR_MIO_HEURISTIC_LEVEL, 437
MSK_IPAR_MIO_INDEPENDENT_BLOCK_LEVEL, 437
MSK_IPAR_MIO_MAX_NUM_BRANCHES, 438
MSK_IPAR_MIO_MAX_NUM_RELAXS, 438
MSK_IPAR_MIO_MAX_NUM_RESTARTS, 438
MSK_IPAR_MIO_MAX_NUM_ROOT_CUT_ROUNDS, 438
MSK_IPAR_MIO_MAX_NUM_SOLUTIONS, 439
MSK_IPAR_MIO_MEMORY_EMPHASIS_LEVEL, 439
MSK_IPAR_MIO_MIN_REL, 439
MSK_IPAR_MIO_MODE, 439
MSK_IPAR_MIO_NODE_OPTIMIZER, 440
MSK_IPAR_MIO_NODE_SELECTION, 440
MSK_IPAR_MIO_NUMERICAL_EMPHASIS_LEVEL, 440
MSK_IPAR_MIO_OPT_FACE_MAX_NODES, 440
MSK_IPAR_MIO_PERSPECTIVE_REFORMULATE, 441
MSK_IPAR_MIO_PRESOLVE_AGGREGATOR_USE, 441
MSK_IPAR_MIO_PROBING_LEVEL, 441
MSK_IPAR_MIO_PROPAGATE_OBJECTIVE_CONSTRAINT, 441
MSK_IPAR_MIO_QCQO_REFORMULATION_METHOD, 441
MSK_IPAR_MIO_RENS_MAX_NODES, 442
MSK_IPAR_MIO_RINS_MAX_NODES, 442
MSK_IPAR_MIO_ROOT_OPTIMIZER, 442
MSK_IPAR_MIO_SEED, 442
MSK_IPAR_MIO_SYMMETRY_LEVEL, 443
MSK_IPAR_MIO_VAR_SELECTION, 443
MSK_IPAR_MIO_VB_DETECTION_LEVEL, 443
MSK_IPAR_MT_SPINCOUNT, 443
MSK_IPAR_NG, 444
MSK_IPAR_NUM_THREADS, 444
MSK_IPAR_OPF_WRITE_HEADER, 444
MSK_IPAR_OPF_WRITE_HINTS, 444
MSK_IPAR_OPF_WRITE_LINE_LENGTH, 444
MSK_IPAR_OPF_WRITE_PARAMETERS, 445
MSK_IPAR_OPF_WRITE_PROBLEM, 445
MSK_IPAR_OPF_WRITE_SOL_BAS, 445
MSK_IPAR_OPF_WRITE_SOL_ITG, 445
MSK_IPAR_OPF_WRITE_SOL_ITR, 446
MSK_IPAR_OPF_WRITE_SOLUTIONS, 446
MSK_IPAR_OPTIMIZER, 446
MSK_IPAR_PARAM_READ_CASE_NAME, 446
MSK_IPAR_PARAM_READ_IGN_ERROR, 446
MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_FILL, 447
MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_NUM_TRIES, 447
MSK_IPAR_PRESOLVE_LINDEP_ABS_WORK_TRH, 447
MSK_IPAR_PRESOLVE_LINDEP_NEW, 447
MSK_IPAR_PRESOLVE_LINDEP_REL_WORK_TRH, 448
MSK_IPAR_PRESOLVE_LINDEP_USE, 448
MSK_IPAR_PRESOLVE_MAX_NUM_PASS, 448
MSK_IPAR_PRESOLVE_MAX_NUM_REDUCTIONS, 448
MSK_IPAR_PRESOLVE_USE, 448
MSK_IPAR_PRIMAL_REPAIR_OPTIMIZER, 449
MSK_IPAR_PTF_WRITE_PARAMETERS, 449
MSK_IPAR_PTF_WRITE_SINGLE_PSD_TERMS, 449
MSK_IPAR_PTF_WRITE_SOLUTIONS, 449
MSK_IPAR_PTF_WRITE_TRANSFORM, 450
MSK_IPAR_READ_ASYNC, 450
MSK_IPAR_READ_DEBUG, 450
MSK_IPAR_READ_KEEP_FREE_CON, 450
MSK_IPAR_READ_MPS_FORMAT, 450
MSK_IPAR_READ_MPS_WIDTH, 451
MSK_IPAR_READ_TASK_IGNORE_PARAM, 451
MSK_IPAR_REMOTE_USE_COMPRESSION, 451
MSK_IPAR_REMOVE_UNUSED_SOLUTIONS, 451
MSK_IPAR_SENSITIVITY_ALL, 451
MSK_IPAR_SENSITIVITY_TYPE, 452
MSK_IPAR_SIM_BASIS_FACTOR_USE, 452
MSK_IPAR_SIM_DEGEN, 452
MSK_IPAR_SIM_DETECT_PWL, 452
MSK_IPAR_SIM_DUAL_CRASH, 453
MSK_IPAR_SIM_DUAL_PHASEONE_METHOD, 453
MSK_IPAR_SIM_DUAL_RESTRICT_SELECTION, 453
MSK_IPAR_SIM_DUAL_SELECTION, 453
MSK_IPAR_SIM_EXPLOIT_DUPVEC, 454
MSK_IPAR_SIM_HOTSTART, 454
MSK_IPAR_SIM_HOTSTART_LU, 454
MSK_IPAR_SIM_MAX_ITERATIONS, 454
MSK_IPAR_SIM_MAX_NUM_SETBACKS, 454
MSK_IPAR_SIM_NON_SINGULAR, 455
MSK_IPAR_SIM_PRECISION, 455
MSK_IPAR_SIM_PRECISION_BOOST, 455
MSK_IPAR_SIM_PRIMAL_CRASH, 455
MSK_IPAR_SIM_PRIMAL_PHASEONE_METHOD, 456
MSK_IPAR_SIM_PRIMAL_RESTRICT_SELECTION, 456

MSK_IPAR_SIM_PRIMAL_SELECTION, 456
MSK_IPAR_SIM_REFACTOR_FREQ, 456
MSK_IPAR_SIM_REFORMULATION, 457
MSK_IPAR_SIM_SAVE_LU, 457
MSK_IPAR_SIM_SCALING, 457
MSK_IPAR_SIM_SCALING_METHOD, 457
MSK_IPAR_SIM_SEED, 457
MSK_IPAR_SIM_SOLVE_FORM, 458
MSK_IPAR_SIM_SWITCH_OPTIMIZER, 458
MSK_IPAR_SOL_FILTER_KEEP_BASIC, 458
MSK_IPAR_SOL_READ_NAME_WIDTH, 458
MSK_IPAR_SOL_READ_WIDTH, 459
MSK_IPAR_TIMING_LEVEL, 459
MSK_IPAR_WRITE_ASYNC, 459
MSK_IPAR_WRITE_BAS_CONSTRAINTS, 459
MSK_IPAR_WRITE_BAS_HEAD, 459
MSK_IPAR_WRITE_BAS_VARIABLES, 460
MSK_IPAR_WRITE_COMPRESSION, 460
MSK_IPAR_WRITE_FREE_CON, 460
MSK_IPAR_WRITE_GENERIC_NAMES, 460
MSK_IPAR_WRITE_IGNORE_INCOMPATIBLE_ITEMS, 460
MSK_IPAR_WRITE_INT_CONSTRAINTS, 461
MSK_IPAR_WRITE_INT_HEAD, 461
MSK_IPAR_WRITE_INT_VARIABLES, 461
MSK_IPAR_WRITE_JSON_INDENTATION, 461
MSK_IPAR_WRITE_LP_FULL_OBJ, 462
MSK_IPAR_WRITE_LP_LINE_WIDTH, 462
MSK_IPAR_WRITE_MPS_FORMAT, 462
MSK_IPAR_WRITE_MPS_INT, 462
MSK_IPAR_WRITE_SOL_BARVARIABLES, 462
MSK_IPAR_WRITE_SOL_CONSTRAINTS, 463
MSK_IPAR_WRITE_SOL_HEAD, 463
MSK_IPAR_WRITE_SOL_IGNORE_INVALID_NAMES, 463
MSK_IPAR_WRITE_SOL_VARIABLES, 463
String parameters, 464
MSK_SPAR_BAS_SOL_FILE_NAME, 464
MSK_SPAR_DATA_FILE_NAME, 464
MSK_SPAR_DEBUG_FILE_NAME, 464
MSK_SPAR_INT_SOL_FILE_NAME, 464
MSK_SPAR_ITR_SOL_FILE_NAME, 464
MSK_SPAR_MIO_DEBUG_STRING, 464
MSK_SPAR_PARAM_COMMENT_SIGN, 465
MSK_SPAR_PARAM_READ_FILE_NAME, 465
MSK_SPAR_PARAM_WRITE_FILE_NAME, 465
MSK_SPAR_READ_MPS_BOU_NAME, 465
MSK_SPAR_READ_MPS_OBJ_NAME, 465
MSK_SPAR_READ_MPS_RAN_NAME, 465
MSK_SPAR_READ_MPS_RHS_NAME, 466
MSK_SPAR_REMOTE_OPTSERVER_HOST, 466
MSK_SPAR_REMOTE_TLS_CERT, 466
MSK_SPAR_REMOTE_TLS_CERT_PATH, 466
MSK_SPAR_SENSITIVITY_FILE_NAME, 466
MSK_SPAR_SENSITIVITY_RES_FILE_NAME, 467
MSK_SPAR_SOL_FILTER_XC_LOW, 467
MSK_SPAR_SOL_FILTER_XC_UPR, 467
MSK_SPAR_SOL_FILTER_XX_LOW, 467
MSK_SPAR_SOL_FILTER_XX_UPR, 467
MSK_SPAR_STAT_KEY, 468
MSK_SPAR_STAT_NAME, 468

Response codes

Termination, 468
MSK_RES_OK, 468
MSK_RES_TRM_INTERNAL, 469
MSK_RES_TRM_INTERNAL_STOP, 469
MSK_RES_TRM_LOST_RACE, 469
MSK_RES_TRM_MAX_ITERATIONS, 468
MSK_RES_TRM_MAX_NUM_SETBACKS, 469
MSK_RES_TRM_MAX_TIME, 468
MSK_RES_TRM_MIO_NUM_BRANCHES, 468
MSK_RES_TRM_MIO_NUM_RELAXS, 468
MSK_RES_TRM_NUM_MAX_NUM_INT_SOLUTIONS, 468
MSK_RES_TRM_NUMERICAL_PROBLEM, 469
MSK_RES_TRM_OBJECTIVE_RANGE, 468
MSK_RES_TRM_SERVER_MAX_MEMORY, 469
MSK_RES_TRM_SERVER_MAX_TIME, 469
MSK_RES_TRM_STALL, 468
MSK_RES_TRM_USER_CALLBACK, 469
Warnings, 469
MSK_RES_WRN_ANA_ALMOST_INT_BOUNDS, 471
MSK_RES_WRN_ANA_C_ZERO, 471
MSK_RES_WRN_ANA_CLOSE_BOUNDS, 471
MSK_RES_WRN_ANA_EMPTY_COLS, 471
MSK_RES_WRN_ANA_LARGE_BOUNDS, 471
MSK_RES_WRN_DROPPED_NZ_QOBJ, 470
MSK_RES_WRN_DUPLICATE_BARVARIABLE_NAMES, 471
MSK_RES_WRN_DUPLICATE_CONE_NAMES, 471
MSK_RES_WRN_DUPLICATE_CONSTRAINT_NAMES, 471
MSK_RES_WRN_DUPLICATE_VARIABLE_NAMES, 471
MSK_RES_WRN_ELIMINATOR_SPACE, 471
MSK_RES_WRN_EMPTY_NAME, 470
MSK_RES_WRN_GETDUAL_IGNORES_INTEGRALITY, 472
MSK_RES_WRN_IGNORE_INTEGER, 470
MSK_RES_WRN_INCOMPLETE_LINEAR_DEPENDENCY_CHECK, 471
MSK_RES_WRN_INVALID_MPS_NAME, 470
MSK_RES_WRN_INVALID_MPS_OBJ_NAME, 470
MSK_RES_WRN_LARGE_AIJ, 469
MSK_RES_WRN_LARGE_BOUND, 469
MSK_RES_WRN_LARGE_CJ, 469
MSK_RES_WRN_LARGE_CON_FX, 469
MSK_RES_WRN_LARGE_FIJ, 472
MSK_RES_WRN_LARGE_LO_BOUND, 469
MSK_RES_WRN_LARGE_UP_BOUND, 469
MSK_RES_WRN_LICENSE_EXPIRE, 470
MSK_RES_WRN_LICENSE_FEATURE_EXPIRE, 470
MSK_RES_WRN_LICENSE_SERVER, 470
MSK_RES_WRN_LP_DROP_VARIABLE, 470
MSK_RES_WRN_LP_OLD_QUAD_FORMAT, 470
MSK_RES_WRN_MIO_INFEASIBLE_FINAL, 470
MSK_RES_WRN_MODIFIED_DOUBLE_PARAMETER, 472
MSK_RES_WRN_MPS_SPLIT_BOU_VECTOR, 470

MSK_RES_WRN_MPS_SPLIT_RAN_VECTOR, 470
MSK_RES_WRN_MPS_SPLIT_RHS_VECTOR, 470
MSK_RES_WRN_NAME_MAX_LEN, 469
MSK_RES_WRN_NO_DUALIZER, 472
MSK_RES_WRN_NO_GLOBAL_OPTIMIZER, 470
MSK_RES_WRN_NO_INFEASIBILITY_REPORT_WHEN_MATRIX_VARIABLES, 472
MSK_RES_WRN_NZ_IN_UPR_TRI, 470
MSK_RES_WRN_OPEN_PARAM_FILE, 469
MSK_RES_WRN_PARAM_IGNORED_CMIO, 471
MSK_RES_WRN_PARAM_NAME_DOU, 470
MSK_RES_WRN_PARAM_NAME_INT, 471
MSK_RES_WRN_PARAM_NAME_STR, 471
MSK_RES_WRN_PARAM_STR_VALUE, 471
MSK_RES_WRN_PRESOLVE_OUTOFSPACE, 471
MSK_RES_WRN_PRESOLVE_PRIMAL_PERTURBATIONS, 471
MSK_RES_WRN_PTF_UNKNOWN_SECTION, 472
MSK_RES_WRN_SOL_FILE_IGNORED_CON, 470
MSK_RES_WRN_SOL_FILE_IGNORED_VAR, 470
MSK_RES_WRN_SOL_FILTER, 470
MSK_RES_WRN_SPAR_MAX_LEN, 469
MSK_RES_WRN_SYM_MAT_LARGE, 472
MSK_RES_WRN_TOO_FEW_BASIS_VARS, 470
MSK_RES_WRN_TOO_MANY_BASIS_VARS, 470
MSK_RES_WRN_UNDEF_SOL_FILE_NAME, 470
MSK_RES_WRN_USING_GENERIC_NAMES, 470
MSK_RES_WRN_WRITE_CHANGED_NAMES, 471
MSK_RES_WRN_WRITE_DISCARDED_CFIX, 471
MSK_RES_WRN_ZERO_AIJ, 469
MSK_RES_WRN_ZEROS_IN_SPARSE_COL, 471
MSK_RES_WRN_ZEROS_IN_SPARSE_ROW, 471
Errors, 472
MSK_RES_ERR_ACC_AFE_DOMAIN_MISMATCH, 491
MSK_RES_ERR_ACC_INVALID_ENTRY_INDEX, 491
MSK_RES_ERR_ACC_INVALID_INDEX, 491
MSK_RES_ERR_AD_INVALID_CODELIST, 485
MSK_RES_ERR_AFE_INVALID_INDEX, 491
MSK_RES_ERR_API_ARRAY_TOO_SMALL, 485
MSK_RES_ERR_API_CB_CONNECT, 485
MSK_RES_ERR_API_FATAL_ERROR, 485
MSK_RES_ERR_API_INTERNAL, 485
MSK_RES_ERR_APPENDING_TOO_BIG_CONE, 481
MSK_RES_ERR_ARG_IS_TOO_LARGE, 479
MSK_RES_ERR_ARG_IS_TOO_SMALL, 479
MSK_RES_ERR_ARGUMENT_DIMENSION, 478
MSK_RES_ERR_ARGUMENT_IS_TOO_LARGE, 487
MSK_RES_ERR_ARGUMENT_IS_TOO_SMALL, 487
MSK_RES_ERR_ARGUMENT_LENNEQ, 478
MSK_RES_ERR_ARGUMENT_PERM_ARRAY, 481
MSK_RES_ERR_ARGUMENT_TYPE, 478
MSK_RES_ERR_AXIS_NAME_SPECIFICATION, 475
MSK_RES_ERR_BAR_VAR_DIM, 486
MSK_RES_ERR_BASIS, 480
MSK_RES_ERR_BASIS_FACTOR, 484
MSK_RES_ERR_BASIS_SINGULAR, 484
MSK_RES_ERR_BLANK_NAME, 474
MSK_RES_ERR_CBF_DUPLICATE_ACOORD, 488
MSK_RES_ERR_CBF_DUPLICATE_BCOORD, 488
MSK_RES_ERR_CBF_DUPLICATE_CON, 488
MSK_RES_ERR_CBF_DUPLICATE_INT, 488
MSK_RES_ERR_CBF_DUPLICATE_OBJ, 488
MSK_RES_ERR_CBF_DUPLICATE_OBJACOORD, 488
MSK_RES_ERR_CBF_DUPLICATE_POW_CONES, 488
MSK_RES_ERR_CBF_DUPLICATE_POW_STAR_CONES, 488
MSK_RES_ERR_CBF_DUPLICATE_PSDCON, 489
MSK_RES_ERR_CBF_DUPLICATE_PSDVAR, 488
MSK_RES_ERR_CBF_DUPLICATE_VAR, 488
MSK_RES_ERR_CBF_EXPECTED_A_KEYWORD, 489
MSK_RES_ERR_CBF_INVALID_CON_TYPE, 488
MSK_RES_ERR_CBF_INVALID_DIMENSION_OF_CONES, 489
MSK_RES_ERR_CBF_INVALID_DIMENSION_OF_PSDCON, 489
MSK_RES_ERR_CBF_INVALID_DOMAIN_DIMENSION, 488
MSK_RES_ERR_CBF_INVALID_EXP_DIMENSION, 488
MSK_RES_ERR_CBF_INVALID_INT_INDEX, 488
MSK_RES_ERR_CBF_INVALID_NUM_ACOORD, 489
MSK_RES_ERR_CBF_INVALID_NUM_BCOORD, 489
MSK_RES_ERR_CBF_INVALID_NUM_DCOORD, 489
MSK_RES_ERR_CBF_INVALID_NUM_FCOORD, 489
MSK_RES_ERR_CBF_INVALID_NUM_HCOORD, 489
MSK_RES_ERR_CBF_INVALID_NUM_OBJACOORD, 489
MSK_RES_ERR_CBF_INVALID_NUM_OBJFCOORD, 489
MSK_RES_ERR_CBF_INVALID_NUM_PSDCON, 489
MSK_RES_ERR_CBF_INVALID_NUMBER_OF_CONES, 489
MSK_RES_ERR_CBF_INVALID_POWER, 488
MSK_RES_ERR_CBF_INVALID_POWER_CONE_INDEX, 488
MSK_RES_ERR_CBF_INVALID_POWER_STAR_CONE_INDEX, 489
MSK_RES_ERR_CBF_INVALID_PSDCON_BLOCK_INDEX, 489
MSK_RES_ERR_CBF_INVALID_PSDCON_INDEX, 489
MSK_RES_ERR_CBF_INVALID_PSDCON_VARIABLE_INDEX, 489
MSK_RES_ERR_CBF_INVALID_PSDVAR_DIMENSION, 488
MSK_RES_ERR_CBF_INVALID_VAR_TYPE, 488
MSK_RES_ERR_CBF_NO_VARIABLES, 487
MSK_RES_ERR_CBF_NO_VERSION_SPECIFIED, 487
MSK_RES_ERR_CBF_OBJ_SENSE, 487
MSK_RES_ERR_CBF_PARSE, 487
MSK_RES_ERR_CBF_POWER_CONE_IS_TOO_LONG, 488
MSK_RES_ERR_CBF_POWER_CONE_MISMATCH, 489
MSK_RES_ERR_CBF_POWER_STAR_CONE_MISMATCH, 489
MSK_RES_ERR_CBF_SYNTAX, 488
MSK_RES_ERR_CBF_TOO_FEW_CONSTRAINTS, 488
MSK_RES_ERR_CBF_TOO_FEW_INTS, 488
MSK_RES_ERR_CBF_TOO_FEW_PSDVAR, 488
MSK_RES_ERR_CBF_TOO_FEW_VARIABLES, 488
MSK_RES_ERR_CBF_TOO_MANY_CONSTRAINTS, 487

MSK_RES_ERR_CBF_TOO_MANY_INTS, 488
MSK_RES_ERR_CBF_TOO_MANY_VARIABLES, 487
MSK_RES_ERR_CBF_UNHANDLED_POWER_CONE_TYPE, 489
MSK_RES_ERR_CBF_UNHANDLED_POWER_STAR_CONE_TYPE, 489
MSK_RES_ERR_CBF_UNSUPPORTED, 488
MSK_RES_ERR_CBF_UNSUPPORTED_CHANGE, 489
MSK_RES_ERR_CON_Q_NOT_NSD, 481
MSK_RES_ERR_CON_Q_NOT_PSD, 481
MSK_RES_ERR_CONE_INDEX, 481
MSK_RES_ERR_CONE_OVERLAP, 481
MSK_RES_ERR_CONE_OVERLAP_APPEND, 481
MSK_RES_ERR_CONE_PARAMETER, 481
MSK_RES_ERR_CONE_REP_VAR, 481
MSK_RES_ERR_CONE_SIZE, 481
MSK_RES_ERR_CONE_TYPE, 481
MSK_RES_ERR_CONE_TYPE_STR, 481
MSK_RES_ERR_DATA_FILE_EXT, 474
MSK_RES_ERR_DIMENSION_SPECIFICATION, 475
MSK_RES_ERR_DJC_AFE_DOMAIN_MISMATCH, 491
MSK_RES_ERR_DJC_DOMAIN_TERMSIZE_MISMATCH, 491
MSK_RES_ERR_DJC_INVALID_INDEX, 491
MSK_RES_ERR_DJC_INVALID_TERM_SIZE, 491
MSK_RES_ERR_DJC_TOTAL_NUM_TERMS_MISMATCH, 491
MSK_RES_ERR_DJC_UNSUPPORTED_DOMAIN_TYPE, 491
MSK_RES_ERR_DOMAIN_DIMENSION, 490
MSK_RES_ERR_DOMAIN_DIMENSION_PSD, 490
MSK_RES_ERR_DOMAIN_INVALID_INDEX, 490
MSK_RES_ERR_DOMAIN_POWER_INVALID_ALPHA, 491
MSK_RES_ERR_DOMAIN_POWER_NEGATIVE_ALPHA, 491
MSK_RES_ERR_DOMAIN_POWER_NLEFT, 491
MSK_RES_ERR_DUP_NAME, 474
MSK_RES_ERR_DUPLICATE_AIJ, 482
MSK_RES_ERR_DUPLICATE_BARVARIABLE_NAMES, 486
MSK_RES_ERR_DUPLICATE_CONE_NAMES, 486
MSK_RES_ERR_DUPLICATE_CONSTRAINT_NAMES, 486
MSK_RES_ERR_DUPLICATE_DJC_NAMES, 486
MSK_RES_ERR_DUPLICATE_DOMAIN_NAMES, 486
MSK_RES_ERR_DUPLICATE_FIJ, 490
MSK_RES_ERR_DUPLICATE_INDEX_IN_A_SPARSE_MATRIX, 490
MSK_RES_ERR_DUPLICATE_INDEX_IN_AFEIDX_LIST, 490
MSK_RES_ERR_DUPLICATE_VARIABLE_NAMES, 486
MSK_RES_ERR_END_OF_FILE, 474
MSK_RES_ERR_FACTOR, 484
MSK_RES_ERR_FEASREPAIR_CANNOT_RELAX, 484
MSK_RES_ERR_FEASREPAIR_INCONSISTENT_BOUND, 484
MSK_RES_ERR_FEASREPAIR_SOLVING_RELAXED, 484
MSK_RES_ERR_FILE_LICENSE, 472
MSK_RES_ERR_FILE_OPEN, 473
MSK_RES_ERR_FILE_READ, 474
MSK_RES_ERR_FILE_WRITE, 474
MSK_RES_ERR_FINAL_SOLUTION, 483
MSK_RES_ERR_FIRST, 483
MSK_RES_ERR_FIRSTI, 480
MSK_RES_ERR_FIRSTJ, 480
MSK_RES_ERR_FIXED_BOUND_VALUES, 482
MSK_RES_ERR_FLEXLM, 472
MSK_RES_ERR_FORMAT_STRING, 474
MSK_RES_ERR_GETDUAL_NOT_AVAILABLE, 490
MSK_RES_ERR_GLOBAL_INV_CONIC_PROBLEM, 483
MSK_RES_ERR_HUGE_AIJ, 482
MSK_RES_ERR_HUGE_C, 482
MSK_RES_ERR_HUGE_FIJ, 490
MSK_RES_ERR_IDENTICAL_TASKS, 485
MSK_RES_ERR_IN_ARGUMENT, 478
MSK_RES_ERR_INDEX, 479
MSK_RES_ERR_INDEX_ARR_IS_TOO_LARGE, 478
MSK_RES_ERR_INDEX_ARR_IS_TOO_SMALL, 478
MSK_RES_ERR_INDEX_IS_NOT_UNIQUE, 478
MSK_RES_ERR_INDEX_IS_TOO_LARGE, 478
MSK_RES_ERR_INDEX_IS_TOO_SMALL, 478
MSK_RES_ERR_INF_DOU_INDEX, 478
MSK_RES_ERR_INF_DOU_NAME, 479
MSK_RES_ERR_INF_IN_DOUBLE_DATA, 482
MSK_RES_ERR_INF_INT_INDEX, 478
MSK_RES_ERR_INF_INT_NAME, 479
MSK_RES_ERR_INF_LINT_INDEX, 479
MSK_RES_ERR_INF_LINT_NAME, 479
MSK_RES_ERR_INF_TYPE, 479
MSK_RES_ERR_INFEAS_UNDEFINED, 486
MSK_RES_ERR_INFINITE_BOUND, 482
MSK_RES_ERR_INT64_TO_INT32_CAST, 485
MSK_RES_ERR_INTERNAL, 485
MSK_RES_ERR_INTERNAL_TEST_FAILED, 485
MSK_RES_ERR_INV_APTRE, 479
MSK_RES_ERR_INV_BK, 480
MSK_RES_ERR_INV_BKC, 480
MSK_RES_ERR_INV_BKX, 480
MSK_RES_ERR_INV_CONE_TYPE, 480
MSK_RES_ERR_INV_CONE_TYPE_STR, 480
MSK_RES_ERR_INV_DINF, 480
MSK_RES_ERR_INV_IINF, 480
MSK_RES_ERR_INV_LIINF, 480
MSK_RES_ERR_INV_MARKI, 484
MSK_RES_ERR_INV_MARKJ, 484
MSK_RES_ERR_INV_NAME_ITEM, 480
MSK_RES_ERR_INV_NUMI, 484
MSK_RES_ERR_INV_NUMJ, 484
MSK_RES_ERR_INV_OPTIMIZER, 483
MSK_RES_ERR_INV_PROBLEM, 483
MSK_RES_ERR_INV_QCON_SUBI, 482
MSK_RES_ERR_INV_QCON_SUBJ, 482
MSK_RES_ERR_INV_QCON_SUBK, 482
MSK_RES_ERR_INV_QCON_VAL, 482
MSK_RES_ERR_INV_QOBJ_SUBI, 482
MSK_RES_ERR_INV_QOBJ_SUBJ, 482
MSK_RES_ERR_INV_QOBJ_VAL, 482

MSK_RES_ERR_INV_RESCODE, 480
MSK_RES_ERR_INV_SK, 480
MSK_RES_ERR_INV_SK_STR, 480
MSK_RES_ERR_INV_SKC, 480
MSK_RES_ERR_INV_SKN, 480
MSK_RES_ERR_INV_SKX, 480
MSK_RES_ERR_INV_VAR_TYPE, 480
MSK_RES_ERR_INVALID_AIJ, 483
MSK_RES_ERR_INVALID_B, 490
MSK_RES_ERR_INVALID_BARVAR_NAME, 475
MSK_RES_ERR_INVALID_CFIX, 483
MSK_RES_ERR_INVALID_CJ, 483
MSK_RES_ERR_INVALID_COMPRESSION, 484
MSK_RES_ERR_INVALID_CON_NAME, 474
MSK_RES_ERR_INVALID_CONE_NAME, 474
MSK_RES_ERR_INVALID_FIJ, 490
MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_AFFINE_CONIC_CONSTRAINTS, 486
MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_CFIX, 486
MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_CONES, 486
MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_DISJUNCTIVE_CONSTRAINTS, 486
MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_FREE_CONSTRAINTS, 486
MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_NONLINEAR, 486
MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_QUADRATIC_TERMS, 486
MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_RANGED_CONSTRAINTS, 486
MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_SYM_MAT, 486
MSK_RES_ERR_INVALID_FILE_NAME, 474
MSK_RES_ERR_INVALID_FORMAT_TYPE, 480
MSK_RES_ERR_INVALID_G, 490
MSK_RES_ERR_INVALID_IDX, 479
MSK_RES_ERR_INVALID_IOMODE, 484
MSK_RES_ERR_INVALID_MAX_NUM, 479
MSK_RES_ERR_INVALID_NAME_IN_SOL_FILE, 477
MSK_RES_ERR_INVALID_OBJ_NAME, 474
MSK_RES_ERR_INVALID_OBJECTIVE_SENSE, 482
MSK_RES_ERR_INVALID_PROBLEM_TYPE, 487
MSK_RES_ERR_INVALID_SOL_FILE_NAME, 474
MSK_RES_ERR_INVALID_STREAM, 474
MSK_RES_ERR_INVALID_SURPLUS, 480
MSK_RES_ERR_INVALID_SYM_MAT_DIM, 486
MSK_RES_ERR_INVALID_TASK, 474
MSK_RES_ERR_INVALID_UTF8, 485
MSK_RES_ERR_INVALID_VAR_NAME, 474
MSK_RES_ERR_INVALID_WCHAR, 485
MSK_RES_ERR_INVALID_WHICHSOL, 479
MSK_RES_ERR_JSON_DATA, 477
MSK_RES_ERR_JSON_FORMAT, 477
MSK_RES_ERR_JSON_MISSING_DATA, 477
MSK_RES_ERR_JSON_NUMBER_OVERFLOW, 477
MSK_RES_ERR_JSON_STRING, 477
MSK_RES_ERR_JSON_SYNTAX, 477
MSK_RES_ERR_LAST, 483
MSK_RES_ERR_LASTI, 480
MSK_RES_ERR_LASTJ, 481
MSK_RES_ERR_LAU_ARG_K, 487
MSK_RES_ERR_LAU_ARG_M, 487
MSK_RES_ERR_LAU_ARG_N, 487
MSK_RES_ERR_LAU_ARG_TRANS, 487
MSK_RES_ERR_LAU_ARG_TRANSA, 487
MSK_RES_ERR_LAU_ARG_TRANSB, 487
MSK_RES_ERR_LAU_ARG_UPLO, 487
MSK_RES_ERR_LAU_INVALID_LOWER_TRIANGULAR_MATRIX, 487
MSK_RES_ERR_LAU_INVALID_SPARSE_SYMMETRIC_MATRIX, 487
MSK_RES_ERR_LAU_NOT_POSITIVE_DEFINITE, 487
MSK_RES_ERR_LAU_SINGULAR_MATRIX, 487
MSK_RES_ERR_LAU_UNKNOWN, 487
MSK_RES_ERR_LICENSE, 472
MSK_RES_ERR_LICENSE_CANNOT_ALLOCATE, 473
MSK_RES_ERR_LICENSE_CANNOT_CONNECT, 473
MSK_RES_ERR_LICENSE_EXPIRED, 472
MSK_RES_ERR_LICENSE_FEATURE, 473
MSK_RES_ERR_LICENSE_INVALID_HOSTID, 473
MSK_RES_ERR_LICENSE_MAX, 472
MSK_RES_ERR_LICENSE_MOSEKLM_DAEMON, 473
MSK_RES_ERR_LICENSE_NO_SERVER_LINE, 473
MSK_RES_ERR_LICENSE_NO_SERVER_SUPPORT, 473
MSK_RES_ERR_LICENSE_OLD_SERVER_VERSION, 472
MSK_RES_ERR_LICENSE_SERVER, 472
MSK_RES_ERR_LICENSE_SERVER_VERSION, 473
MSK_RES_ERR_LICENSE_VERSION, 472
MSK_RES_ERR_LINK_FILE_DLL, 473
MSK_RES_ERR_LIVING_TASKS, 474
MSK_RES_ERR_LOWER_BOUND_IS_A_NAN, 482
MSK_RES_ERR_LP_AMBIGUOUS_CONSTRAINT_BOUND, 477
MSK_RES_ERR_LP_DUPLICATE_SECTION, 477
MSK_RES_ERR_LP_EMPTY, 477
MSK_RES_ERR_LP_EXPECTED_CONSTRAINT_RELATION, 477
MSK_RES_ERR_LP_EXPECTED_NUMBER, 477
MSK_RES_ERR_LP_EXPECTED_OBJECTIVE, 477
MSK_RES_ERR_LP_FILE_FORMAT, 477
MSK_RES_ERR_LP_INDICATOR_VAR, 477
MSK_RES_ERR_LP_INVALID_VAR_NAME, 477
MSK_RES_ERR_LU_MAX_NUM_TRIES, 484
MSK_RES_ERR_MAX_LEN_IS_TOO_SMALL, 481
MSK_RES_ERR_MAXNUMBARVAR, 479
MSK_RES_ERR_MAXNUMCON, 479
MSK_RES_ERR_MAXNUMCONE, 481
MSK_RES_ERR_MAXNUMQNZ, 479
MSK_RES_ERR_MAXNUMVAR, 479
MSK_RES_ERR_MIO_INTERNAL, 487
MSK_RES_ERR_MIO_INVALID_NODE_OPTIMIZER, 489
MSK_RES_ERR_MIO_INVALID_ROOT_OPTIMIZER, 489
MSK_RES_ERR_MIO_NO_OPTIMIZER, 483
MSK_RES_ERR_MISMATCHING_DIMENSION, 474

MSK_RES_ERR_MISSING_LICENSE_FILE, 472
MSK_RES_ERR_MIXED_CONIC_AND_NL, 483
MSK_RES_ERR_MPS_CONE_OVERLAP, 476
MSK_RES_ERR_MPS_CONE_REPEAT, 476
MSK_RES_ERR_MPS_CONE_TYPE, 476
MSK_RES_ERR_MPS_DUPLICATE_Q_ELEMENT, 476
MSK_RES_ERR_MPS_FILE, 475
MSK_RES_ERR_MPS_INV_FIELD, 475
MSK_RES_ERR_MPS_INV_MARKER, 475
MSK_RES_ERR_MPS_INV_SEC_ORDER, 476
MSK_RES_ERR_MPS_INVALID_BOUND_KEY, 475
MSK_RES_ERR_MPS_INVALID_CON_KEY, 475
MSK_RES_ERR_MPS_INVALID_INDICATOR_CONSTRAINT, 476
MSK_RES_ERR_MPS_INVALID_INDICATOR_QUADRATIC_CONSTRAINT, 476
MSK_RES_ERR_MPS_INVALID_INDICATOR_VALUE, 476
MSK_RES_ERR_MPS_INVALID_INDICATOR_VARIABLE, 476
MSK_RES_ERR_MPS_INVALID_KEY, 476
MSK_RES_ERR_MPS_INVALID_OBJ_NAME, 476
MSK_RES_ERR_MPS_INVALID_OBJSENSE, 476
MSK_RES_ERR_MPS_INVALID_SEC_NAME, 475
MSK_RES_ERR_MPS_MUL_CON_NAME, 475
MSK_RES_ERR_MPS_MUL_CSEC, 476
MSK_RES_ERR_MPS_MUL_QOBJ, 476
MSK_RES_ERR_MPS_MUL_QSEC, 475
MSK_RES_ERR_MPS_NO_OBJECTIVE, 475
MSK_RES_ERR_MPS_NON_SYMMETRIC_Q, 476
MSK_RES_ERR_MPS_NULL_CON_NAME, 475
MSK_RES_ERR_MPS_NULL_VAR_NAME, 475
MSK_RES_ERR_MPS_SPLITTED_VAR, 475
MSK_RES_ERR_MPS_TAB_IN_FIELD2, 476
MSK_RES_ERR_MPS_TAB_IN_FIELD3, 476
MSK_RES_ERR_MPS_TAB_IN_FIELD5, 476
MSK_RES_ERR_MPS_UNDEF_CON_NAME, 475
MSK_RES_ERR_MPS_UNDEF_VAR_NAME, 475
MSK_RES_ERR_MPS_WRITE_CPLEX_INVALID_CONE_TYPE, 489
MSK_RES_ERR_MUL_A_ELEMENT, 480
MSK_RES_ERR_NAME_IS_NULL, 484
MSK_RES_ERR_NAME_MAX_LEN, 484
MSK_RES_ERR_NAN_IN_BLC, 483
MSK_RES_ERR_NAN_IN_BLX, 483
MSK_RES_ERR_NAN_IN_BUC, 483
MSK_RES_ERR_NAN_IN_BUX, 483
MSK_RES_ERR_NAN_IN_C, 483
MSK_RES_ERR_NAN_IN_DOUBLE_DATA, 482
MSK_RES_ERR_NEGATIVE_APPEND, 483
MSK_RES_ERR_NEGATIVE_SURPLUS, 483
MSK_RES_ERR_NEWER_DLL, 473
MSK_RES_ERR_NO_BARS_FOR_SOLUTION, 486
MSK_RES_ERR_NO_BARX_FOR_SOLUTION, 486
MSK_RES_ERR_NO_BASIS_SOL, 483
MSK_RES_ERR_NO_DOTY, 491
MSK_RES_ERR_NO_DUAL_FOR_ITG_SOL, 485
MSK_RES_ERR_NO_DUAL_INFEAS_CER, 484
MSK_RES_ERR_NO_INIT_ENV, 474
MSK_RES_ERR_NO_OPTIMIZER_VAR_TYPE, 483
MSK_RES_ERR_NO_PRIMAL_INFEAS_CER, 484
MSK_RES_ERR_NO_SNX_FOR_BAS_SOL, 485
MSK_RES_ERR_NO_SOLUTION_IN_CALLBACK, 484
MSK_RES_ERR_NON_UNIQUE_ARRAY, 487
MSK_RES_ERR_NONCONVEX, 481
MSK_RES_ERR_NONLINEAR_EQUALITY, 481
MSK_RES_ERR_NONLINEAR_RANGED, 481
MSK_RES_ERR_NOT_POWER_DOMAIN, 491
MSK_RES_ERR_NULL_ENV, 474
MSK_RES_ERR_NULL_POINTER, 474
MSK_RES_ERR_NULL_TASK, 474
MSK_RES_ERR_NUM_ARGUMENTS, 478
MSK_RES_ERR_NUMCONLIM, 479
MSK_RES_ERR_NUMVARLIM, 479
MSK_RES_ERR_OBJ_Q_NOT_NSD, 481
MSK_RES_ERR_OBJ_Q_NOT_PSD, 481
MSK_RES_ERR_OBJECTIVE_RANGE, 480
MSK_RES_ERR_OLDER_DLL, 473
MSK_RES_ERR_OPF_DUAL_INTEGER_SOLUTION, 477
MSK_RES_ERR_OPF_DUPLICATE_BOUND, 476
MSK_RES_ERR_OPF_DUPLICATE_CONE_ENTRY, 477
MSK_RES_ERR_OPF_DUPLICATE_CONSTRAINT_NAME, 476
MSK_RES_ERR_OPF_INCORRECT_TAG_PARAM, 476
MSK_RES_ERR_OPF_INVALID_CONE_TYPE, 476
MSK_RES_ERR_OPF_INVALID_TAG, 476
MSK_RES_ERR_OPF_MISMATCHED_TAG, 476
MSK_RES_ERR_OPF_PREMATURE_EOF, 476
MSK_RES_ERR_OPF_SYNTAX, 476
MSK_RES_ERR_OPF_TOO_LARGE, 477
MSK_RES_ERR_OPTIMIZER_LICENSE, 472
MSK_RES_ERR_OVERFLOW, 483
MSK_RES_ERR_PARAM_INDEX, 478
MSK_RES_ERR_PARAM_IS_TOO_LARGE, 478
MSK_RES_ERR_PARAM_IS_TOO_SMALL, 478
MSK_RES_ERR_PARAM_NAME, 478
MSK_RES_ERR_PARAM_NAME_DOU, 478
MSK_RES_ERR_PARAM_NAME_INT, 478
MSK_RES_ERR_PARAM_NAME_STR, 478
MSK_RES_ERR_PARAM_TYPE, 478
MSK_RES_ERR_PARAM_VALUE_STR, 478
MSK_RES_ERR_PLATFORM_NOT_LICENSED, 473
MSK_RES_ERR_POSTSOLVE, 483
MSK_RES_ERR_PRO_ITEM, 480
MSK_RES_ERR_PROB_LICENSE, 472
MSK_RES_ERR_PTF_FORMAT, 478
MSK_RES_ERR_PTF_INCOMPATIBILITY, 478
MSK_RES_ERR_PTF_INCONSISTENCY, 478
MSK_RES_ERR_PTF_UNDEFINED_ITEM, 478
MSK_RES_ERR_QCON_SUBI_TOO_LARGE, 482
MSK_RES_ERR_QCON_SUBI_TOO_SMALL, 482
MSK_RES_ERR_QCON_UPPER_TRIANGLE, 482
MSK_RES_ERR_QOBJ_UPPER_TRIANGLE, 482
MSK_RES_ERR_READ_ASYNC, 474
MSK_RES_ERR_READ_FORMAT, 475
MSK_RES_ERR_READ_GZIP, 474

MSK_RES_ERR_READ_LP_DELAYED_ROWS_NOT_SUPPORTED, 477
MSK_RES_ERR_READ_LP_MISSING_END_TAG, 477
MSK_RES_ERR_READ_PREMATURE_EOF, 475
MSK_RES_ERR_READ_WRITE, 484
MSK_RES_ERR_READ_ZSTD, 474
MSK_RES_ERR_REMOVE_CONE_VARIABLE, 481
MSK_RES_ERR_REPAIR_INVALID_PROBLEM, 484
MSK_RES_ERR_REPAIR_OPTIMIZATION_FAILED, 484
MSK_RES_ERR_SEN_BOUND_INVALID_LO, 485
MSK_RES_ERR_SEN_BOUND_INVALID_UP, 485
MSK_RES_ERR_SEN_FORMAT, 485
MSK_RES_ERR_SEN_INDEX_INVALID, 485
MSK_RES_ERR_SEN_INDEX_RANGE, 485
MSK_RES_ERR_SEN_INVALID_REGEXP, 485
MSK_RES_ERR_SEN_NUMERICAL, 485
MSK_RES_ERR_SEN_SOLUTION_STATUS, 485
MSK_RES_ERR_SEN_UNDEF_NAME, 485
MSK_RES_ERR_SEN_UNHANDLED_PROBLEM_TYPE, 485
MSK_RES_ERR_SERVER_ACCESS_TOKEN, 490
MSK_RES_ERR_SERVER_ADDRESS, 490
MSK_RES_ERR_SERVER_CERTIFICATE, 490
MSK_RES_ERR_SERVER_CONNECT, 490
MSK_RES_ERR_SERVER_HARD_TIMEOUT, 490
MSK_RES_ERR_SERVER_PROBLEM_SIZE, 490
MSK_RES_ERR_SERVER_PROTOCOL, 490
MSK_RES_ERR_SERVER_STATUS, 490
MSK_RES_ERR_SERVER_TLS_CLIENT, 490
MSK_RES_ERR_SERVER_TOKEN, 490
MSK_RES_ERR_SHAPE_IS_TOO_LARGE, 478
MSK_RES_ERR_SIZE_LICENSE, 472
MSK_RES_ERR_SIZE_LICENSE_CON, 472
MSK_RES_ERR_SIZE_LICENSE_INTVAR, 472
MSK_RES_ERR_SIZE_LICENSE_VAR, 472
MSK_RES_ERR_SLICE_SIZE, 483
MSK_RES_ERR_SOL_FILE_INVALID_NUMBER, 481
MSK_RES_ERR_SOLITEM, 479
MSK_RES_ERR_SOLVER_PROBTYPE, 480
MSK_RES_ERR_SPACE, 473
MSK_RES_ERR_SPACE_LEAKING, 475
MSK_RES_ERR_SPACE_NO_INFO, 475
MSK_RES_ERR_SPARSITY_SPECIFICATION, 474
MSK_RES_ERR_SYM_MAT_DUPLICATE, 486
MSK_RES_ERR_SYM_MAT_HUGE, 483
MSK_RES_ERR_SYM_MAT_INVALID, 483
MSK_RES_ERR_SYM_MAT_INVALID_COL_INDEX, 486
MSK_RES_ERR_SYM_MAT_INVALID_ROW_INDEX, 486
MSK_RES_ERR_SYM_MAT_INVALID_VALUE, 486
MSK_RES_ERR_SYM_MAT_NOT_LOWER_TRINGULAR, 486
MSK_RES_ERR_TASK_INCOMPATIBLE, 484
MSK_RES_ERR_TASK_INVALID, 484
MSK_RES_ERR_TASK_PREMATURE_EOF, 484
MSK_RES_ERR_TASK_WRITE, 484
MSK_RES_ERR_THREAD_COND_INIT, 473
MSK_RES_ERR_THREAD_CREATE, 473
MSK_RES_ERR_THREAD_MUTEX_INIT, 473
MSK_RES_ERR_THREAD_MUTEX_LOCK, 473
MSK_RES_ERR_THREAD_MUTEX_UNLOCK, 473
MSK_RES_ERR_TOCONIC_CONSTR_NOT_CONIC, 490
MSK_RES_ERR_TOCONIC_CONSTR_Q_NOT_PSD, 489
MSK_RES_ERR_TOCONIC_CONSTRAINT_FX, 490
MSK_RES_ERR_TOCONIC_CONSTRAINT_RA, 490
MSK_RES_ERR_TOCONIC_OBJECTIVE_NOT_PSD, 490
MSK_RES_ERR_TOO_SMALL_A_TRUNCATION_VALUE, 482
MSK_RES_ERR_TOO_SMALL_MAX_NUM_NZ, 479
MSK_RES_ERR_TOO_SMALL_MAXNUMANZ, 479
MSK_RES_ERR_UNALLOWED_WHICHSOL, 479
MSK_RES_ERR_UNB_STEP_SIZE, 485
MSK_RES_ERR_UNDEF_SOLUTION, 491
MSK_RES_ERR_UNDEFINED_OBJECTIVE_SENSE, 482
MSK_RES_ERR_UNHANDLED_SOLUTION_STATUS, 487
MSK_RES_ERR_UNKNOWN, 473
MSK_RES_ERR_UPPER_BOUND_IS_A_NAN, 482
MSK_RES_ERR_UPPER_TRIANGLE, 487
MSK_RES_ERR_WHICHITEM_NOT_ALLOWED, 479
MSK_RES_ERR_WHICHSOL, 479
MSK_RES_ERR_WRITE_ASYNC, 477
MSK_RES_ERR_WRITE_LP_DUPLICATE_CON_NAMES, 475
MSK_RES_ERR_WRITE_LP_DUPLICATE_VAR_NAMES, 475
MSK_RES_ERR_WRITE_LP_INVALID_CON_NAMES, 475
MSK_RES_ERR_WRITE_LP_INVALID_VAR_NAMES, 475
MSK_RES_ERR_WRITE_MPS_INVALID_NAME, 477
MSK_RES_ERR_WRITE_OPF_INVALID_VAR_NAME, 477
MSK_RES_ERR_WRITING_FILE, 477
MSK_RES_ERR_Y_IS_UNDEFINED, 482

Index

A

ACC, 20
affine conic constraints, 20
analysis
    infeasibility, 193
asset, *see* portfolio optimization
attaching
    streams, 18

B

basic
    solution, 79
basis identification, 109, 171
basis type
    sensitivity analysis, 199
big-M, 189
bound
    constraint, 15, 155, 158, 162
    linear optimization, 15
    variable, 15, 155, 158, 162
Branch-and-Bound, 180

C

callback, 88
cardinality constraints, 143
CBF format, 553
ceo1
    example, 35
certificate, 80
    dual, 157, 160
    infeasibility, 74
    infeasible, 74
    primal, 157, 160
Cholesky factorization, 115, 131
CLASSPATH, 10
column ordered
    matrix format, 207
complementarity, 156, 160
concurrent optimizer, 151
cone
    dual, 159
    dual exponential, 35
    exponential, 35
    power, 32
    quadratic, 28
    rotated quadratic, 28
    semidefinite, 42
conic exponential optimization, 35
conic optimization, 20, 28, 32, 35, 158
    interior-point, 175
    mixed-integer, 188
    termination criteria, 177
conic problem
    example, 29, 32, 35
conic quadratic optimization, 28
constraint
    bound, 15, 155, 158, 162
    linear optimization, 15
    matrix, 15, 155, 158, 162
    quadratic, 163
constraint programming, 56
correlation matrix, 124
covariance matrix, *see* correlation matrix
cqo1
    example, 29
cuts, 186
cutting planes, 186

D

defining
    objective, 18
determinism, 119
disjunction, 56
disjunctive constraint, 189
disjunctive constraints, 56
DJC, 56
domain, 523
dual
    certificate, 157, 160
    cone, 159
    feasible, 156
    infeasible, 156, 157, 160
    problem, 156, 159, 163
    solution, 81
    variable, 156, 159
duality
    conic, 159
    linear, 156
    semidefinite, 163
dualizer, 167

E

efficient frontier, 128
eliminator, 167
environment variable
    MOSEKJL_FORCE_DOWNLOAD, 10
error
    optimization, 79

errors, 82
example
    ceo1, 35
    conic problem, 29, 32, 35
    cqo1, 29
    lo1, 18
    pow1, 32
    qo1, 62
    quadratic objective, 62
exceptions, 82
exponential cone, 35

F

factor model, 131
feasibility
    integer feasibility, 182
feasible
    dual, 156
    primal, 155, 169, 176
    problem, 155
format, 86
    CBF, 553
    json, 579
    LP, 527
    MPS, 531
    OPF, 543
    PTF, 571
    sol, 585
    task, 579
full
    vector format, 206

G

geometric programming, 38
GP, 38

H

heuristic, 185
hot-start, 173

I

I/O, 86
infeasibility, 80, 157, 160
    analysis, 193
    linear optimization, 157
    repair, 193
    semidefinite, 163
infeasibility certificate, 74
infeasible
    dual, 156, 157, 160
    primal, 155, 157, 160, 169, 176
    problem, 155, 157, 163
information item, 87, 89
installation, 9
    requirements, 9
    troubleshooting, 9
integer
    solution, 79
    variable, 52
integer feasibility, 182
    feasibility, 182
integer optimization, 52, 56
    initial solution, 54
    parameter, 52
interior-point
    conic optimization, 175
    linear optimization, 169
    logging, 172, 178
    optimizer, 169, 175
    solution, 79
    termination criteria, 170, 177

J

json format, 579

L

license, 121
linear
    objective, 18
linear constraint matrix, 15
linear dependency, 167
linear optimization, 15, 155
    bound, 15
    constraint, 15
    infeasibility, 157
    interior-point, 169
    objective, 15
    simplex, 173
    termination criteria, 170, 173
    variable, 15
linearity interval, 198
lo1
    example, 18
log-sum-exp, 147
logging, 85
    interior-point, 172, 178
    mixed-integer optimizer, 183
    optimizer, 172, 174, 178
    simplex, 174
logistic regression, 147
LP format, 527

M

machine learning
    logistic regression, 147
market impact cost, 136
Markowitz model, 124
matrix
    constraint, 15, 155, 158, 162
    semidefinite, 42
    symmetric, 42
matrix format
    column ordered, 207
    row ordered, 207
    triplets, 207
memory management, 119

MI(QC)QO, 189
MICO, 188
MIP, *see* integer optimization
mixed-integer, *see* integer
    conic optimization, 188
    optimizer, 179
    presolve, 185
    quadratic, 189
mixed-integer optimization, *see* integer optimization, 179
mixed-integer optimizer
    logging, 183
modeling
    design, 11
MPS format, 531
    free, 543

N

numerical issues
    presolve, 167
    scaling, 168
    simplex, 174

O

objective, 155, 158, 162
    defining, 18
    linear, 18
    linear optimization, 15
OPF format, 543
optimal
    solution, 80
optimality gap, 181
optimization
    conic, 20, 158
    conic quadratic, 158
    error, 79
    integer, 56
    linear, 15, 155
    semidefinite, 161
optimizer
    concurrent, 151
    conic, 20
    determinism, 119
    interior-point, 169, 175
    interrupt, 88
    logging, 172, 174, 178
    mixed-integer, 56, 179
    parallel, 73
    selection, 167, 168
    simplex, 173
    termination, 181

P

parallel optimization, 73, 151
parallelization, 119
parameter, 86
    integer optimization, 52
    simplex, 174
Pareto optimality, 124
portfolio optimization, 123
    cardinality constraints, 143
    efficient frontier, 128
    factor model, 131
    market impact cost, 136
    Markowitz model, 124
    Pareto optimality, 124
    slippage cost, 136
    transaction cost, 140
positive semidefinite, 62
pow1
    example, 32
power cone, 32
power cone optimization, 32
presolve, 166
    eliminator, 167
    linear dependency check, 167
    mixed-integer, 185
    numerical issues, 167
primal
    certificate, 157, 160
    feasible, 155, 169, 176
    infeasible, 155, 157, 160, 169, 176
    problem, 156, 159, 163
    solution, 81, 155
primal heuristics, 185
primal-dual
    problem, 169, 175
    solution, 156
problem
    dual, 156, 159, 163
    feasible, 155
    infeasible, 155, 157, 163
    load, 86
    primal, 156, 159, 163
    primal-dual, 169, 175
    save, 86
    status, 79
    unbounded, 157, 161
PTF format, 571

Q

qo1
    example, 62
quadratic
    constraint, 163
    mixed-integer, 189
quadratic cone, 28
quadratic objective
    example, 62
quadratic optimization, 163

R

regression
    logistic, 147
relaxation, 180
repair

infeasibility, 193
response code, 82
rotated quadratic cone, 28
row ordered
matrix format, 207

S

scaling, 168
semidefinite
cone, 42
infeasibility, 163
matrix, 42
variable, 42
semidefinite optimization, 42, 161
sensitivity analysis, 197
basis type, 199
shadow price, 198
simplex
linear optimization, 173
logging, 174
numerical issues, 174
optimizer, 173
parameter, 174
termination criteria, 173
slippage cost, 136
sol format, 585
solution
basic, 79
dual, 81
file format, 585
integer, 79
interior-point, 79
optimal, 80
primal, 81, 155
primal-dual, 156
retrieve, 79
status, 18, 80
solving linear system, 113
sparse
vector format, 206
sparse vector, 206
status
problem, 79
solution, 18, 80
streams
attaching, 18
symmetric
matrix, 42

T

task format, 579
termination, 79
optimizer, 181
termination criteria, 88, 181
conic optimization, 177
interior-point, 170, 177
linear optimization, 170, 173
simplex, 173
tolerance, 171, 178
thread, 119
time, 120
time limit, 88
timing, 120
tolerance
termination criteria, 171, 178
transaction cost, 140
triplets
matrix format, 207
troubleshooting
installation, 9

U

unbounded
problem, 157, 161
user callback, *see* callback

V

valid inequalities, 186
variable, 155, 158, 162
bound, 15, 155, 158, 162
dual, 156, 159
integer, 52
linear optimization, 15
semidefinite, 42
vector format
full, 206
sparse, 206