MOSEK Optimizer API for C

Release 10.0.46

MOSEK ApS

23 May 2023

Contents

Chapter 1

Introduction

The **MOSEK** Optimization Suite 10.0.46 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 C?

The Optimizer API for C provides low-level access to all functionalities of **MOSEK** from any C compatible language. It consists of a single header file and a set of library files which an application must link against when building. This interface has the smallest possible overhead, however other interfaces might be considered more convenient to use for the project at hand.

The Optimizer API for C provides access to:

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

as well as additional interfaces for:

- problem analysis,
- sensitivity analysis,
- infeasibility analysis,
- BLAS/LAPACK linear algebra routines.

Chapter 2

Contact Information

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

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

Blogger	https://blog.mosek.com/
Google Group	https://groups.google.com/forum/#!forum/mosek
Twitter	https://twitter.com/mosektw
Linkedin	https://www.linkedin.com/company/mosek-aps
Youtube	https://www.youtube.com/channel/UCvIyectEVLP31NXeD5mIbEw

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

Chapter 3

License Agreement

3.1 MOSEK end-user license agreement

Before using the **MOSEK** software, please read the license agreement available in the distribution at `<MSKHOME>/mosek/10.0/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 1.2.7, May 2nd, 2012

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

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

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

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

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

fplib

MOSEK 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.

{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.

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.

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

(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)

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.

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.

(continues on next page)

(continued from previous page)

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 C.

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

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

Compatibility

The Optimizer API for C is compatible with the following compiler tool chains:

Platform	Supported compiler	Framework
Linux 64 bit x86	gcc ($\geq$ 4.5)	glibc ($\geq$ 2.17)
Linux 64 bit ARM	clang ($\geq$ 10)	glibc ($\geq$ 2.29)
macOS 64 bit x86	Xcode ($\geq$ 11)	MAC OS SDK ($\geq$ 10.15)
macOS 64 bit ARM	Xcode ($\geq$ 12)	MAC OS SDK ($\geq$ 11)
Windows 32 and 64 bit	Visual Studio ($\geq$ 2017)	

In many cases older versions can also be used.

Locating files in the MOSEK Optimization Suite

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

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

Relative Path	Description	Label
`<MSKHOME>/mosek/10.0/tools/platform/<PLATFORM>/h`	Header files	`<HEADERDIR>`
`<MSKHOME>/mosek/10.0/tools/platform/<PLATFORM>/bin`	Libraries and DLLs	`<LIBDIR>`
`<MSKHOME>/mosek/10.0/tools/examples/c`	Examples	`<EXDIR>`
`<MSKHOME>/mosek/10.0/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. `win32x86`, `win64x86`, `linux64x86` or `osx64x86`.

Setting up the paths

To compile and link C code using the Optimizer API for C, the relevant path to the header file and library must be included, and run-time dependencies must be resolved. Hence to compile, one should add appropriate compiler and linker options. Details vary depending on the operating system and compiler. See the `Makefile` included in the distribution under `<MSKHOME>/mosek/10.0/tools/examples/c` for a full working example. Examples are given below.

Linux

```
gcc file.c -o file -I<HEADERDIR> -L<LIBDIR> -Wl,-rpath-link,<LIBDIR> -Wl,-rpath=
↪<LIBDIR> -lmosek64
```

The shared library `libmosek64.so.10.0` must be available at runtime.

Windows, 64bit

```
cl.exe /I<HEADERDIR> file.c /link /LIBPATH:<LIBDIR> /out:file.exe mosek64_10_0.lib
```

The shared library `mosek64_10_0.dll` must be available at runtime.

Windows, 32bit

```
cl.exe /I<HEADERDIR> file.c /link /LIBPATH:<LIBDIR> /out:file.exe mosek10_0.lib
```

The shared library `mosek10_0.dll` must be available at runtime.

macOS

```
clang file.c -o file -I<HEADERDIR> -L<LIBDIR> -Wl,-headerpad,128 -lmosek64
install_name_tool -change libmosek64.10.0.dylib <LIBDIR>/libmosek64.10.0.dylib file
```

The shared library `libmosek64.10.0.dylib` must be registered and available at runtime.

4.1 Testing the Installation and Compiling Examples

This section describes how to verify that **MOSEK** has been installed correctly, and how to build and execute the C examples distributed with **MOSEK**.

4.1.1 Windows

Compiling examples using NMake

The example directory `<EXDIR>` contains makefiles for use with Microsoft NMake. These makefiles requires that the Visual Studio tool chain is setup. Usually, the submenu containing Visual Studio also contains a Visual Studio *Command Prompt* which does the necessary setup.

To build the examples, open a DOS box and change directory to `<EXDIR>`. This directory contains a makefile named `Makefile`. To compile all examples, run the command

```
nmake /f Makefile all
```

To build only a single example instead of all examples, replace `all` by the corresponding executable name. For example, to build `lo1.exe` type

```
nmake /f Makefile lo1.exe
```

Compiling from command line

To compile and execute a distributed example, such as `lo1.c`, do the following:

1. Compile the example into an executable `lo1.exe` (we assume that the Visual Studio C compiler `cl.exe` is available). For 64-bit Windows:

   ```
   cl <EXDIR>\lo1.c /I <HEADERDIR> /link <LIBDIR>\mosek64_10_0.lib
   ```

2. To run the compiled example, enter

   ```
   lo1.exe
   ```

Adding MOSEK to a Visual Studio Project

The following walk-through is specific for Microsoft Visual Studio 2012, but may work for other versions too. To compile a project linking to **MOSEK** in Visual Studio, the following steps are necessary:

1. Create a project or open an existing project in Visual Studio.
2. In the **Solution Explorer** right-click on the relevant project and select **Properties**. This will open the **Property pages** dialog.
3. In the selection box **Configuration:** select **All Configurations**.
4. In the tree-view open **Configuration Properties** → **C/C++** → **General**.
5. In the properties click the **Additional Include Directories** field and select edit.
6. Click on the **New Folder** button and write the *full path* to the `h` header file or browse for the file. For example, for 64-bit Windows use `<HEADERDIR>`.
7. Click **OK**.
8. Back in the **Property Pages** dialog select from the tree-view **Configuration Properties** → **Linker** → **Input**.
9. In the properties view click in the **Additional Dependencies** field and select edit. This will open the **Additional Dependencies** dialog.
10. Add the full path of the **MOSEK** `lib`. For example, for 64-bit Windows:

    ```
    <LIBDIR>\mosek64_10_0.lib
    ```

11. Click **OK**.
12. Back in the **Property Pages** dialog click **OK**.

If you have selected to link with the 64 bit version of **MOSEK** you must also target the 64-bit platform. To do this follow the steps below:

1. Open the **property pages** for that project.
2. Click **Configuration Manager** to open the Configuration Manager Dialog Box.
3. Click the **Active Solution Platform** list, and then select the **New** option to open the New Solution Platform Dialog Box.
4. Click the Type or select the new platform drop-down arrow, and then select the x64 platform.
5. Click **OK**. The platform you selected in the preceding step will appear under Active Solution Platform in the Configuration Manager dialog box.

4.1.2 macOS and Linux

The example directory `<EXDIR>` contains makefiles for use with GNU Make. To build the examples enter

```
make -f Makefile all
```

To build one example instead of all examples, replace `all` by the corresponding executable name. For example, to build the `lo1` executable enter

```
make -f Makefile lo1
```

Chapter 5

Design Overview

5.1 Modeling

Optimizer API for C 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 C 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 C.

Creating an environment and task

Optionally, an interaction with **MOSEK** using Optimizer API for C 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 *MSK_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

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

#include "mosek.h"
#include <stdio.h>

/* Error checking not included */
int main() {
  MSKrescodee      r, trmcode;
  MSKenv_t         env  = NULL;
  MSKtask_t        task = NULL;
  double           xx = 0.0;

  MSK_maketask(NULL, 0, 1, &task);      // Create task

  MSK_appendvars(task, 1);                               // 1 variable x
  MSK_putcj(task, 0, 1.0);                               // c_0 = 1.0
  MSK_putvarbound(task, 0, MSK_BK_RA, 2.0, 3.0);         // 2.0 <= x <= 3.0
  MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MINIMIZE); // Minimize

  MSK_optimizetrm(task, &trmcode);          // Optimize

  MSK_getxx(task, MSK_SOL_ITR, &xx);        // Get solution
  printf("Solution x = %f\n", xx);          // Print solution

  MSK_deletetask(&task); // Clean up task
  return 0;
}
```

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* 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

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

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

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

6.1.1 Example LO1

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

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

under the bounds

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

Solving the problem

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

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

Below we explain each of these steps.

Create 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.

Create 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.

```
    /* Create the optimization task. */
    r = MSK_maketask(NULL, numcon, numvar, &task);

    /* Directs the log task stream to the 'printstr' function. */
    if (r == MSK_RES_OK)
      r = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);
```

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.

Load 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 *MSK_appendcons* and *MSK_appendvars*.

```
    /* Append 'numcon' empty constraints.
     The constraints will initially have no bounds. */
    if (r == MSK_RES_OK)
      r = MSK_appendcons(task, numcon);

    /* Append 'numvar' variables.
     The variables will initially be fixed at zero (x=0). */
    if (r == MSK_RES_OK)
      r = MSK_appendvars(task, numvar);
```

New variables can now be referenced from other functions with indexes in $0, \ldots,$ `numvar` -1 and new constraints can be referenced with indexes in $0, \ldots,$ `numcon` -1. More variables and/or constraints can be appended later as needed, these will be assigned indexes from `numvar`/`numcon` and up.

Next step is to set the problem data. We loop over each variable index $j = 0, \ldots,$ `numvar` -1 calling functions to set problem data. We first set the objective coefficient $c_j =$ `c[j]` by calling the function *MSK_putcj*.

```
      /* Set the linear term c_j in the objective.*/
      if (r == MSK_RES_OK)
        r = MSK_putcj(task, j, c[j]);
```

Setting bounds on variables

The bounds on variables are stored in the arrays

```
  const MSKboundkeye bkx[]  = {MSK_BK_LO,     MSK_BK_RA, MSK_BK_LO,     MSK_BK_LO     ␣
↪};
  const double       blx[]  = {0.0,           0.0,       0.0,           0.0           ␣
↪};
  const double       bux[]  = { +MSK_INFINITY, 10.0,      +MSK_INFINITY, +MSK_
↪INFINITY };
```

and are set with calls to *MSK_putvarbound*.

```

      /* Set the bounds on variable j.
       blx[j] <= x_j <= bux[j] */
      if (r == MSK_RES_OK)
        r = MSK_putvarbound(task,
                              j,           /* Index of variable.*/
                              bkx[j],      /* Bound key.*/
                              blx[j],      /* Numerical value of lower bound.*/
                              bux[j]);     /* Numerical value of upper bound.*/
```

The *Bound key* stored in `bkx` specifies the type of the bound according to Table 6.1.

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

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

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

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 in the arrays:

```
  const MSKint32t     aptrb[] = {0, 2, 5, 7},
                      aptre[] = {2, 5, 7, 9},
                      asub[]  = { 0, 1,
                                  0, 1, 2,
                                  0, 1,
                                  1, 2
                                };
  const double        aval[]  = { 3.0, 2.0,
                                  1.0, 1.0, 2.0,
```

(continues on next page)

(continued from previous page)

```
                                                    2.0, 3.0,
                                                    1.0, 3.0
                                                 };
```

The aptrb, aptre, asub, and aval arguments define the constraint matrix A in the column ordered sparse format (for details, see Sec. 15.1.4).

Using the function *MSK_putacol* we set column j of A

```
        r = MSK_putacol(task,
                        j,                 /* Variable (column) index.*/
                        aptre[j] - aptrb[j], /* Number of non-zeros in column j.*/
                        asub + aptrb[j],   /* Pointer to row indexes of column j.*/
                        aval + aptrb[j]);  /* Pointer to Values of column j.*/
```

There are many alternative formats for entering the A matrix. See functions such as *MSK_putarow*, *MSK_putarowlist*, *MSK_putaijlist* and similar.

Finally, the bounds on each constraint are set by looping over each constraint index $i = 0, \ldots, \texttt{numcon} - 1$

```
    /* Set the bounds on constraints.
       for i=1, ...,numcon : blc[i] <= constraint i <= buc[i] */
    for (i = 0; i < numcon && r == MSK_RES_OK; ++i)
      r = MSK_putconbound(task,
                          i,           /* Index of constraint.*/
                          bkc[i],      /* Bound key.*/
                          blc[i],      /* Numerical value of lower bound.*/
                          buc[i]);     /* Numerical value of upper bound.*/
```

Optimization

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

```
      r = MSK_optimizetrm(task, &trmcode);
```

Extracting the solution.

After optimizing the status of the solution is examined with a call to *MSK_getsolsta*. If the solution status is reported as *MSK_SOL_STA_OPTIMAL* the solution is extracted in the lines below:

```
                MSK_getxx(task,
                          MSK_SOL_BAS,    /* Request the basic solution. */
                          xx);
```

The *MSK_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* .

Source code

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

Listing 6.1: Linear optimization example.

```
#include <stdio.h>
#include "mosek.h"

/* This function prints log output from MOSEK to the terminal. */
static void MSKAPI printstr(void       *handle,
```

(continues on next page)

(continued from previous page)

```
                             const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{
  const MSKint32t    numvar = 4,
                     numcon = 3;

  const double       c[]     = {3.0, 1.0, 5.0, 1.0};
  /* Below is the sparse representation of the A
     matrix stored by column. */
  const MSKint32t    aptrb[] = {0, 2, 5, 7},
                     aptre[] = {2, 5, 7, 9},
                     asub[]  = { 0, 1,
                                 0, 1, 2,
                                 0, 1,
                                 1, 2
                               };
  const double       aval[]  = { 3.0, 2.0,
                                 1.0, 1.0, 2.0,
                                 2.0, 3.0,
                                 1.0, 3.0
                               };

  /* Bounds on constraints. */
  const MSKboundkeye bkc[]  = {MSK_BK_FX, MSK_BK_LO,     MSK_BK_UP    };
  const double       blc[]  = {30.0,      15.0,          -MSK_INFINITY};
  const double       buc[]  = {30.0,      +MSK_INFINITY, 25.0         };
  /* Bounds on variables. */
  const MSKboundkeye bkx[]  = {MSK_BK_LO,     MSK_BK_RA, MSK_BK_LO,     MSK_BK_LO     ␣
↪};
  const double       blx[]  = {0.0,           0.0,       0.0,           0.0           ␣
↪};
  const double       bux[]  = { +MSK_INFINITY, 10.0,      +MSK_INFINITY, +MSK_
↪INFINITY };
  MSKtask_t          task = NULL;
  MSKrescodee        r = MSK_RES_OK;
  MSKint32t          i, j;

  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(NULL, numcon, numvar, &task);

    /* Directs the log task stream to the 'printstr' function. */
    if (r == MSK_RES_OK)
      r = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

    /* Append 'numcon' empty constraints.
     The constraints will initially have no bounds. */
    if (r == MSK_RES_OK)
      r = MSK_appendcons(task, numcon);

    /* Append 'numvar' variables.
```

(continues on next page)

(continued from previous page)

```
     The variables will initially be fixed at zero (x=0). */
    if (r == MSK_RES_OK)
      r = MSK_appendvars(task, numvar);

    for (j = 0; j < numvar && r == MSK_RES_OK; ++j)
    {
      /* Set the linear term c_j in the objective.*/
      if (r == MSK_RES_OK)
        r = MSK_putcj(task, j, c[j]);


      /* Set the bounds on variable j.
       blx[j] <= x_j <= bux[j] */
      if (r == MSK_RES_OK)
        r = MSK_putvarbound(task,
                            j,           /* Index of variable.*/
                            bkx[j],      /* Bound key.*/
                            blx[j],      /* Numerical value of lower bound.*/
                            bux[j]);     /* Numerical value of upper bound.*/

      /* Input column j of A */
      if (r == MSK_RES_OK)
        r = MSK_putacol(task,
                        j,                 /* Variable (column) index.*/
                        aptre[j] - aptrb[j], /* Number of non-zeros in column j.*/
                        asub + aptrb[j],   /* Pointer to row indexes of column j.*/
                        aval + aptrb[j]);  /* Pointer to Values of column j.*/
    }

    /* Set the bounds on constraints.
       for i=1, ...,numcon : blc[i] <= constraint i <= buc[i] */
    for (i = 0; i < numcon && r == MSK_RES_OK; ++i)
      r = MSK_putconbound(task,
                          i,           /* Index of constraint.*/
                          bkc[i],      /* Bound key.*/
                          blc[i],      /* Numerical value of lower bound.*/
                          buc[i]);     /* Numerical value of upper bound.*/

    /* Maximize objective function. */
    if (r == MSK_RES_OK)
      r = MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE);

    if (r == MSK_RES_OK)
    {
      MSKrescodee trmcode;

      /* Run optimizer */
      r = MSK_optimizetrm(task, &trmcode);

      /* Print a summary containing information
         about the solution for debugging purposes. */
      MSK_solutionsummary(task, MSK_STREAM_LOG);

      if (r == MSK_RES_OK)
      {
        MSKsolstae solsta;
```

(continues on next page)

(continued from previous page)

```
      if (r == MSK_RES_OK)
        r = MSK_getsolsta(task,
                          MSK_SOL_BAS,
                          &solsta);
      switch (solsta)
      {
        case MSK_SOL_STA_OPTIMAL:
          {
            double *xx = (double*) calloc(numvar, sizeof(double));
            if (xx)
            {
              MSK_getxx(task,
                        MSK_SOL_BAS,    /* Request the basic solution. */
                        xx);

              printf("Optimal primal solution\n");
              for (j = 0; j < numvar; ++j)
                printf("x[%d]: %e\n", j, xx[j]);

              free(xx);
            }
            else
              r = MSK_RES_ERR_SPACE;

            break;
          }
        case MSK_SOL_STA_DUAL_INFEAS_CER:
        case MSK_SOL_STA_PRIM_INFEAS_CER:
          printf("Primal or dual infeasibility certificate found.\n");
          break;
        case MSK_SOL_STA_UNKNOWN:
          {
            char symname[MSK_MAX_STR_LEN];
            char desc[MSK_MAX_STR_LEN];

            /* If the solutions status is unknown, print the termination code
               indicating why the optimizer terminated prematurely. */

            MSK_getcodedesc(trmcode,
                            symname,
                            desc);

            printf("The solution status is unknown.\n");
            printf("The optimizer terminitated with code: %s\n", symname);
            break;
          }
        default:
          printf("Other solution status.\n");
          break;
      }
    }
  }

  if (r != MSK_RES_OK)
  {
```

(continues on next page)

(continued from previous page)

```
      /* In case of an error print error code and description. */
      char symname[MSK_MAX_STR_LEN];
      char desc[MSK_MAX_STR_LEN];

      printf("An error occurred while optimizing.\n");
      MSK_getcodedesc(r,
                      symname,
                      desc);
      printf("Error %s - '%s'\n", symname, desc);
    }

    /* Delete the task and the associated data. */
    MSK_deletetask(&task);
  }

  return r;
}
```

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

- $x \in \mathbb{R}^n$ 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

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

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. 15.10 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:

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

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:

$$\left[\begin{array}{ccc} 0 & 0 & 0 \\ 1.5 & 0.1 & 0 \\ 0.3 & 0 & 2.1 \end{array}\right] x + \left[\begin{array}{c} 0.03 \\ 0 \\ 0.1 \end{array}\right] \in \mathcal{Q}^3. \tag{6.5}$$

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

$$\left[\begin{array}{c} 0 \\ G \end{array}\right] x + \left[\begin{array}{c} \gamma \\ h \end{array}\right] \in \mathcal{Q}^{k+1}.$$

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 **F** and a vector **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 **F** and **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 **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

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

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 */
    if (r == MSK_RES_OK)
      r = MSK_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).

```
      /* Fill in the affine expression storage with data */
      /* F matrix in sparse form */
      MSKint64t Fsubi[] = {1, 1, 2, 2};        /* G is placed from row 1 of F */
      MSKint32t Fsubj[] = {0, 1, 0, 2};
      double    Fval[]  = {1.5, 0.1, 0.3, 2.1};
      int       numEntries = 4;
      /* Other data */
      double h[]    = {0, 0.1};
      double gamma  = 0.03;

      /* Fill in F storage */
      if (r == MSK_RES_OK)
        r = MSK_putafefentrylist(task, numEntries, Fsubi, Fsubj, Fval);

      /* Fill in g storage */
      if (r == MSK_RES_OK)
        r = MSK_putafeg(task, 0, gamma);
      if (r == MSK_RES_OK)
        r = MSK_putafegslice(task, 1, k+1, 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 */
    if (r == MSK_RES_OK)
      r = MSK_appendquadraticconedomain(task, k + 1, &quadDom);
```

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, *MSK_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 */
        MSKint64t afeidx[] = {0, 1, 2};

        if (r == MSK_RES_OK)
          r = MSK_appendacc(task,
                            quadDom,    /* Domain index */
                            k + 1,      /* Dimension */
                            afeidx,     /* Indices of AFE rows [0,...,k] */
                            NULL);      /* 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 *MSK_optimize* and retrieve the solution with *MSK_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.

```
#include <stdio.h>
#include "mosek.h" /* Include the MOSEK definition file. */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{
  MSKrescodee  r;
  MSKint32t i, j;

  MSKenv_t    env  = NULL;
  MSKtask_t   task = NULL;
  MSKint64t   quadDom;

  /* Input data dimensions */
  const MSKint32t n = 3,
                  k = 2;

  /* Create the mosek environment. */
```

(continues on next page)

(continued from previous page)

```
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, 0, 0, &task);

    if (r == MSK_RES_OK)
    {
      MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

      /* Create n free variables */
      if (r == MSK_RES_OK)
        r = MSK_appendvars(task, n);
      if (r == MSK_RES_OK)
        r = MSK_putvarboundsliceconst(task, 0, n, MSK_BK_FR, -MSK_INFINITY, +MSK_
↪INFINITY);

      /* Set up the objective */
      {
        double c[] = {2.0, 3.0, -1.0};

        if (r == MSK_RES_OK)
          r = MSK_putcslice(task, 0, n, c);
        if (r == MSK_RES_OK)
          r = MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE);
      }

      /* One linear constraint sum(x) == 1 */
      if (r == MSK_RES_OK)
        r = MSK_appendcons(task, 1);
      if (r == MSK_RES_OK)
        r = MSK_putconbound(task, 0, MSK_BK_FX, 1.0, 1.0);
      for(i = 0; i < n && r == MSK_RES_OK; i++)
        r = MSK_putaij(task, 0, i, 1.0);

      /* Append empty AFE rows for affine expression storage */
      if (r == MSK_RES_OK)
        r = MSK_appendafes(task, k + 1);

      {
        /* Fill in the affine expression storage with data */
        /* F matrix in sparse form */
        MSKint64t Fsubi[] = {1, 1, 2, 2};        /* G is placed from row 1 of F */
        MSKint32t Fsubj[] = {0, 1, 0, 2};
        double    Fval[]  = {1.5, 0.1, 0.3, 2.1};
        int       numEntries = 4;
        /* Other data */
        double h[]     = {0, 0.1};
        double gamma   = 0.03;

        /* Fill in F storage */
        if (r == MSK_RES_OK)
          r = MSK_putafefentrylist(task, numEntries, Fsubi, Fsubj, Fval);

        /* Fill in g storage */
```

(continues on next page)

(continued from previous page)

```
      if (r == MSK_RES_OK)
        r = MSK_putafeg(task, 0, gamma);
      if (r == MSK_RES_OK)
        r = MSK_putafegslice(task, 1, k+1, h);
    }

    /* Define a conic quadratic domain */
    if (r == MSK_RES_OK)
      r = MSK_appendquadraticconedomain(task, k + 1, &quadDom);

    {
      /* Create the ACC */
      MSKint64t afeidx[] = {0, 1, 2};

      if (r == MSK_RES_OK)
        r = MSK_appendacc(task,
                          quadDom,    /* Domain index */
                          k + 1,      /* Dimension */
                          afeidx,     /* Indices of AFE rows [0,...,k] */
                          NULL);      /* Ignored */
    }


    /* Begin optimization and fetching the solution */
    if (r == MSK_RES_OK)
    {
      MSKrescodee trmcode;

      /* Run optimizer */
      r = MSK_optimizetrm(task, &trmcode);

      /* Print a summary containing information
         about the solution for debugging purposes*/
      MSK_solutionsummary(task, MSK_STREAM_MSG);

      if (r == MSK_RES_OK)
      {
        MSKsolstae solsta;

        MSK_getsolsta(task, MSK_SOL_ITR, &solsta);

        switch (solsta)
        {
          case MSK_SOL_STA_OPTIMAL:
            {
              double *xx, *doty, *activity = NULL;

              /* Fetch the solution */
              xx = calloc(n, sizeof(double));
              MSK_getxx(task,
                        MSK_SOL_ITR,    /* Request the interior solution. */
                        xx);

              printf("Optimal primal solution\n");
              for (j = 0; j < n; ++j)
                printf("x[%d]: %e\n", j, xx[j]);
```

(continues on next page)

(continued from previous page)

```
                free(xx);

                /* Fetch the doty dual of the ACC */
                doty = calloc(k + 1, sizeof(double));
                MSK_getaccdoty(task,
                               MSK_SOL_ITR,    /* Request the interior solution. */
                               0,              /* ACC index. */
                               doty);

                printf("Dual doty of the ACC\n");
                for (j = 0; j < k + 1; ++j)
                  printf("doty[%d]: %e\n", j, doty[j]);

                free(doty);

                /* Fetch the activity of the ACC */
                activity = calloc(k + 1, sizeof(double));
                MSK_evaluateacc(task,
                                MSK_SOL_ITR,    /* Request the interior solution. */
                                0,              /* ACC index. */
                                activity);

                printf("Activity of the ACC\n");
                for (j = 0; j < k + 1; ++j)
                  printf("activity[%d]: %e\n", j, activity[j]);

                free(activity);
              }
              break;
            case MSK_SOL_STA_DUAL_INFEAS_CER:
            case MSK_SOL_STA_PRIM_INFEAS_CER:
              printf("Primal or dual infeasibility certificate found.\n");
              break;
            case MSK_SOL_STA_UNKNOWN:
              printf("The status of the solution could not be determined. Termination␣
↪code: %d.\n", trmcode);
              break;
            default:
              printf("Other solution status.");
              break;
          }
        }
        else
        {
          printf("Error while optimizing.\n");
        }
      }

      if (r != MSK_RES_OK)
      {
        /* In case of an error print error code and description. */
        char symname[MSK_MAX_STR_LEN];
        char desc[MSK_MAX_STR_LEN];

        printf("An error occurred while optimizing.\n");
```

(continues on next page)

(continued from previous page)

```
        MSK_getcodedesc(r,
                        symname,
                        desc);
        printf("Error %s - '%s'\n", symname, desc);
      }
    }
    /* Delete the task and the associated data. */
    MSK_deletetask(&task);
  }

  /* Delete the environment and the associated data. */
  MSK_deleteenv(&env);

  return (r);
} /* main */
```

The answer is

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

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

```
                /* Fetch the doty dual of the ACC */
                doty = calloc(k + 1, sizeof(double));
                MSK_getaccdoty(task,
                               MSK_SOL_ITR,    /* Request the interior solution. */
                               0,              /* ACC index. */
                               doty);

                printf("Dual doty of the ACC\n");
                for (j = 0; j < k + 1; ++j)
                  printf("doty[%d]: %e\n", j, doty[j]);

                free(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 **F**, **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 */
      if (r == MSK_RES_OK)
        r = MSK_appendafes(task, 1);
      for(i = 0; i < n && r == MSK_RES_OK; i++)
        r = MSK_putafefentry(task, 0, i, 1.0);
      if (r == MSK_RES_OK)
        r = MSK_putafeg(task, 0, -1.0);
    }

    {
      /* Set AFE rows representing the quadratic constraint */
      /* F matrix data in sparse form */
      MSKint64t Fsubi[] = {2, 2, 3, 3};        /* G is placed from row 2 of F */
      MSKint32t Fsubj[] = {0, 1, 0, 2};
      double    Fval[]  = {1.5, 0.1, 0.3, 2.1};
      int       numEntries = 4;
      /* Other data */
      double h[]     = {0, 0.1};
      double gamma   = 0.03;

      if (r == MSK_RES_OK)
        r = MSK_appendafes(task, k + 1);
      if (r == MSK_RES_OK)
        r = MSK_putafefentrylist(task, numEntries, Fsubi, Fsubj, Fval);
      if (r == MSK_RES_OK)
        r = MSK_putafeg(task, 1, gamma);
      if (r == MSK_RES_OK)
        r = MSK_putafegslice(task, 2, k+2, h);
    }
```

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

```
    /* Define a conic quadratic domain */
    if (r == MSK_RES_OK)
      r = MSK_appendrzerodomain(task, 1, &zeroDom);
    if (r == MSK_RES_OK)
      r = MSK_appendquadraticconedomain(task, k + 1, &quadDom);
```

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:

```
      /* Linear constraint */
      MSKint64t afeidx[] = {0};

      if (r == MSK_RES_OK)
        r = MSK_appendacc(task,
                          zeroDom,     /* Domain index */
                          1,           /* Dimension */
                          afeidx,      /* Indices of AFE rows */
                          NULL);       /* Ignored */
```

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

```
      /* Quadratic constraint */
      MSKint64t afeidx[] = {1, 2, 3};

      if (r == MSK_RES_OK)
```

(continues on next page)

(continued from previous page)

```
        r = MSK_appendacc(task,
                          quadDom,    /* Domain index */
                          k + 1,      /* Dimension */
                          afeidx,     /* Indices of AFE rows */
                          NULL);      /* Ignored */
```

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

Listing 6.3: Full code of example ACC2.

```
#include <stdio.h>
#include "mosek.h" /* Include the MOSEK definition file. */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{
  MSKrescodee  r;
  MSKint32t i, j;

  MSKenv_t    env  = NULL;
  MSKtask_t   task = NULL;
  MSKint64t   zeroDom, quadDom;

  /* Input data dimensions */
  const MSKint32t n = 3,
                  k = 2;

  /* Create the mosek environment. */
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, 0, 0, &task);

    if (r == MSK_RES_OK)
    {
      MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

      /* Create n free variables */
      if (r == MSK_RES_OK)
        r = MSK_appendvars(task, n);
      if (r == MSK_RES_OK)
        r = MSK_putvarboundsliceconst(task, 0, n, MSK_BK_FR, -MSK_INFINITY, +MSK_
↪INFINITY);

      /* Set up the objective */
      {
        double c[] = {2.0, 3.0, -1.0};

        if (r == MSK_RES_OK)
          r = MSK_putcslice(task, 0, n, c);
```

(continues on next page)

(continued from previous page)

```
      if (r == MSK_RES_OK)
        r = MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE);
    }

    {
      /* Set AFE rows representing the linear constraint */
      if (r == MSK_RES_OK)
        r = MSK_appendafes(task, 1);
      for(i = 0; i < n && r == MSK_RES_OK; i++)
        r = MSK_putafefentry(task, 0, i, 1.0);
      if (r == MSK_RES_OK)
        r = MSK_putafeg(task, 0, -1.0);
    }

    {
      /* Set AFE rows representing the quadratic constraint */
      /* F matrix data in sparse form */
      MSKint64t Fsubi[] = {2, 2, 3, 3};         /* G is placed from row 2 of F */
      MSKint32t Fsubj[] = {0, 1, 0, 2};
      double    Fval[]  = {1.5, 0.1, 0.3, 2.1};
      int       numEntries = 4;
      /* Other data */
      double h[]    = {0, 0.1};
      double gamma  = 0.03;

      if (r == MSK_RES_OK)
        r = MSK_appendafes(task, k + 1);
      if (r == MSK_RES_OK)
        r = MSK_putafefentrylist(task, numEntries, Fsubi, Fsubj, Fval);
      if (r == MSK_RES_OK)
        r = MSK_putafeg(task, 1, gamma);
      if (r == MSK_RES_OK)
        r = MSK_putafegslice(task, 2, k+2, h);
    }

    /* Define a conic quadratic domain */
    if (r == MSK_RES_OK)
      r = MSK_appendrzerodomain(task, 1, &zeroDom);
    if (r == MSK_RES_OK)
      r = MSK_appendquadraticconedomain(task, k + 1, &quadDom);

    /* Append affine conic constraints */
    {
      /* Linear constraint */
      MSKint64t afeidx[] = {0};

      if (r == MSK_RES_OK)
        r = MSK_appendacc(task,
                          zeroDom,    /* Domain index */
                          1,          /* Dimension */
                          afeidx,     /* Indices of AFE rows */
                          NULL);      /* Ignored */
    }

    {
      /* Quadratic constraint */
```

(continues on next page)

(continued from previous page)

```
      MSKint64t afeidx[] = {1, 2, 3};

      if (r == MSK_RES_OK)
        r = MSK_appendacc(task,
                          quadDom,    /* Domain index */
                          k + 1,      /* Dimension */
                          afeidx,     /* Indices of AFE rows */
                          NULL);      /* Ignored */
    }


    /* Begin optimization and fetching the solution */
    if (r == MSK_RES_OK)
    {
      MSKrescodee trmcode;

      /* Run optimizer */
      r = MSK_optimizetrm(task, &trmcode);

      /* Print a summary containing information
         about the solution for debugging purposes*/
      MSK_solutionsummary(task, MSK_STREAM_MSG);

      if (r == MSK_RES_OK)
      {
        MSKsolstae solsta;

        MSK_getsolsta(task, MSK_SOL_ITR, &solsta);
        MSK_getsolsta(task, MSK_SOL_ITR, &solsta);

        switch (solsta)
        {
          case MSK_SOL_STA_OPTIMAL:
            {
              double *xx, *doty, *activity = NULL;

              /* Fetch the primal solution */
              xx = calloc(n, sizeof(double));
              MSK_getxx(task,
                        MSK_SOL_ITR,    /* Request the interior solution. */
                        xx);

              printf("Optimal primal solution\n");
              for (j = 0; j < n; ++j)
                printf("x[%d]: %e\n", j, xx[j]);

              free(xx);

              /* Fetch the dual doty solution for the ACC */
              doty = calloc(k + 1, sizeof(double));
              MSK_getaccdoty(task,
                             MSK_SOL_ITR,    /* Request the interior solution. */
                             1,              /* ACC index of quadratic ACC. */
                             doty);

              printf("Dual doty of the quadratic ACC\n");
```

(continues on next page)

(continued from previous page)

```
                for (j = 0; j < k + 1; ++j)
                  printf("doty[%d]: %e\n", j, doty[j]);

                free(doty);

                /* Fetch the activity of the ACC */
                activity = calloc(k + 1, sizeof(double));
                MSK_evaluateacc(task,
                                MSK_SOL_ITR,    /* Request the interior solution. */
                                0,              /* ACC index. */
                                activity);

                printf("Activity of the ACC\n");
                for (j = 0; j < k + 1; ++j)
                  printf("activity[%d]: %e\n", j, activity[j]);

                free(activity);
              }
              break;
            case MSK_SOL_STA_DUAL_INFEAS_CER:
            case MSK_SOL_STA_PRIM_INFEAS_CER:
              printf("Primal or dual infeasibility certificate found.\n");
              break;
            case MSK_SOL_STA_UNKNOWN:
              printf("The status of the solution could not be determined. Termination␣
↪code: %d.\n", trmcode);
              break;
            default:
              printf("Other solution status.");
              break;
          }
        }
        else
        {
          printf("Error while optimizing.\n");
        }
      }

      if (r != MSK_RES_OK)
      {
        /* In case of an error print error code and description. */
        char symname[MSK_MAX_STR_LEN];
        char desc[MSK_MAX_STR_LEN];

        printf("An error occurred while optimizing.\n");
        MSK_getcodedesc(r,
                        symname,
                        desc);
        printf("Error %s - '%s'\n", symname, desc);
      }
    }
    /* Delete the task and the associated data. */
    MSK_deletetask(&task);
  }

  /* Delete the environment and the associated data. */
```

(continues on next page)

(continued from previous page)

```
  MSK_deleteenv(&env);

  return (r);
} /* main */
```

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 *MSK_appendaccseq* and *MSK_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 *MSK_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. 15.10 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 **F** and vector **g** appearing in (6.11). The matrix **F** is sparse and we input only its nonzeros using *MSK_putafefentrylist*. Since **g** is zero, nothing needs to be done about this vector.

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

```
        /* Set the non-zero entries of the F matrix */
        r = MSK_putafefentrylist(task, f_nnz, afeidx, varidx, f_val);

        /* Append quadratic cone domain */
        if (r == MSK_RES_OK)
          r = MSK_appendquadraticconedomain(task, 3, domidx);
        /* Append rotated quadratic cone domain */
        if (r == MSK_RES_OK)
          r = MSK_appendrquadraticconedomain(task, 3, domidx+1);

        /* Append two ACCs made up of the AFEs and the domains defined above. */
        if (r == MSK_RES_OK)
          r = MSK_appendaccsseq(task, numacc, domidx, numafe, afeidx[0], NULL);
```

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 **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 **F** and vector **g** see Sec. 6.2.

Source code

Listing 6.4: Source code solving problem (6.10).

```
#include <stdio.h>
#include "mosek.h" /* Include the MOSEK definition file. */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{
  MSKrescodee  r;

  const MSKint32t numvar = 6,
                  numcon = 1;
  const MSKint64t numafe = 6,
                  numacc = 2,
```

(continues on next page)

(continued from previous page)

```
                 f_nnz  = 6;

MSKboundkeye bkc[] = { MSK_BK_FX };
double       blc[] = { 1.0 };
double       buc[] = { 1.0 };

MSKboundkeye bkx[] = {MSK_BK_LO,
                      MSK_BK_LO,
                      MSK_BK_LO,
                      MSK_BK_FR,
                      MSK_BK_FR,
                      MSK_BK_FR
                     };
double       blx[] = {0.0,
                      0.0,
                      0.0,
                      -MSK_INFINITY,
                      -MSK_INFINITY,
                      -MSK_INFINITY
                     };
double       bux[] = { +MSK_INFINITY,
                       +MSK_INFINITY,
                       +MSK_INFINITY,
                       +MSK_INFINITY,
                       +MSK_INFINITY,
                       +MSK_INFINITY
                     };

double       c[]   = {0.0,
                      0.0,
                      0.0,
                      1.0,
                      1.0,
                      1.0
                     };

MSKint32t   aptrb[] = {0, 1, 2, 3, 3, 3},
            aptre[] = {1, 2, 3, 3, 3, 3},
            asub[]  = {0, 0, 0, 0};
double      aval[]  = {1.0, 1.0, 2.0};

MSKint64t   afeidx[] = {0, 1, 2, 3, 4, 5};
MSKint32t   varidx[] = {3, 0, 1, 4, 5, 2};
MSKrealt    f_val[]  = {1, 1, 1, 1, 1, 1};

MSKint64t   domidx[] = {0, 0};

MSKint32t   i, j, csub[3];

MSKenv_t    env  = NULL;
MSKtask_t   task = NULL;

/* Create the mosek environment. */
r = MSK_makeenv(&env, NULL);

if (r == MSK_RES_OK)
```

(continues on next page)

(continued from previous page)

```
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, numcon, numvar, &task);

    if (r == MSK_RES_OK)
    {
      MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

      /* Append 'numcon' empty constraints.
      The constraints will initially have no bounds. */
      if (r == MSK_RES_OK)
        r = MSK_appendcons(task, numcon);

      /* Append 'numvar' variables.
      The variables will initially be fixed at zero (x=0). */
      if (r == MSK_RES_OK)
        r = MSK_appendvars(task, numvar);

      /* Append 'numafe' affine expressions.
      The affine expressions will initially be empty. */
      if (r == MSK_RES_OK)
        r = MSK_appendafes(task, numafe);


      for (j = 0; j < numvar && r == MSK_RES_OK; ++j)
      {
        /* Set the linear term c_j in the objective.*/
        if (r == MSK_RES_OK)
          r = MSK_putcj(task, j, c[j]);

        /* Set the bounds on variable j.
        blx[j] <= x_j <= bux[j] */
        if (r == MSK_RES_OK)
          r = MSK_putvarbound(task,
                              j,           /* Index of variable.*/
                              bkx[j],      /* Bound key.*/
                              blx[j],      /* Numerical value of lower bound.*/
                              bux[j]);     /* Numerical value of upper bound.*/

        /* Input column j of A */
        if (r == MSK_RES_OK)
          r = MSK_putacol(task,
                          j,                   /* Variable (column) index.*/
                          aptre[j] - aptrb[j], /* Number of non-zeros in column j.*/
                          asub + aptrb[j],     /* Pointer to row indexes of column j.*/
                          aval + aptrb[j]);    /* Pointer to Values of column j.*/

      }

      /* Set the bounds on constraints.
       for i=1, ...,numcon : blc[i] <= constraint i <= buc[i] */
      for (i = 0; i < numcon && r == MSK_RES_OK; ++i)
        r = MSK_putconbound(task,
                            i,           /* Index of constraint.*/
                            bkc[i],      /* Bound key.*/
                            blc[i],      /* Numerical value of lower bound.*/
```

(continues on next page)

(continued from previous page)

```
                         buc[i]);     /* Numerical value of upper bound.*/

    if (r == MSK_RES_OK)
    {
      /* Set the non-zero entries of the F matrix */
      r = MSK_putafefentrylist(task, f_nnz, afeidx, varidx, f_val);

      /* Append quadratic cone domain */
      if (r == MSK_RES_OK)
        r = MSK_appendquadraticconedomain(task, 3, domidx);
      /* Append rotated quadratic cone domain */
      if (r == MSK_RES_OK)
        r = MSK_appendrquadraticconedomain(task, 3, domidx+1);

      /* Append two ACCs made up of the AFEs and the domains defined above. */
      if (r == MSK_RES_OK)
        r = MSK_appendaccsseq(task, numacc, domidx, numafe, afeidx[0], NULL);
    }

    if (r == MSK_RES_OK)
    {
      MSKrescodee trmcode;

      /* Run optimizer */
      r = MSK_optimizetrm(task, &trmcode);


      /* Print a summary containing information
         about the solution for debugging purposes*/
      MSK_solutionsummary(task, MSK_STREAM_MSG);

      if (r == MSK_RES_OK)
      {
        MSKsolstae solsta;

        MSK_getsolsta(task, MSK_SOL_ITR, &solsta);

        switch (solsta)
        {
          case MSK_SOL_STA_OPTIMAL:
            {
              double *xx = NULL;

              xx = calloc(numvar, sizeof(double));
              if (xx)
              {
                MSK_getxx(task,
                          MSK_SOL_ITR,    /* Request the interior solution. */
                          xx);

                printf("Optimal primal solution\n");
                for (j = 0; j < numvar; ++j)
                  printf("x[%d]: %e\n", j, xx[j]);
              }
              else
              {
```

(continues on next page)

(continued from previous page)

```
                r = MSK_RES_ERR_SPACE;
              }
              free(xx);
            }
            break;
          case MSK_SOL_STA_DUAL_INFEAS_CER:
          case MSK_SOL_STA_PRIM_INFEAS_CER:
            printf("Primal or dual infeasibility certificate found.\n");
            break;
          case MSK_SOL_STA_UNKNOWN:
            printf("The status of the solution could not be determined. Termination␣
↪code: %d.\n", trmcode);
            break;
          default:
            printf("Other solution status.");
            break;
        }
      }
      else
      {
        printf("Error while optimizing.\n");
      }
    }

    if (r != MSK_RES_OK)
    {
      /* In case of an error print error code and description. */
      char symname[MSK_MAX_STR_LEN];
      char desc[MSK_MAX_STR_LEN];

      printf("An error occurred while optimizing.\n");
      MSK_getcodedesc(r,
                      symname,
                      desc);
      printf("Error %s - '%s'\n", symname, desc);
    }
  }
  /* Delete the task and the associated data. */
  MSK_deletetask(&task);
}

/* Delete the environment and the associated data. */
MSK_deleteenv(&env);

return (r);
} /* main */
```

6.4 Power Cone Optimization

The structure of a typical conic optimization problem is

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

(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 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. 15.10 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:

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

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

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

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 **F** and vector **g** which together determine all the six affine expressions appearing in the conic constraints of (6.13)

```
        /* Set the non-zero entries of the F matrix */
        r = MSK_putafefentrylist(task, f_nnz, afeidx, varidx, f_val);
        /* Set the non-zero element of the g vector */
        if (r == MSK_RES_OK)
          r = MSK_putafeg(task, 4, g);

        /* Append the primal power cone domains with their respective parameter␣
↪values. */
        if (r == MSK_RES_OK)
          r = MSK_appendprimalpowerconedomain(task, 3, 2, alpha_1, domidx);
        if (r == MSK_RES_OK)
          r = MSK_appendprimalpowerconedomain(task, 3, 2, alpha_2, domidx+1);

        /* Append two ACCs made up of the AFEs and the domains defined above. */
        if (r == MSK_RES_OK)
          r = MSK_appendaccsseq(task, numacc, domidx, numafe, afeidx[0], NULL);
```

Following that, each of the affine conic constraints is appended using the function *MSK_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 **F** and **g** while in the second call we use the remaining three rows of **F** and **g**.

Variants of this method are available to append multiple ACCs at a time. It is also possible to define the matrix **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 **F** and vector **g** see Sec. 6.2.

Source code

Listing 6.5: Source code solving problem (6.12).

```
#include <stdio.h>
#include "mosek.h" /* Include the MOSEK definition file. */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
```

(continues on next page)

(continued from previous page)

```
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{
  MSKrescodee  r;

  const MSKint32t numvar = 5,
                  numcon = 1;
  const MSKint64t numafe = 6,
                  numacc = 2,
                  f_nnz  = 5;

  MSKboundkeye bkx[5];
  double       blx[5], bux[5];

  double       val[] = { 1.0, 1.0, -1.0 };
  MSKint32t    sub[] = { 3, 4, 0 };

  double       aval[] = { 1.0, 1.0, 0.5 };
  MSKint32t    asub[] = { 0, 1, 2 };

  MSKint64t  afeidx[] = {0, 1, 2, 3, 5};
  MSKint32t  varidx[] = {0, 1, 3, 2, 4};
  MSKrealt    f_val[] = {1, 1, 1, 1, 1},
                  g = 1;

  const MSKrealt alpha_1[]= {0.2, 0.8},
                 alpha_2[]= {0.4, 0.6};

  MSKint64t   domidx[] = {0, 0};

  MSKint32t   i, j;

  MSKenv_t    env  = NULL;
  MSKtask_t   task = NULL;

  /* Create the mosek environment. */
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, numcon, numvar, &task);

    if (r == MSK_RES_OK)
    {
      MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

      /* Append 'numcon' empty constraints.
      The constraints will initially have no bounds. */
      if (r == MSK_RES_OK)
        r = MSK_appendcons(task, numcon);

      /* Append 'numvar' variables.
      The variables will initially be fixed at zero (x=0). */
```

(continues on next page)

(continued from previous page)

```
      if (r == MSK_RES_OK)
        r = MSK_appendvars(task, numvar);

      /* Append 'numafe' affine expressions.
      The affine expressions will initially be empty. */
      if (r == MSK_RES_OK)
        r = MSK_appendafes(task, numafe);


      /* Set up the linear part */
      MSK_putclist(task, 3, sub, val);
      MSK_putarow(task, 0, 3, asub, aval);
      MSK_putconbound(task, 0, MSK_BK_FX, 2.0, 2.0);
      for(i=0;i<5;i++)
        bkx[i] = MSK_BK_FR, blx[i] = -MSK_INFINITY, bux[i] = MSK_INFINITY;
      MSK_putvarboundslice(task, 0, numvar, bkx, blx, bux);

      if (r == MSK_RES_OK)
      {
        /* Set the non-zero entries of the F matrix */
        r = MSK_putafefentrylist(task, f_nnz, afeidx, varidx, f_val);
        /* Set the non-zero element of the g vector */
        if (r == MSK_RES_OK)
          r = MSK_putafeg(task, 4, g);

        /* Append the primal power cone domains with their respective parameter␣
↪values. */
        if (r == MSK_RES_OK)
          r = MSK_appendprimalpowerconedomain(task, 3, 2, alpha_1, domidx);
        if (r == MSK_RES_OK)
          r = MSK_appendprimalpowerconedomain(task, 3, 2, alpha_2, domidx+1);

        /* Append two ACCs made up of the AFEs and the domains defined above. */
        if (r == MSK_RES_OK)
          r = MSK_appendaccsseq(task, numacc, domidx, numafe, afeidx[0], NULL);
      }

      MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE);

      if (r == MSK_RES_OK)
      {
        MSKrescodee trmcode;

        /* Run optimizer */
        r = MSK_optimizetrm(task, &trmcode);

        /* Print a summary containing information
           about the solution for debugging purposes*/
        MSK_solutionsummary(task, MSK_STREAM_MSG);

        if (r == MSK_RES_OK)
        {
          MSKsolstae solsta;

          MSK_getsolsta(task, MSK_SOL_ITR, &solsta);
```

(continues on next page)

(continued from previous page)

```
          switch (solsta)
          {
            case MSK_SOL_STA_OPTIMAL:
              {
                double *xx = NULL;

                xx = calloc(numvar, sizeof(double));
                if (xx)
                {
                  MSK_getxx(task,
                            MSK_SOL_ITR,    /* Request the interior solution. */
                            xx);

                  printf("Optimal primal solution\n");
                  for (j = 0; j < 3; ++j)
                    printf("x[%d]: %e\n", j, xx[j]);
                }
                else
                {
                  r = MSK_RES_ERR_SPACE;
                }
                free(xx);
              }
              break;
            case MSK_SOL_STA_DUAL_INFEAS_CER:
            case MSK_SOL_STA_PRIM_INFEAS_CER:
              printf("Primal or dual infeasibility certificate found.\n");
              break;
            case MSK_SOL_STA_UNKNOWN:
              printf("The status of the solution could not be determined. Termination␣
↪code: %d.\n", trmcode);
              break;
            default:
              printf("Other solution status.");
              break;
          }
        }
        else
        {
          printf("Error while optimizing.\n");
        }
      }

      if (r != MSK_RES_OK)
      {
        /* In case of an error print error code and description. */
        char symname[MSK_MAX_STR_LEN];
        char desc[MSK_MAX_STR_LEN];

        printf("An error occurred while optimizing.\n");
        MSK_getcodedesc(r,
                        symname,
                        desc);
        printf("Error %s - '%s'\n", symname, desc);
      }
    }
```

(continues on next page)

(continued from previous page)

```
    /* Delete the task and the associated data. */
    MSK_deletetask(&task);
  }

  /* Delete the environment and the associated data. */
  MSK_deleteenv(&env);

  return (r);
} /* main */
```

6.5 Conic Exponential Optimization

The structure of a typical conic optimization problem is

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

(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:

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

- Dual exponential cone:

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

For example, consider the following constraint:

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

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. 15.10 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:

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

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 *MSK_appendacc*, with the primal exponential domain and the list of $\mathbf{F}$ rows, in this case consisting of all rows in their natural order.

```
        r = MSK_putafefentrylist(task, f_nnz, afeidx, varidx, f_val);

        if (r == MSK_RES_OK)
          r = MSK_appendprimalexpconedomain(task, &domidx);

        if (r == MSK_RES_OK)
          r = MSK_appendaccseq(task, domidx, numafe, 0, NULL);
```

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).

```
#include <stdio.h>
#include "mosek.h" /* Include the MOSEK definition file. */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{
  MSKrescodee  r;

  const MSKint32t numvar = 3,
                  numcon = 1;
  const MSKint64t numafe = 3,
                  f_nnz  = 3;

  MSKboundkeye bkc = MSK_BK_FX;
  double       blc = 1.0;
  double       buc = 1.0;

  MSKboundkeye bkx[] = {MSK_BK_FR,
                        MSK_BK_FR,
                        MSK_BK_FR
```

(continues on next page)

(continued from previous page)

```
                        };
  double       blx[] = {-MSK_INFINITY,
                         -MSK_INFINITY,
                         -MSK_INFINITY
                        };
  double       bux[] = { +MSK_INFINITY,
                          +MSK_INFINITY,
                          +MSK_INFINITY
                        };

  double       c[]   = {1.0,
                         1.0,
                         0.0
                        };

  double    a[]     = {1.0, 1.0, 1.0};
  MSKint32t asub[]  = {0, 1, 2};

  MSKint64t   afeidx[] = {0, 1, 2};
  MSKint32t   varidx[] = {0, 1, 2};
  MSKrealt    f_val[]  = {1, 1, 1};

  MSKint64t   domidx = 0;

  MSKint32t   i, j, csub[3];

  MSKenv_t    env  = NULL;
  MSKtask_t   task = NULL;

  /* Create the mosek environment. */
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, numcon, numvar, &task);

    if (r == MSK_RES_OK)
    {
      MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

      /* Append 'numcon' empty constraints.
      The constraints will initially have no bounds. */
      if (r == MSK_RES_OK)
        r = MSK_appendcons(task, numcon);

      /* Append 'numvar' variables.
      The variables will initially be fixed at zero (x=0). */
      if (r == MSK_RES_OK)
        r = MSK_appendvars(task, numvar);

      /* Append 'numafe' affine expressions.
      The affine expressions will initially be empty. */
      if (r == MSK_RES_OK)
        r = MSK_appendafes(task, numafe);
```

(continues on next page)

(continued from previous page)

```
    /* Set up the linear part */
    if (r == MSK_RES_OK)
      r = MSK_putcslice(task, 0, numvar, c);
    if (r == MSK_RES_OK)
      r = MSK_putarow(task, 0, numvar, asub, a);
    if (r == MSK_RES_OK)
      r = MSK_putconbound(task, 0, bkc, blc, buc);
    if (r == MSK_RES_OK)
      r = MSK_putvarboundslice(task, 0, numvar, bkx, blx, bux);

    if (r == MSK_RES_OK)
    {
      r = MSK_putafefentrylist(task, f_nnz, afeidx, varidx, f_val);

      if (r == MSK_RES_OK)
        r = MSK_appendprimalexpconedomain(task, &domidx);

      if (r == MSK_RES_OK)
        r = MSK_appendaccseq(task, domidx, numafe, 0, NULL);
    }

    if (r == MSK_RES_OK)
    {
      MSKrescodee trmcode;

      /* Run optimizer */
      r = MSK_optimizetrm(task, &trmcode);

      /* Print a summary containing information
         about the solution for debugging purposes*/
      MSK_solutionsummary(task, MSK_STREAM_MSG);

      if (r == MSK_RES_OK)
      {
        MSKsolstae solsta;

        MSK_getsolsta(task, MSK_SOL_ITR, &solsta);

        switch (solsta)
        {
          case MSK_SOL_STA_OPTIMAL:
            {
              double *xx = NULL;

              xx = calloc(numvar, sizeof(double));
              if (xx)
              {
                MSK_getxx(task,
                          MSK_SOL_ITR,    /* Request the interior solution. */
                          xx);

                printf("Optimal primal solution\n");
                for (j = 0; j < numvar; ++j)
                  printf("x[%d]: %e\n", j, xx[j]);
              }
```

(continues on next page)

(continued from previous page)

```
              else
              {
                r = MSK_RES_ERR_SPACE;
              }
              free(xx);
            }
            break;
          case MSK_SOL_STA_DUAL_INFEAS_CER:
          case MSK_SOL_STA_PRIM_INFEAS_CER:
            printf("Primal or dual infeasibility certificate found.\n");
            break;
          case MSK_SOL_STA_UNKNOWN:
            printf("The status of the solution could not be determined. Termination␣
↪code: %d.\n", trmcode);
            break;
          default:
            printf("Other solution status.");
            break;
        }
      }
      else
      {
        printf("Error while optimizing.\n");
      }
    }

    if (r != MSK_RES_OK)
    {
      /* In case of an error print error code and description. */
      char symname[MSK_MAX_STR_LEN];
      char desc[MSK_MAX_STR_LEN];

      printf("An error occurred while optimizing.\n");
      MSK_getcodedesc(r,
                      symname,
                      desc);
      printf("Error %s - '%s'\n", symname, desc);
    }
  }
  /* Delete the task and the associated data. */
  MSK_deletetask(&task);
}

/* Delete the environment and the associated data. */
MSK_deleteenv(&env);

return (r);
} /* main */
```

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:

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

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:

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

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).

```
    MSKint64t acc1_afeidx[] = {0, 4, 2};
    MSKint64t acc2_afeidx[] = {1, 4, 3};
    MSKint64t acc3_afeidx[] = {5};

    // 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
    if (r == MSK_RES_OK)
      r = MSK_appendvars(task, 2);
    if (r == MSK_RES_OK)
      r = MSK_putvarboundsliceconst(task, numvar, numvar+2, MSK_BK_FR, -MSK_INFINITY,␣
↪+MSK_INFINITY);

    if (r == MSK_RES_OK)
      r = MSK_putafefentrylist(task, f_nnz, afeidx, varidx, f_val);
    if (r == MSK_RES_OK)
      r = MSK_putafegslice(task, 0, numafe, g);

    /* Append the primal exponential cone domain */
    if (r == MSK_RES_OK)
      r = MSK_appendprimalexpconedomain(task, &expdomidx);

    /* (u1, 1, x+y+log(2/Awall)) \in EXP */
    if (r == MSK_RES_OK)
      r = MSK_appendacc(task, expdomidx, 3, acc1_afeidx, NULL);

    /* (u2, 1, x+z+log(2/Awall)) \in EXP */
    if (r == MSK_RES_OK)
      r = MSK_appendacc(task, expdomidx, 3, acc2_afeidx, NULL);

    /* The constraint u1+u2-1 \in \ZERO is added also as an ACC */
    if (r == MSK_RES_OK)
      r = MSK_appendrzerodomain(task, 1, &rzerodomidx);
    if (r == MSK_RES_OK)
      r = MSK_appendacc(task, rzerodomidx, 1, acc3_afeidx, NULL);
```

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).

```
int max_volume_box(double Aw, double Af,
                   double alpha, double beta, double gamma, double delta,
                   double hwd[])
{
  // Basic dimensions of our problem
  const int numvar    = 3;  // Variables in original problem
  const int numcon    = 3;  // Linear constraints in original problem

  // Linear part of the problem involving x, y, z
  const double       cval[]  = {1, 1, 1};
  const int          asubi[] = {0, 0, 1, 1, 2, 2};
  const int          asubj[] = {1, 2, 0, 1, 2, 1};
  const int          alen    = 6;
  const double       aval[]  = {1.0, 1.0, 1.0, -1.0, 1.0, -1.0};
  const MSKboundkeye bkc[]   = {MSK_BK_UP, MSK_BK_RA, MSK_BK_RA};
  const double       blc[]   = {-MSK_INFINITY, log(alpha), log(gamma)};
  const double       buc[]   = {log(Af), log(beta), log(delta)};

  // Affine conic constraint data of the problem
  MSKint64t       expdomidx, rzerodomidx;
  const MSKint64t numafe = 6, f_nnz = 8;
  const MSKint64t afeidx[] = {0, 1, 2, 2, 3, 3, 5, 5};
  const MSKint32t varidx[] = {3, 4, 0, 1, 0, 2, 3, 4};
  const double     f_val[] = {1, 1, 1, 1, 1, 1, 1, 1};
  const double         g[] = {0, 0, log(2/Aw), log(2/Aw), 1, -1};

  MSKtask_t          task = NULL;
  MSKrescodee        r = MSK_RES_OK, trmcode;
  MSKsolstae         solsta;
  MSKint32t          i;
  double             *xyz = (double*) calloc(numvar, sizeof(double));

  if (r == MSK_RES_OK)
    r = MSK_maketask(NULL, 0, 0, &task);

  if (r == MSK_RES_OK)
    r = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

  if (r == MSK_RES_OK)
    r = MSK_appendvars(task, numvar);

  if (r == MSK_RES_OK)
    r = MSK_appendcons(task, numcon);

  if (r == MSK_RES_OK)
    r = MSK_appendafes(task, numafe);

  // Objective is the sum of three first variables
  if (r == MSK_RES_OK)
    r = MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE);
  if (r == MSK_RES_OK)
    r = MSK_putcslice(task, 0, numvar, cval);
  if (r == MSK_RES_OK)
```

(continues on next page)

(continued from previous page)

```
    r = MSK_putvarboundsliceconst(task, 0, numvar, MSK_BK_FR, -MSK_INFINITY, +MSK_
↪INFINITY);

  // Add the three linear constraints
  if (r == MSK_RES_OK)
    r = MSK_putaijlist(task, alen, asubi, asubj, aval);
  if (r == MSK_RES_OK)
    r = MSK_putconboundslice(task, 0, numvar, bkc, blc, buc);

  if (r == MSK_RES_OK)
  {
    MSKint64t acc1_afeidx[] = {0, 4, 2};
    MSKint64t acc2_afeidx[] = {1, 4, 3};
    MSKint64t acc3_afeidx[] = {5};

    // 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
    if (r == MSK_RES_OK)
      r = MSK_appendvars(task, 2);
    if (r == MSK_RES_OK)
      r = MSK_putvarboundsliceconst(task, numvar, numvar+2, MSK_BK_FR, -MSK_INFINITY,␣
↪+MSK_INFINITY);

    if (r == MSK_RES_OK)
      r = MSK_putafefentrylist(task, f_nnz, afeidx, varidx, f_val);
    if (r == MSK_RES_OK)
      r = MSK_putafegslice(task, 0, numafe, g);

    /* Append the primal exponential cone domain */
    if (r == MSK_RES_OK)
      r = MSK_appendprimalexpconedomain(task, &expdomidx);

    /* (u1, 1, x+y+log(2/Awall)) \in EXP */
    if (r == MSK_RES_OK)
      r = MSK_appendacc(task, expdomidx, 3, acc1_afeidx, NULL);

    /* (u2, 1, x+z+log(2/Awall)) \in EXP */
    if (r == MSK_RES_OK)
      r = MSK_appendacc(task, expdomidx, 3, acc2_afeidx, NULL);

    /* The constraint u1+u2-1 \in \ZERO is added also as an ACC */
    if (r == MSK_RES_OK)
      r = MSK_appendrzerodomain(task, 1, &rzerodomidx);
    if (r == MSK_RES_OK)
      r = MSK_appendacc(task, rzerodomidx, 1, acc3_afeidx, NULL);
  }

  // Solve and map to original h, w, d
  if (r == MSK_RES_OK)
    r = MSK_optimizetrm(task, &trmcode);

  if (r == MSK_RES_OK)
    MSK_getsolsta(task, MSK_SOL_ITR, &solsta);

  if (solsta == MSK_SOL_STA_OPTIMAL)
```

(continues on next page)

(continued from previous page)

```
  {
    if (r == MSK_RES_OK)
      r = MSK_getxxslice(task, MSK_SOL_ITR, 0, numvar, xyz);
    for(i = 0; i < numvar; i++) hwd[i] = exp(xyz[i]);
  }
  else
  {
    printf("Solution not optimal, termination code %d.\n", trmcode);
    r = trmcode;
  }

  free(xyz);
  return r;
}
```

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

Listing 6.9: Sample data for problem (6.19).

```
int main()
{
  const double Aw    = 200.0;
  const double Af    = 50.0;
  const double alpha = 2.0;
  const double beta  = 10.0;
  const double gamma = 2.0;
  const double delta = 10.0;
  MSKrescodee  r;
  double       hwd[3];

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

  printf("Response code: %d\n", r);
  if (r == MSK_RES_OK)
    printf("Solution h=%.4f w=%.4f d=%.4f\n", hwd[0], hwd[1], hwd[2]);

  return r;
}
```

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,

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

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:

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

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

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

  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. LMIs can be expressed by restricting appropriate affine expressions to this cone type.

For other types of cones supported by **MOSEK**, see Sec. 15.10 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:

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

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 *MSK_appendbarvars*.

```
        r = MSK_appendbarvars(task, NUMBARVAR, DIMBARVAR);
```

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 *MSK_appendsparsesymmat*.

```
        r = MSK_appendsparsesymmat(task,
                                   DIMBARVAR[0],
                                   5,
                                   barc_i,
                                   barc_j,
                                   barc_v,
                                   &idx);
```

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 *MSK_appendsparsesymmat*, we can combine them to set up the objective matrix coefficient $\overline{C}_j$ using *MSK_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.

```
        r = MSK_putbarcj(task, 0, 1, &idx, &falpha);
```

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

```
        r = MSK_putbaraij(task, 0, 0, 1, &idx, &falpha);
```

Retrieving the solution

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

```
              MSK_getbarxj(task,
                           MSK_SOL_ITR,    /* Request the interior solution. */
                           0,
                           barx);
```

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).

```
#include <stdio.h>

#include "mosek.h"    /* Include the MOSEK definition file.  */

#define NUMCON    2   /* Number of constraints.              */
#define NUMVAR    3   /* Number of conic quadratic variables */
#define NUMANZ    3   /* Number of non-zeros in A            */
#define NUMAFE    3   /* Number of affine expressions        */
#define NUMFNZ    3   /* Number of non-zeros in F            */
#define NUMBARVAR 1   /* Number of semidefinite variables    */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{
  MSKrescodee  r;

  MSKint32t    DIMBARVAR[] = {3};          /* Dimension of semidefinite cone */
  MSKint64t    LENBARVAR[] = {3 * (3 + 1) / 2}; /* Number of scalar SD variables  */

  MSKboundkeye bkc[] = { MSK_BK_FX, MSK_BK_FX };
  double       blc[] = { 1.0, 0.5 };
  double       buc[] = { 1.0, 0.5 };

  MSKint32t    barc_i[] = {0, 1, 1, 2, 2},
               barc_j[] = {0, 0, 1, 1, 2};
  double       barc_v[] = {2.0, 1.0, 2.0, 1.0, 2.0};

  MSKint32t    aptrb[]  = {0, 1},
               aptre[]  = {1, 3},
               asub[]   = {0, 1, 2}; /* column subscripts of A */
  double       aval[]   = {1.0, 1.0, 1.0};

  MSKint32t    bara_i[] = {0, 1, 2, 0, 1, 2, 1, 2, 2},
               bara_j[] = {0, 1, 2, 0, 0, 0, 1, 1, 2};
  double       bara_v[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
  MSKint32t    conesub[] = {0, 1, 2};
```

(continues on next page)

(continued from previous page)

```
  MSKint64t    afeidx[] = {0, 1, 2};
  int          varidx[] = {0, 1, 2};
  double       f_val[]  = {1, 1, 1};

  MSKint32t    i, j;
  MSKint64t    idx;
  double       falpha = 1.0;

  MSKrealt     *xx;
  MSKrealt     *barx;
  MSKenv_t     env = NULL;
  MSKtask_t    task = NULL;

  /* Create the mosek environment. */
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, NUMCON, 0, &task);

    if (r == MSK_RES_OK)
    {
      MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

      /* Append 'NUMCON' empty constraints.
       The constraints will initially have no bounds. */
      if (r == MSK_RES_OK)
        r = MSK_appendcons(task, NUMCON);

      /* Append 'NUMVAR' variables.
      The variables will initially be fixed at zero (x=0). */
      if (r == MSK_RES_OK)
        r = MSK_appendvars(task, NUMVAR);

      /* Append 'NUMAFE' affine expressions.*/
      if (r == MSK_RES_OK)
        r = MSK_appendafes(task, NUMAFE);

      /* Append 'NUMBARVAR' semidefinite variables. */
      if (r == MSK_RES_OK) {
        r = MSK_appendbarvars(task, NUMBARVAR, DIMBARVAR);
      }

      /* Optionally add a constant term to the objective. */
      if (r == MSK_RES_OK)
        r = MSK_putcfix(task, 0.0);

      /* Set the linear term c_j in the objective.*/
      if (r == MSK_RES_OK)
        r = MSK_putcj(task, 0, 1.0);

      for (j = 0; j < NUMVAR && r == MSK_RES_OK; ++j)
        r = MSK_putvarbound(task,
                            j,
                            MSK_BK_FR,
```

(continues on next page)

(continued from previous page)

```
                        -MSK_INFINITY,
                        MSK_INFINITY);

      /* Set the linear term barc_j in the objective.*/
      if (r == MSK_RES_OK)
        r = MSK_appendsparsesymmat(task,
                                   DIMBARVAR[0],
                                   5,
                                   barc_i,
                                   barc_j,
                                   barc_v,
                                   &idx);

      if (r == MSK_RES_OK)
        r = MSK_putbarcj(task, 0, 1, &idx, &falpha);

      /* Set the bounds on constraints.
        for i=1, ...,NUMCON : blc[i] <= constraint i <= buc[i] */
      for (i = 0; i < NUMCON && r == MSK_RES_OK; ++i)
        r = MSK_putconbound(task,
                            i,           /* Index of constraint.*/
                            bkc[i],      /* Bound key.*/
                            blc[i],      /* Numerical value of lower bound.*/
                            buc[i]);     /* Numerical value of upper bound.*/

      /* Input A row by row */
      for (i = 0; i < NUMCON && r == MSK_RES_OK; ++i)
        r = MSK_putarow(task,
                        i,
                        aptre[i] - aptrb[i],
                        asub     + aptrb[i],
                        aval     + aptrb[i]);

      /* Append the affine conic constraint with quadratic cone */
      if (r == MSK_RES_OK)
      {
        r = MSK_putafefentrylist(task, NUMFNZ, afeidx, varidx, f_val);
        if (r == MSK_RES_OK)
          r = MSK_appendquadraticconedomain(task, 3, NULL);
        if (r == MSK_RES_OK)
          r = MSK_appendacc(task, 0, 3, afeidx, NULL);
      }

      /* Add the first row of barA */
      if (r == MSK_RES_OK)
        r = MSK_appendsparsesymmat(task,
                                   DIMBARVAR[0],
                                   3,
                                   bara_i,
                                   bara_j,
                                   bara_v,
                                   &idx);

      if (r == MSK_RES_OK)
        r = MSK_putbaraij(task, 0, 0, 1, &idx, &falpha);
```

(continues on next page)

(continued from previous page)

```
      /* Add the second row of barA */
      if (r == MSK_RES_OK)
        r = MSK_appendsparsesymmat(task,
                                   DIMBARVAR[0],
                                   6,
                                   bara_i + 3,
                                   bara_j + 3,
                                   bara_v + 3,
                                   &idx);

      if (r == MSK_RES_OK)
        r = MSK_putbaraij(task, 1, 0, 1, &idx, &falpha);

      if (r == MSK_RES_OK)
      {
        MSKrescodee trmcode;

        /* Run optimizer */
        r = MSK_optimizetrm(task, &trmcode);

        /* Print a summary containing information
           about the solution for debugging purposes*/
        MSK_solutionsummary(task, MSK_STREAM_MSG);

        if (r == MSK_RES_OK)
        {
          MSKsolstae solsta;

          MSK_getsolsta(task, MSK_SOL_ITR, &solsta);

          switch (solsta)
          {
            case MSK_SOL_STA_OPTIMAL:
              xx   = (MSKrealt *) MSK_calloctask(task, NUMVAR, sizeof(MSKrealt));
              barx = (MSKrealt *) MSK_calloctask(task, LENBARVAR[0],␣
↪sizeof(MSKrealt));

              MSK_getxx(task,
                        MSK_SOL_ITR,
                        xx);
              MSK_getbarxj(task,
                           MSK_SOL_ITR,    /* Request the interior solution. */
                           0,
                           barx);

              printf("Optimal primal solution\n");
              for (i = 0; i < NUMVAR; ++i)
                printf("x[%d]   : % e\n", i, xx[i]);

              for (i = 0; i < LENBARVAR[0]; ++i)
                printf("barx[%d]: % e\n", i, barx[i]);

              MSK_freetask(task, xx);
              MSK_freetask(task, barx);

              break;
```

(continues on next page)

(continued from previous page)

```
            case MSK_SOL_STA_DUAL_INFEAS_CER:
            case MSK_SOL_STA_PRIM_INFEAS_CER:
              printf("Primal or dual infeasibility certificate found.\n");
              break;

            case MSK_SOL_STA_UNKNOWN:
              printf("The status of the solution could not be determined. Termination␣
↪code: %d.\n", trmcode);
              break;

            default:
              printf("Other solution status.");
              break;
          }
        }
        else
        {
          printf("Error while optimizing.\n");
        }
      }

      if (r != MSK_RES_OK)
      {
        /* In case of an error print error code and description. */
        char symname[MSK_MAX_STR_LEN];
        char desc[MSK_MAX_STR_LEN];

        printf("An error occurred while optimizing.\n");
        MSK_getcodedesc(r,
                        symname,
                        desc);
        printf("Error %s - '%s'\n", symname, desc);
      }
    }
    /* Delete the task and the associated data. */
    MSK_deletetask(&task);
  }

  /* Delete the environment and the associated data. */
  MSK_deleteenv(&env);

  return (r);
} /* main */
```

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 *MSK_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 *MSK_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).

```
#include <stdio.h>
#include "mosek.h"    /* Include the MOSEK definition file.  */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{
  MSKrescodee  r;

  /* Input data */
  MSKint32t    numbarvar = 2;
  MSKint32t    dimbarvar[] = {3, 4};         /* Dimension of semidefinite variables␣
↪*/

  /* Objective coefficients concatenated */
  MSKint32t    Cj[] = { 0, 0, 1, 1, 1, 1 };   /* Which symmetric variable (j) */
  MSKint32t    Ck[] = { 0, 2, 0, 1, 1, 2 };   /* Which entry (k,l)->v */
  MSKint32t    Cl[] = { 0, 2, 0, 0, 1, 2 };
```

(continues on next page)

(continued from previous page)

```
  MSKrealt     Cv[] = { 1.0, 6.0, 1.0, -3.0, 2.0, 1.0 };

  /* Equality constraints coefficients concatenated */
  MSKint32t    Ai[] = { 0, 0, 0, 0, 0, 0 };   /* Which constraint (i = 0) */
  MSKint32t    Aj[] = { 0, 0, 0, 1, 1, 1 };   /* Which symmetric variable (j) */
  MSKint32t    Ak[] = { 0, 2, 2, 1, 1, 3 };   /* Which entry (k,l)->v */
  MSKint32t    Al[] = { 0, 0, 2, 0, 1, 3 };
  MSKrealt     Av[] = { 1.0, 1.0, 2.0, 1.0, -1.0, -3.0 };

  /* The second constraint - one-term inequality */
  MSKint32t    A2i = 1;                        /* Which constraint (i = 1) */
  MSKint32t    A2j = 1;                        /* Which symmetric variable (j = 1) */
  MSKint32t    A2k = 1;                        /* Which entry A(1,0) = A(0,1) = 0.5 */
↪
  MSKint32t    A2l = 0;
  MSKrealt     A2v = 0.5;

  /* Constraint bounds and values */
  MSKint32t    numcon = 2;
  MSKboundkeye bkc[] = { MSK_BK_FX, MSK_BK_UP };
  double       blc[] = { 23.0, -MSK_INFINITY };
  double       buc[] = { 23.0, -3.0 };

  MSKint32t    i, j, dim;
  MSKrealt     *barx;
  MSKenv_t     env = NULL;
  MSKtask_t    task = NULL;

  /* Create the mosek environment. */
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, 0, 0, &task);

    if (r == MSK_RES_OK)
    {
      MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

      /* Append empty constraints.
       The constraints will initially have no bounds. */
      if (r == MSK_RES_OK)
        r = MSK_appendcons(task, numcon);

      /* Append semidefinite variables. */
      if (r == MSK_RES_OK)
        r = MSK_appendbarvars(task, numbarvar, dimbarvar);

      /* Set objective (6 nonzeros).*/
      if (r == MSK_RES_OK)
        r = MSK_putbarcblocktriplet(task, 6, Cj, Ck, Cl, Cv);

      /* Set the equality constraint (6 nonzeros).*/
      if (r == MSK_RES_OK)
        r = MSK_putbarablocktriplet(task, 6, Ai, Aj, Ak, Al, Av);
```

(continues on next page)

(continued from previous page)

```
    /* Set the inequality constraint (1 nonzero).*/
    if (r == MSK_RES_OK)
      r = MSK_putbarablocktriplet(task, 1, &A2i, &A2j, &A2k, &A2l, &A2v);

    /* Set constraint bounds */
    if (r == MSK_RES_OK)
      r = MSK_putconboundslice(task, 0, 2, bkc, blc, buc);

    if (r == MSK_RES_OK)
    {
      MSKrescodee trmcode;

      /* Run optimizer */
      r = MSK_optimizetrm(task, &trmcode);
      MSK_solutionsummary(task, MSK_STREAM_MSG);

      if (r == MSK_RES_OK)
      {
        MSKsolstae solsta;

        MSK_getsolsta(task, MSK_SOL_ITR, &solsta);

        switch (solsta)
        {
          case MSK_SOL_STA_OPTIMAL:

            /* Retrieve the soution for all symmetric variables */
            printf("Solution (lower triangular part vectorized):\n");
            for(i = 0; i < numbarvar; i++) {
              dim = dimbarvar[i] * (dimbarvar[i] + 1) / 2;
              barx = (MSKrealt *) MSK_calloctask(task, dim, sizeof(MSKrealt));

              MSK_getbarxj(task, MSK_SOL_ITR, i, barx);

              printf("X%d: ", i + 1);
              for (j = 0; j < dim; ++j)
                printf("%.3f ", barx[j]);
              printf("\n");

              MSK_freetask(task, barx);
            }

            break;

          case MSK_SOL_STA_DUAL_INFEAS_CER:
          case MSK_SOL_STA_PRIM_INFEAS_CER:
            printf("Primal or dual infeasibility certificate found.\n");
            break;

          case MSK_SOL_STA_UNKNOWN:
            printf("The status of the solution could not be determined. Termination␣
↪code: %d.\n", trmcode);
            break;

          default:
```

(continues on next page)

(continued from previous page)

```
            printf("Other solution status.");
            break;
          }
        }
        else
        {
          printf("Error while optimizing.\n");
        }
      }

      if (r != MSK_RES_OK)
      {
        /* In case of an error print error code and description. */
        char symname[MSK_MAX_STR_LEN];
        char desc[MSK_MAX_STR_LEN];

        printf("An error occurred while optimizing.\n");
        MSK_getcodedesc(r,
                        symname,
                        desc);
        printf("Error %s - '%s'\n", symname, desc);
      }
    }
    /* Delete the task and the associated data. */
    MSK_deletetask(&task);
  }

  /* Delete the environment and the associated data. */
  MSK_deleteenv(&env);

  return (r);
} /* main */
```

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)

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

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).

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

Vectorization of the LMI is performed as explained in Sec. 15.10.

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 *MSK_putafebarfblocktriplet*.

```
                r = MSK_putafebarfblocktriplet(task,
                                               2,
                                               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.

```
            MSKint64t acc1_afeidx[] = {0};
            if (r == MSK_RES_OK)
                r = MSK_appendrplusdomain(task, 1, NULL);
            if (r == MSK_RES_OK)
                r = MSK_appendacc(task, 0, 1, acc1_afeidx, NULL);

            /* Append the SVEC_PSD domain and the corresponding ACC */
            MSKint64t acc2_afeidx[] = {1, 2, 3};
            if (r == MSK_RES_OK)
                r = MSK_appendsvecpsdconedomain(task, 3, NULL);
            if (r == MSK_RES_OK)
                r = MSK_appendacc(task, 1, 3, acc2_afeidx, NULL);
```

Source code

Listing 6.12: Source code solving problem (6.25).

```
#include <stdio.h>
#include <math.h>

#include "mosek.h"    /* Include the MOSEK definition file.  */

#define NUMVAR    2   /* Number of scalar variables */
#define NUMAFE    4   /* Number of affine expressions        */
#define NUMFNZ    6   /* Number of non-zeros in F            */
#define NUMBARVAR 1   /* Number of semidefinite variables    */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
    printf("%s", str);
} /* printstr */
```

(continues on next page)

(continued from previous page)

```
int main(int argc, const char *argv[])
{
    MSKrescodee  r;

    const MSKint32t    DIMBARVAR[] = {2};          /* Dimension of semidefinite cone */
↪
          MSKint64t    LENBARVAR[] = {2 * (2 + 1) / 2}; /* Number of scalar SD␣
↪variables  */

    const MSKint32t   barc_j[] = {0, 0},
                      barc_k[] = {0, 1},
                      barc_l[] = {0, 1};
    const MSKrealt    barc_v[] = {1, 1};

    const MSKint64t   barf_i[] = {0,0};
    const MSKint32t   barf_j[] = {0,0},
                      barf_k[] = {0,1},
                      barf_l[] = {0,0};
    const MSKrealt    barf_v[] = {0,1};

    const MSKint64t   afeidx[] = {0, 0, 1, 2, 2, 3};
    const MSKint32t   varidx[] = {0, 1, 1, 0, 1, 0};
    const MSKrealt     f_val[] = {-1, -1, 3, sqrt(2), sqrt(2), 3},
                           g[] = {0, -1, 0, -1};

    MSKrealt *xx, *barx;
    MSKint32t i, j;

    MSKenv_t     env = NULL;
    MSKtask_t    task = NULL;

    /* Create the mosek environment. */
    r = MSK_makeenv(&env, NULL);

    if (r == MSK_RES_OK)
    {
        /* Create the optimization task. */
        r = MSK_maketask(env, 0, 0, &task);

        if (r == MSK_RES_OK)
        {
            r = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

            /* Append 'NUMAFE' empty affine expressions. */
            if (r == MSK_RES_OK)
                r = MSK_appendafes(task, NUMAFE);

            /* Append 'NUMVAR' scalar variables.
            The variables will initially be fixed at zero (x=0). */
            if (r == MSK_RES_OK)
                r = MSK_appendvars(task, NUMVAR);

            /* Append 'NUMBARVAR' semidefinite variables. */
            if (r == MSK_RES_OK)
            {
                r = MSK_appendbarvars(task, NUMBARVAR, DIMBARVAR);
```

(continues on next page)

(continued from previous page)

```
            }

            /* Set the constant term in the objective. */
            if (r == MSK_RES_OK)
                r = MSK_putcfix(task, 1.0);

            /* Set c_j and the bounds for each scalar variable*/
            for (j = 0; j < NUMVAR && r == MSK_RES_OK; ++j)
            {
                r = MSK_putcj(task, j, 1.0);
                if (r==MSK_RES_OK)
                    r = MSK_putvarbound(task, j, MSK_BK_FR, -MSK_INFINITY, MSK_
↪INFINITY);
            }

            /* Set the linear term barc_j in the objective.*/
            if (r == MSK_RES_OK)
                r = MSK_putbarcblocktriplet(task,
                                            2,
                                            barc_j,
                                            barc_k,
                                            barc_l,
                                            barc_v);

            /* Set the F matrix */
            if (r == MSK_RES_OK)
                r = MSK_putafefentrylist(task, NUMFNZ, afeidx, varidx, f_val);
            /* Set the g vector */
            if (r == MSK_RES_OK)
                r = MSK_putafegslice(task, 0, NUMAFE, g);

            /* Set the barF matrix */
            if (r == MSK_RES_OK)
                r = MSK_putafebarfblocktriplet(task,
                                               2,
                                               barf_i,
                                               barf_j,
                                               barf_k,
                                               barf_l,
                                               barf_v);

            /* Append R+ domain and the corresponding ACC */
            MSKint64t acc1_afeidx[] = {0};
            if (r == MSK_RES_OK)
                r = MSK_appendrplusdomain(task, 1, NULL);
            if (r == MSK_RES_OK)
                r = MSK_appendacc(task, 0, 1, acc1_afeidx, NULL);

            /* Append the SVEC_PSD domain and the corresponding ACC */
            MSKint64t acc2_afeidx[] = {1, 2, 3};
            if (r == MSK_RES_OK)
                r = MSK_appendsvecpsdconedomain(task, 3, NULL);
            if (r == MSK_RES_OK)
                r = MSK_appendacc(task, 1, 3, acc2_afeidx, NULL);

            if (r == MSK_RES_OK)
```

(continues on next page)

(continued from previous page)

```
            {
                MSKrescodee trmcode;

                /* Run optimizer */
                r = MSK_optimizetrm(task, &trmcode);

                /* Print a summary containing information
                about the solution for debugging purposes*/
                MSK_solutionsummary(task, MSK_STREAM_MSG);

                if (r == MSK_RES_OK)
                {
                    MSKsolstae solsta;
                    MSK_getsolsta(task, MSK_SOL_ITR, &solsta);

                    switch (solsta)
                    {
                        case MSK_SOL_STA_OPTIMAL:
                        xx   = (MSKrealt *) MSK_calloctask(task, NUMVAR,␣
↪sizeof(MSKrealt));
                        barx = (MSKrealt *) MSK_calloctask(task, LENBARVAR[0],␣
↪sizeof(MSKrealt));

                        MSK_getxx(task,MSK_SOL_ITR,xx);
                        MSK_getbarxj(task, MSK_SOL_ITR, 0, barx);

                        printf("Optimal primal solution\n");
                        for (i = 0; i < NUMVAR; ++i)
                            printf("x[%d]   : % e\n", i, xx[i]);

                        for (i = 0; i < LENBARVAR[0]; ++i)
                            printf("barx[%d]: % e\n", i, barx[i]);

                        MSK_freetask(task, xx);
                        MSK_freetask(task, barx);
                        break;

                        case MSK_SOL_STA_DUAL_INFEAS_CER:
                        case MSK_SOL_STA_PRIM_INFEAS_CER:
                            printf("Primal or dual infeasibility certificate found.\n
↪");
                            break;
                        case MSK_SOL_STA_UNKNOWN:
                            printf("The status of the solution could not be␣
↪determined. Termination code: %d.\n", trmcode);
                            break;
                        default:
                            printf("Other solution status.");
                            break;
                    }
                }
                else
                    printf("Error while optimizing.\n");
            }
            if (r != MSK_RES_OK)
            {
```

(continues on next page)

(continued from previous page)

```
                /* In case of an error print error code and description. */
                char symname[MSK_MAX_STR_LEN];
                char desc[MSK_MAX_STR_LEN];
                printf("An error occurred while optimizing.\n");
                MSK_getcodedesc(r, symname, desc);
                printf("Error %s - '%s'\n", symname, desc);
            }
        }
        /* Delete the task and the associated data. */
        MSK_deletetask(&task);
    }

    /* Delete the environment and the associated data. */
    MSK_deleteenv(&env);

    return (r);
} /* main */
```

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

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

to demonstrate how to set up and solve a problem with integer variables. It has the structure of a linear optimization problem (see Sec. 6.1) except for integrality constraints on the variables. Therefore, only the specification of the integer constraints requires something new compared to the linear optimization problem discussed previously.

First, the integrality constraints are imposed using the function *MSK_putvartype*:

```
    for (j = 0; j < numvar && r == MSK_RES_OK; ++j)
      r = MSK_putvartype(task, j, 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 */
      r = MSK_putdouparam(task,
                          MSK_DPAR_MIO_MAX_TIME,
                          60.0);
```

The complete source for the example is listed Listing 6.13. Please note that when *MSK_getsolutionslice* is called, 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).

```
#include <stdio.h>
#include "mosek.h" /* Include the MOSEK definition file. */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, char *argv[])
{
  const MSKint32t numvar = 2,
                  numcon = 2;

  double       c[]   = {  1.0, 0.64 };
  MSKboundkeye bkc[] = { MSK_BK_UP,    MSK_BK_LO };
  double       blc[] = { -MSK_INFINITY, -4.0 };
  double       buc[] = { 250.0,        MSK_INFINITY };

  MSKboundkeye bkx[] = { MSK_BK_LO,    MSK_BK_LO };
  double       blx[] = { 0.0,          0.0 };
  double       bux[] = { MSK_INFINITY, MSK_INFINITY };


  MSKint32t    aptrb[] = { 0, 2 },
               aptre[] = { 2, 4 },
               asub[] = { 0,    1,    0,    1 };
  double       aval[] = { 50.0, 3.0, 31.0, -2.0 };
  MSKint32t    i, j;

  MSKenv_t     env = NULL;
  MSKtask_t    task = NULL;
  MSKrescodee  r;

  /* Create the mosek environment. */
  r = MSK_makeenv(&env, NULL);

  /* Check if return code is ok. */
  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, 0, 0, &task);

    if (r == MSK_RES_OK)
      r = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

    /* Append 'numcon' empty constraints.
     The constraints will initially have no bounds. */
    if (r == MSK_RES_OK)
      r = MSK_appendcons(task, numcon);

    /* Append 'numvar' variables.
     The variables will initially be fixed at zero (x=0). */
    if (r == MSK_RES_OK)
      r = MSK_appendvars(task, numvar);
```

(continues on next page)

(continued from previous page)

```
/* Optionally add a constant term to the objective. */
if (r == MSK_RES_OK)
  r = MSK_putcfix(task, 0.0);
for (j = 0; j < numvar && r == MSK_RES_OK; ++j)
{
  /* Set the linear term c_j in the objective.*/
  if (r == MSK_RES_OK)
    r = MSK_putcj(task, j, c[j]);

  /* Set the bounds on variable j.
   blx[j] <= x_j <= bux[j] */
  if (r == MSK_RES_OK)
    r = MSK_putvarbound(task,
                        j,           /* Index of variable.*/
                        bkx[j],      /* Bound key.*/
                        blx[j],      /* Numerical value of lower bound.*/
                        bux[j]);     /* Numerical value of upper bound.*/

  /* Input column j of A */
  if (r == MSK_RES_OK)
    r = MSK_putacol(task,
                    j,                  /* Variable (column) index.*/
                    aptre[j] - aptrb[j], /* Number of non-zeros in column j.*/
                    asub + aptrb[j],   /* Pointer to row indexes of column j.*/
                    aval + aptrb[j]);  /* Pointer to Values of column j.*/

}

/* Set the bounds on constraints.
   for i=1, ...,numcon : blc[i] <= constraint i <= buc[i] */
for (i = 0; i < numcon && r == MSK_RES_OK; ++i)
  r = MSK_putconbound(task,
                      i,           /* Index of constraint.*/
                      bkc[i],      /* Bound key.*/
                      blc[i],      /* Numerical value of lower bound.*/
                      buc[i]);     /* Numerical value of upper bound.*/

/* Specify integer variables. */
for (j = 0; j < numvar && r == MSK_RES_OK; ++j)
  r = MSK_putvartype(task, j, MSK_VAR_TYPE_INT);

if (r == MSK_RES_OK)
  r =  MSK_putobjsense(task,
                       MSK_OBJECTIVE_SENSE_MAXIMIZE);

if (r == MSK_RES_OK)
  /* Set max solution time */
  r = MSK_putdouparam(task,
                      MSK_DPAR_MIO_MAX_TIME,
                      60.0);

if (r == MSK_RES_OK)
{
  MSKrescodee trmcode;
```

(continues on next page)

(continued from previous page)

```
    /* Run optimizer */
    r = MSK_optimizetrm(task, &trmcode);

    /* Print a summary containing information
       about the solution for debugging purposes*/
    MSK_solutionsummary(task, MSK_STREAM_MSG);

    if (r == MSK_RES_OK)
    {
      MSKint32t  j;
      MSKsolstae solsta;
      double     *xx = NULL;

      MSK_getsolsta(task, MSK_SOL_ITG, &solsta);

      xx = calloc(numvar, sizeof(double));
      if (xx)
      {
        switch (solsta)
        {
          case MSK_SOL_STA_INTEGER_OPTIMAL:
            MSK_getxx(task,
                      MSK_SOL_ITG,    /* Request the integer solution. */
                      xx);

            printf("Optimal solution.\n");
            for (j = 0; j < numvar; ++j)
              printf("x[%d]: %e\n", j, xx[j]);
            break;
          case MSK_SOL_STA_PRIM_FEAS:
            /* A feasible but not necessarily optimal solution was located. */
            MSK_getxx(task, MSK_SOL_ITG, xx);

            printf("Feasible solution.\n");
            for (j = 0; j < numvar; ++j)
              printf("x[%d]: %e\n", j, xx[j]);
            break;
          case MSK_SOL_STA_UNKNOWN:
            {
              MSKprostae prosta;
              MSK_getprosta(task, MSK_SOL_ITG, &prosta);
              switch (prosta)
              {
                case MSK_PRO_STA_PRIM_INFEAS_OR_UNBOUNDED:
                  printf("Problem status Infeasible or unbounded\n");
                  break;
                case MSK_PRO_STA_PRIM_INFEAS:
                  printf("Problem status Infeasible.\n");
                  break;
                case MSK_PRO_STA_UNKNOWN:
                  printf("Problem status unknown. Termination code %d.\n", trmcode);
                  break;
                default:
                  printf("Other problem status.");
                  break;
              }
```

(continues on next page)

(continued from previous page)

```
              }
              break;
            default:
              printf("Other solution status.");
              break;
          }
        }
        else
        {
          r = MSK_RES_ERR_SPACE;
        }
        free(xx);
      }
    }

    if (r != MSK_RES_OK)
    {
      /* In case of an error print error code and description. */
      char symname[MSK_MAX_STR_LEN];
      char desc[MSK_MAX_STR_LEN];

      printf("An error occurred while optimizing.\n");
      MSK_getcodedesc(r,
                      symname,
                      desc);
      printf("Error %s - '%s'\n", symname, desc);
    }

    MSK_deletetask(&task);
  }
  MSK_deleteenv(&env);

  printf("Return code: %d.\n", r);
  return (r);
} /* main */
```

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 *MSK_putsolution* .

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

```
  if (r == MSK_RES_OK)
  {
    /* Assign values to integer variables
       (we only set a slice of xx) */
    double        xxInit[] = {1.0, 1.0, 0.0};
    r = MSK_putxxslice(task, MSK_SOL_ITG, 0, 3, xxInit);
  }
  if (r == MSK_RES_OK)
  {
    /* Request constructing the solution from integer variable values */
    r = MSK_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

```
    MSK_getintinf(task, MSK_IINF_MIO_CONSTRUCT_SOLUTION, &constr);
    MSK_getdouinf(task, MSK_DINF_MIO_CONSTRUCT_SOLUTION_OBJ, &constr_obj);
    printf("Construct solution utilization: %d\nConstruct solution objective: %.3f\n",
↪ constr, constr_obj);
```

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 C is

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

Listing 6.16: Implementation of problem (6.30).

```
int main(int argc, char *argv[])
{
  MSKint32t         numvar = 3;  /* x, y, t */

  MSKvariabletypee  vart[] = { MSK_VAR_TYPE_INT, MSK_VAR_TYPE_INT };
  MSKint32t       intsub[] = { 0, 1 };

  MSKint32t    i, j;

  MSKenv_t     env = NULL;
  MSKtask_t    task = NULL;
  MSKrescodee  r, trm;

  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    r = MSK_maketask(env, 0, 0, &task);

    if (r == MSK_RES_OK)
      r = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

    if (r == MSK_RES_OK)
      r = MSK_appendvars(task, numvar);
    if (r == MSK_RES_OK)
      r = MSK_putvarboundsliceconst(task, 0, numvar, MSK_BK_FR, -0.0, 0.0);

    /* Integrality constraints */
    if (r == MSK_RES_OK)
      r = MSK_putvartypelist(task, 2, intsub, vart);

    /* Objective */
    if (r == MSK_RES_OK)
      r = MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MINIMIZE);
    if (r == MSK_RES_OK)
      r = MSK_putcj(task, 2, 1.0);    /* Minimize t */

    /* Conic part of the problem */
    if (r == MSK_RES_OK)
    {
      /* Set up the affine expressions */
      /* x, x-3.8, y, t, 1.0 */
      MSKint64t afeidx[] = {0, 1, 2, 3};
      MSKint32t varidx[] = {0, 0, 1, 2};
      MSKrealt     val[] = {1.0, 1.0, 1.0, 1.0};
      MSKrealt       g[] = {0.0, -3.8, 0.0, 0.0, 1.0};
      MSKint64t domExp, domQuad;
      MSKint64t afeidxExp[]  = {1, 4, 2};
      MSKint64t afeidxQuad[] = {3, 0, 2};

      if (r == MSK_RES_OK)
        r = MSK_appendafes(task, 5);
      if (r == MSK_RES_OK)
        r = MSK_putafefentrylist(task,
                                 4,
```

(continues on next page)

(continued from previous page)

```
                                  afeidx,
                                  varidx,
                                  val);
      if (r == MSK_RES_OK)
        r = MSK_putafegslice(task, 0, 5, g);

      // Add constraint (x-3.8, 1, y) \in \EXP
      if (r == MSK_RES_OK)
        r = MSK_appendprimalexpconedomain(task, &domExp);
      if (r == MSK_RES_OK)
        r = MSK_appendacc(task, domExp, 3, afeidxExp, NULL);

      // Add constraint (t, x, y) \in \QUAD
      if (r == MSK_RES_OK)
        r = MSK_appendquadraticconedomain(task, 3, &domQuad);
      if (r == MSK_RES_OK)
        r = MSK_appendacc(task, domQuad, 3, afeidxQuad, NULL);
    }

    /* Optimize the problem */
    if (r == MSK_RES_OK)
      r = MSK_optimizetrm(task, &trm);

    if (r == MSK_RES_OK)
      r = MSK_solutionsummary(task, MSK_STREAM_MSG);

    if (r == MSK_RES_OK)
    {
      MSKrealt xx[] = {0, 0};

      r = MSK_getxxslice(task, MSK_SOL_ITG, 0, 2, xx);

      if (r == MSK_RES_OK)
        printf("x = %.2f, y = %.2f\n", xx[0], xx[1]);
    }

    if (task) MSK_deletetask(&task);
  }

  if (env) MSK_deleteenv(&env);
  return r;
}
```

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. 15.10.1. 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** 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 is satisfied. A term 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. 13.4.6 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:

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

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.

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*:

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

where 0^n denotes the n-dimensional zero domain and $\mathbb{R}^n_{\leq 0}$ denotes the n-dimensional nonpositive orthant, as in Sec. 15.10.

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):

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

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;
    r = MSK_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).

```
    const MSKint64t fafeidx[] = {0, 0, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    const MSKint32t fvaridx[] = {0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3};
    const MSKrealt  fval[]    = {1.0, -2.0, 1.0, -3.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
↪ 1.0, 1.0};
    const MSKrealt  g[]       = {1.0, 2.0, 0.0, 0.0, 0.0, 0.0, -2.5, -2.5, -2.5, -2.
↪5};

    r = MSK_putafefentrylist(task, 12, fafeidx, fvaridx, fval);
    if (r == MSK_RES_OK)
      r = MSK_putafegslice(task, 0, numafe, 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:

```
    MSK_appendrzerodomain(task, 1, &zero1);
    MSK_appendrzerodomain(task, 2, &zero2);
    MSK_appendrminusdomain(task, 1, &rminus1);
```

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 *MSK_appenddjcs* will append a number of initially empty DJCs to the task:

```
    numdjc = 2;
    r = MSK_appenddjcs(task, numdjc);
```

We can then define each disjunction with the method *MSK_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:

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

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:

```
    const MSKint64t domidxlist[] = {rminus1, zero2, rminus1, zero2};
    const MSKint64t afeidxlist[] = {0, 4, 5, 1, 2, 3};
    const MSKint64t termsizelist[] = {2, 2};

    r = MSK_putdjc(task,
                   0,                       // DJC index
                   4, domidxlist,
                   6, afeidxlist,
                   NULL,                    // Unused
                   2, 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:

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

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

```
      const MSKint64t domidxlist[] = {zero1, zero1, zero1, zero1};
      const MSKint64t afeidxlist[] = {6, 7, 8, 9};
      const MSKint64t termsizelist[] = {1, 1, 1, 1};

      r = MSK_putdjc(task,
                     1,                        // DJC index
                     4, domidxlist,
                     4, afeidxlist,
                     NULL,                     // Unused
                     4, termsizelist);
```

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 *MSK_optimize* and retrieve the solution with *MSK_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.

```
#include <stdio.h>
#include "mosek.h"

/* This function prints log output from MOSEK to the terminal. */
static void MSKAPI printstr(void       *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{
  MSKenv_t           env  = NULL;
  MSKtask_t          task = NULL;
  MSKrescodee        r = MSK_RES_OK;
  MSKint32t          i, j, numvar;
  MSKint64t          k, l, numafe, numdjc;
  MSKint64t          zero1, zero2, rminus1;  // Domain indices

  /* Create the mosek environment. */
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, 0, 0, &task);

    if (r == MSK_RES_OK)
    {
      // Append free variables
      numvar = 4;
      r = MSK_appendvars(task, numvar);
      if (r == MSK_RES_OK)
        MSK_putvarboundsliceconst(task, 0, numvar, MSK_BK_FR, -MSK_INFINITY, MSK_
↪INFINITY);
```

(continues on next page)

(continued from previous page)

```
  }

  if (r == MSK_RES_OK)
  {
    // The linear part: the linear constraint
    const MSKint32t idx[] = {0, 1, 2, 3};
    const MSKrealt  val[] = {1, 1, 1, 1};

    r = MSK_appendcons(task, 1);
    if (r == MSK_RES_OK) MSK_putarow(task, 0, 4, idx, val);
    if (r == MSK_RES_OK) MSK_putconbound(task, 0, MSK_BK_LO, -10.0, -10.0);
  }

  if (r == MSK_RES_OK)
  {
    // The linear part: objective
    const MSKint32t idx[] = {0, 1, 2, 3};
    const MSKrealt  val[] = {2, 1, 3, 1};

    r = MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MINIMIZE);
    if (r == MSK_RES_OK) MSK_putclist(task, 4, idx, val);
  }

  // Fill in the affine expression storage F, g
  if (r == MSK_RES_OK)
  {
    numafe = 10;
    r = MSK_appendafes(task, numafe);
  }

  if (r == MSK_RES_OK)
  {
    const MSKint64t fafeidx[] = {0, 0, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    const MSKint32t fvaridx[] = {0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3};
    const MSKrealt  fval[]    = {1.0, -2.0, 1.0, -3.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
↪ 1.0, 1.0};
    const MSKrealt  g[]       = {1.0, 2.0, 0.0, 0.0, 0.0, 0.0, -2.5, -2.5, -2.5, -2.
↪5};

    r = MSK_putafefentrylist(task, 12, fafeidx, fvaridx, fval);
    if (r == MSK_RES_OK)
      r = MSK_putafegslice(task, 0, numafe, g);
  }

  if (r == MSK_RES_OK)
  {
    // Create domains
    MSK_appendrzerodomain(task, 1, &zero1);
    MSK_appendrzerodomain(task, 2, &zero2);
    MSK_appendrminusdomain(task, 1, &rminus1);
  }

  if (r == MSK_RES_OK)
  {
    // Append disjunctive constraints
    numdjc = 2;
```

(continues on next page)

(continued from previous page)

```
    r = MSK_appenddjcs(task, numdjc);
  }

  if (r == MSK_RES_OK)
  {
    // First disjunctive constraint
    const MSKint64t domidxlist[] = {rminus1, zero2, rminus1, zero2};
    const MSKint64t afeidxlist[] = {0, 4, 5, 1, 2, 3};
    const MSKint64t termsizelist[] = {2, 2};

    r = MSK_putdjc(task,
                   0,                      // DJC index
                   4, domidxlist,
                   6, afeidxlist,
                   NULL,                   // Unused
                   2, termsizelist);
  }

  if (r == MSK_RES_OK)
  {
    // Second disjunctive constraint
    const MSKint64t domidxlist[] = {zero1, zero1, zero1, zero1};
    const MSKint64t afeidxlist[] = {6, 7, 8, 9};
    const MSKint64t termsizelist[] = {1, 1, 1, 1};

    r = MSK_putdjc(task,
                   1,                      // DJC index
                   4, domidxlist,
                   4, afeidxlist,
                   NULL,                   // Unused
                   4, termsizelist);
  }

  // Useful for debugging
  if (r == MSK_RES_OK)
  {
    // Write a human-readable file
    r = MSK_writedata(task, "djc.ptf");
    // Directs the log task stream to the 'printstr' function.
    if (r == MSK_RES_OK)
      r = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);
  }

  // Solve the problem
  if (r == MSK_RES_OK)
  {
    MSKrescodee trmcode;

    r = MSK_optimizetrm(task, &trmcode);

    /* Print a summary containing information
       about the solution for debugging purposes. */
    MSK_solutionsummary(task, MSK_STREAM_LOG);

    if (r == MSK_RES_OK)
    {
```

(continues on next page)

(continued from previous page)

```
        MSKsolstae solsta;

        if (r == MSK_RES_OK)
          r = MSK_getsolsta(task,
                            MSK_SOL_ITG,
                            &solsta);
        switch (solsta)
        {
          case MSK_SOL_STA_INTEGER_OPTIMAL:
            {
              double *xx = (double*) calloc(numvar, sizeof(double));
              if (xx)
              {
                MSK_getxx(task,
                          MSK_SOL_ITG,
                          xx);

                printf("Optimal primal solution\n");
                for (j = 0; j < numvar; ++j)
                  printf("x[%d]: %e\n", j, xx[j]);

                free(xx);
              }
              else
                r = MSK_RES_ERR_SPACE;

              break;
            }
          default:
            printf("Another solution status.\n");
            break;
        }
      }
    }

    if (r != MSK_RES_OK)
    {
      /* In case of an error print error code and description. */
      char symname[MSK_MAX_STR_LEN];
      char desc[MSK_MAX_STR_LEN];

      printf("An error occurred while optimizing.\n");
      MSK_getcodedesc(r,
                      symname,
                      desc);
      printf("Error %s - '%s'\n", symname, desc);
    }

    /* Delete the task and the associated data. */
    MSK_deletetask(&task);
  }

  /* Delete the environment and the associated data. */
  MSK_deleteenv(&env);

  return r;
```

(continues on next page)

(continued from previous page)

```
}
```

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:

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

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

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

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

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

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

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

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:

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

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 *MSK_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):

```
        qsubi[0] = 0;   qsubj[0] = 0;  qval[0] = 2.0;
        qsubi[1] = 1;   qsubj[1] = 1;  qval[1] = 0.2;
        qsubi[2] = 2;   qsubj[2] = 0;  qval[2] = -1.0;
        qsubi[3] = 2;   qsubj[3] = 2;  qval[3] = 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:

```
      r = MSK_putqobj(task, NUMQNZ, qsubi, qsubj, qval);
```

Source code

Listing 6.18: Source code implementing problem (6.42).

```
#include <stdio.h>

#include "mosek.h" /* Include the MOSEK definition file. */

#define NUMCON 1   /* Number of constraints.             */
#define NUMVAR 3   /* Number of variables.               */
#define NUMANZ 3   /* Number of non-zeros in A.           */
#define NUMQNZ 4   /* Number of non-zeros in Q.           */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{

  double        c[]   = {0.0, -1.0, 0.0};

  MSKboundkeye  bkc[] = {MSK_BK_LO};
  double        blc[] = {1.0};
  double        buc[] = { +MSK_INFINITY };

  MSKboundkeye  bkx[] = {MSK_BK_LO,
                         MSK_BK_LO,
                         MSK_BK_LO
                        };
  double        blx[] = {0.0,
                         0.0,
                         0.0
                        };
  double        bux[] = { +MSK_INFINITY,
                          +MSK_INFINITY,
                          +MSK_INFINITY
                        };

  MSKint32t     aptrb[] = {0,   1,   2},
                aptre[] = {1,   2,   3},
                asub[]  = {0,   0,   0};
  double        aval[]  = {1.0, 1.0, 1.0};

  MSKint32t     qsubi[NUMQNZ];
  MSKint32t     qsubj[NUMQNZ];
  double        qval[NUMQNZ];

  MSKint32t     i, j;
  double        xx[NUMVAR];

  MSKenv_t      env = NULL;
```

(continues on next page)

(continued from previous page)

```
  MSKtask_t     task = NULL;
  MSKrescodee   r;

  /* Create the mosek environment. */
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, NUMCON, NUMVAR, &task);

    if (r == MSK_RES_OK)
    {
      r = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

      /* Append 'NUMCON' empty constraints.
       The constraints will initially have no bounds. */
      if (r == MSK_RES_OK)
        r = MSK_appendcons(task, NUMCON);

      /* Append 'NUMVAR' variables.
       The variables will initially be fixed at zero (x=0). */
      if (r == MSK_RES_OK)
        r = MSK_appendvars(task, NUMVAR);

      /* Optionally add a constant term to the objective. */
      if (r == MSK_RES_OK)
        r = MSK_putcfix(task, 0.0);
      for (j = 0; j < NUMVAR && r == MSK_RES_OK; ++j)
      {
        /* Set the linear term c_j in the objective.*/
        if (r == MSK_RES_OK)
          r = MSK_putcj(task, j, c[j]);

        /* Set the bounds on variable j.
         blx[j] <= x_j <= bux[j] */
        if (r == MSK_RES_OK)
          r = MSK_putvarbound(task,
                              j,           /* Index of variable.*/
                              bkx[j],      /* Bound key.*/
                              blx[j],      /* Numerical value of lower bound.*/
                              bux[j]);     /* Numerical value of upper bound.*/

        /* Input column j of A */
        if (r == MSK_RES_OK)
          r = MSK_putacol(task,
                          j,                 /* Variable (column) index.*/
                          aptre[j] - aptrb[j], /* Number of non-zeros in column j.*/
                          asub + aptrb[j],   /* Pointer to row indexes of column j.*/
                          aval + aptrb[j]);  /* Pointer to Values of column j.*/

      }

      /* Set the bounds on constraints.
         for i=1, ...,NUMCON : blc[i] <= constraint i <= buc[i] */
      for (i = 0; i < NUMCON && r == MSK_RES_OK; ++i)
```

(continues on next page)

(continued from previous page)

```
    r = MSK_putconbound(task,
                        i,            /* Index of constraint.*/
                        bkc[i],       /* Bound key.*/
                        blc[i],       /* Numerical value of lower bound.*/
                        buc[i]);      /* Numerical value of upper bound.*/

  if (r == MSK_RES_OK)
  {
    /*
     * The lower triangular part of the Q
     * matrix in the objective is specified.
     */

    qsubi[0] = 0;   qsubj[0] = 0;  qval[0] = 2.0;
    qsubi[1] = 1;   qsubj[1] = 1;  qval[1] = 0.2;
    qsubi[2] = 2;   qsubj[2] = 0;  qval[2] = -1.0;
    qsubi[3] = 2;   qsubj[3] = 2;  qval[3] = 2.0;

    /* Input the Q for the objective. */

    r = MSK_putqobj(task, NUMQNZ, qsubi, qsubj, qval);
  }

  if (r == MSK_RES_OK)
  {
    MSKrescodee trmcode;

    /* Run optimizer */
    r = MSK_optimizetrm(task, &trmcode);

    /* Print a summary containing information
       about the solution for debugging purposes*/
    MSK_solutionsummary(task, MSK_STREAM_MSG);

    if (r == MSK_RES_OK)
    {
      MSKsolstae solsta;
      int j;

      MSK_getsolsta(task, MSK_SOL_ITR, &solsta);

      switch (solsta)
      {
        case MSK_SOL_STA_OPTIMAL:
          MSK_getxx(task,
                    MSK_SOL_ITR,    /* Request the interior solution. */
                    xx);

          printf("Optimal primal solution\n");
          for (j = 0; j < NUMVAR; ++j)
            printf("x[%d]: %e\n", j, xx[j]);

          break;

        case MSK_SOL_STA_DUAL_INFEAS_CER:
        case MSK_SOL_STA_PRIM_INFEAS_CER:
```

(continues on next page)

(continued from previous page)

```
              printf("Primal or dual infeasibility certificate found.\n");
              break;

            case MSK_SOL_STA_UNKNOWN:
              printf("The status of the solution could not be determined. Termination␣
↪code: %d.\n", trmcode);
              break;

            default:
              printf("Other solution status.");
              break;
          }
        }
        else
        {
          printf("Error while optimizing.\n");
        }
      }

      if (r != MSK_RES_OK)
      {
        /* In case of an error print error code and description. */
        char symname[MSK_MAX_STR_LEN];
        char desc[MSK_MAX_STR_LEN];

        printf("An error occurred while optimizing.\n");
        MSK_getcodedesc(r,
                        symname,
                        desc);
        printf("Error %s - '%s'\n", symname, desc);
      }
    }
    MSK_deletetask(&task);
  }
  MSK_deleteenv(&env);

  return (r);
} /* main */
```

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:

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

This is equivalent to

$$\begin{array}{lccc}\text{minimize} & \frac{1}{2}x^TQ^ox + c^Tx & & \\ \text{subject to} & \frac{1}{2}x^TQ^0x + Ax & \geq & b, \\ & x \geq 0, & & \end{array} \tag{6.43}$$

where

$$Q^o = \left[\begin{array}{ccc} 2 & 0 & -1 \\ 0 & 0.2 & 0 \\ -1 & 0 & 2 \end{array}\right], c = \left[\begin{array}{ccc} 0 & -1 & 0 \end{array}\right]^T, A = \left[\begin{array}{ccc} 1 & 1 & 1 \end{array}\right], b = 1.$$

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

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 *MSK_putqconk*.

```
        qsubi[0] = 0;   qsubj[0] = 0;  qval[0] = -2.0;
        qsubi[1] = 1;   qsubj[1] = 1;  qval[1] = -2.0;
        qsubi[2] = 2;   qsubj[2] = 2;  qval[2] = -0.2;
        qsubi[3] = 2;   qsubj[3] = 0;  qval[3] = 0.2;

        /* Put Q^0 in constraint with index 0. */

        r = MSK_putqconk(task,
                         0,
                         4,
                         qsubi,
                         qsubj,
                         qval);
```

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

Source code

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

```
#include <stdio.h>

#include "mosek.h" /* Include the MOSEK definition file. */

#define NUMCON 1   /* Number of constraints.             */
#define NUMVAR 3   /* Number of variables.               */
#define NUMANZ 3   /* Number of non-zeros in A.          */
#define NUMQNZ 4   /* Number of non-zeros in Q.          */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
{

  MSKrescodee  r;

  double       c[]    = {0.0, -1.0, 0.0};

  MSKboundkeye bkc[]  = {MSK_BK_LO};
  double       blc[]  = {1.0};
  double       buc[]  = { +MSK_INFINITY};

  MSKboundkeye bkx[]  = {MSK_BK_LO,
```

(continues on next page)

(continued from previous page)

```
                        MSK_BK_LO,
                        MSK_BK_LO
                       };
double       blx[]  = {0.0,
                        0.0,
                        0.0
                       };
double       bux[]  = { +MSK_INFINITY,
                         +MSK_INFINITY,
                         +MSK_INFINITY
                       };

MSKint32t    aptrb[] = {0, 1, 2 },
             aptre[] = {1, 2, 3},
             asub[] = { 0,   0,   0};
double       aval[] = { 1.0, 1.0, 1.0};
MSKint32t    qsubi[NUMQNZ],
             qsubj[NUMQNZ];
double       qval[NUMQNZ];

MSKint32t    j, i;
double       xx[NUMVAR];
MSKenv_t     env;
MSKtask_t    task;

/* Create the mosek environment. */
r = MSK_makeenv(&env, NULL);

if (r == MSK_RES_OK)
{
  /* Create the optimization task. */
  r = MSK_maketask(env, NUMCON, NUMVAR, &task);

  if (r == MSK_RES_OK)
  {
    r = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

    /* Append 'NUMCON' empty constraints.
     The constraints will initially have no bounds. */
    if (r == MSK_RES_OK)
      r = MSK_appendcons(task, NUMCON);

    /* Append 'NUMVAR' variables.
     The variables will initially be fixed at zero (x=0). */
    if (r == MSK_RES_OK)
      r = MSK_appendvars(task, NUMVAR);

    /* Optionally add a constant term to the objective. */
    if (r == MSK_RES_OK)
      r = MSK_putcfix(task, 0.0);
    for (j = 0; j < NUMVAR && r == MSK_RES_OK; ++j)
    {
      /* Set the linear term c_j in the objective.*/
      if (r == MSK_RES_OK)
        r = MSK_putcj(task, j, c[j]);
```

(continues on next page)

(continued from previous page)

```
      /* Set the bounds on variable j.
       blx[j] <= x_j <= bux[j] */
      if (r == MSK_RES_OK)
        r = MSK_putvarbound(task,
                            j,           /* Index of variable.*/
                            bkx[j],      /* Bound key.*/
                            blx[j],      /* Numerical value of lower bound.*/
                            bux[j]);     /* Numerical value of upper bound.*/

      /* Input column j of A */
      if (r == MSK_RES_OK)
        r = MSK_putacol(task,
                        j,                 /* Variable (column) index.*/
                        aptre[j] - aptrb[j], /* Number of non-zeros in column j.*/
                        asub + aptrb[j],   /* Pointer to row indexes of column j.*/
                        aval + aptrb[j]);  /* Pointer to Values of column j.*/

    }

    /* Set the bounds on constraints.
       for i=1, ...,NUMCON : blc[i] <= constraint i <= buc[i] */
    for (i = 0; i < NUMCON && r == MSK_RES_OK; ++i)
      r = MSK_putconbound(task,
                          i,           /* Index of constraint.*/
                          bkc[i],      /* Bound key.*/
                          blc[i],      /* Numerical value of lower bound.*/
                          buc[i]);     /* Numerical value of upper bound.*/

    if (r == MSK_RES_OK)
    {
      /*
       * The lower triangular part of the Q^o
       * matrix in the objective is specified.
       */

      qsubi[0] = 0;   qsubj[0] = 0;  qval[0] = 2.0;
      qsubi[1] = 1;   qsubj[1] = 1;  qval[1] = 0.2;
      qsubi[2] = 2;   qsubj[2] = 0;  qval[2] = -1.0;
      qsubi[3] = 2;   qsubj[3] = 2;  qval[3] = 2.0;

      /* Input the Q^o for the objective. */

      r = MSK_putqobj(task, NUMQNZ, qsubi, qsubj, qval);
    }

    if (r == MSK_RES_OK)
    {
      /*
      * The lower triangular part of the Q^0
      * matrix in the first constraint is specified.
      This corresponds to adding the term
      - x_1^2 - x_2^2 - 0.1 x_3^2 + 0.2 x_1 x_3
      */

      qsubi[0] = 0;   qsubj[0] = 0;  qval[0] = -2.0;
      qsubi[1] = 1;   qsubj[1] = 1;  qval[1] = -2.0;
```

(continues on next page)

(continued from previous page)

```
        qsubi[2] = 2;   qsubj[2] = 2;  qval[2] = -0.2;
        qsubi[3] = 2;   qsubj[3] = 0;  qval[3] = 0.2;

        /* Put Q^0 in constraint with index 0. */

        r = MSK_putqconk(task,
                         0,
                         4,
                         qsubi,
                         qsubj,
                         qval);
      }

      if (r == MSK_RES_OK)
        r = MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MINIMIZE);

      if (r == MSK_RES_OK)
      {
        MSKrescodee trmcode;

        /* Run optimizer */
        r = MSK_optimizetrm(task, &trmcode);

        /* Print a summary containing information
           about the solution for debugging purposes*/
        MSK_solutionsummary(task, MSK_STREAM_LOG);

        if (r == MSK_RES_OK)
        {
          MSKsolstae solsta;
          int j;

          MSK_getsolsta(task, MSK_SOL_ITR, &solsta);

          switch (solsta)
          {
            case MSK_SOL_STA_OPTIMAL:
              MSK_getxx(task,
                        MSK_SOL_ITR,    /* Request the interior solution. */
                        xx);

              printf("Optimal primal solution\n");
              for (j = 0; j < NUMVAR; ++j)
                printf("x[%d]: %e\n", j, xx[j]);

              break;

            case MSK_SOL_STA_DUAL_INFEAS_CER:
            case MSK_SOL_STA_PRIM_INFEAS_CER:
              printf("Primal or dual infeasibility certificate found.\n");
              break;

            case MSK_SOL_STA_UNKNOWN:
              printf("The status of the solution could not be determined. Termination␣
↪code: %d.\n", trmcode);
              break;
```

(continues on next page)

(continued from previous page)

```
          default:
            printf("Other solution status.");
            break;
        }
      }
      else
      {
        printf("Error while optimizing.\n");
      }
    }

    if (r != MSK_RES_OK)
    {
      /* In case of an error print error code and description. */
      char symname[MSK_MAX_STR_LEN];
      char desc[MSK_MAX_STR_LEN];

      printf("An error occurred while optimizing.\n");
      MSK_getcodedesc(r,
                      symname,
                      desc);
      printf("Error %s - '%s'\n", symname, desc);
    }
  }

  MSK_deletetask(&task);
}
MSK_deleteenv(&env);

return (r);
} /* main */
```

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:

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

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)

```
MSKint32t       numvar = 3,
                numcon = 3;
MSKint32t       i, j;
double          c[]    = {1.5, 2.5, 3.0};
MSKint32t       ptrb[] = {0, 3, 6},
                ptre[] = {3, 6, 9},
                asub[] = { 0, 1, 2,
                           0, 1, 2,
                           0, 1, 2
                         };

double          aval[] = { 2.0, 3.0, 2.0,
                           4.0, 2.0, 3.0,
                           3.0, 3.0, 2.0
                         };

MSKboundkeye    bkc[]  = {MSK_BK_UP, MSK_BK_UP, MSK_BK_UP    };
double          blc[]  = { -MSK_INFINITY, -MSK_INFINITY, -MSK_INFINITY};
double          buc[]  = {100000, 50000, 60000};

MSKboundkeye    bkx[]  = {MSK_BK_LO,     MSK_BK_LO,     MSK_BK_LO};
double          blx[]  = {0.0,           0.0,           0.0,};
double          bux[]  = { +MSK_INFINITY, +MSK_INFINITY, +MSK_INFINITY};

double          *xx = NULL;
```

(continues on next page)

(continued from previous page)

```
  MSKenv_t        env;
  MSKtask_t       task;
  MSKint32t       varidx, conidx;
  MSKrescodee     r,lr;

  /* Create the mosek environment. */
  r = MSK_makeenv(&env,DEBUG ? "" :  NULL);

  if (r == MSK_RES_OK)
  {
    /* Create the optimization task. */
    r = MSK_maketask(env, numcon, numvar, &task);

    /* Append the constraints. */
    if (r == MSK_RES_OK)
      r = MSK_appendcons(task, numcon);

    /* Append the variables. */
    if (r == MSK_RES_OK)
      r = MSK_appendvars(task, numvar);

    /* Put C. */
    if (r == MSK_RES_OK)
      r = MSK_putcfix(task, 0.0);

    if (r == MSK_RES_OK)
      for (j = 0; j < numvar; ++j)
        r = MSK_putcj(task, j, c[j]);

    /* Put constraint bounds. */
    if (r == MSK_RES_OK)
      for (i = 0; i < numcon; ++i)
        r = MSK_putconbound(task, i, bkc[i], blc[i], buc[i]);

    /* Put variable bounds. */
    if (r == MSK_RES_OK)
      for (j = 0; j < numvar; ++j)
        r = MSK_putvarbound(task, j, bkx[j], blx[j], bux[j]);

    /* Put A. */
    if (r == MSK_RES_OK)
      if (numcon > 0)
        for (j = 0; j < numvar; ++j)
          r = MSK_putacol(task,
                          j,
                          ptre[j] - ptrb[j],
                          asub + ptrb[j],
                          aval + ptrb[j]);

    if (r == MSK_RES_OK)
      r = MSK_putobjsense(task,
                          MSK_OBJECTIVE_SENSE_MAXIMIZE);

    if (r == MSK_RES_OK)
      r = MSK_optimizetrm(task, NULL);
```

(continues on next page)

(continued from previous page)

```
    if (r == MSK_RES_OK)
    {
      xx = calloc(numvar, sizeof(double));
      if (!xx)
        r = MSK_RES_ERR_SPACE;
    }

    if (r == MSK_RES_OK)
      r = MSK_getxx(task,
                    MSK_SOL_BAS,       /* Basic solution.       */
                    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 *MSK_putaij* as shown below.

```
    if (r == MSK_RES_OK)
      r = MSK_putaij(task, 0, 0, 3.0);
```

The problem now has the form:

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

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)

```
    /*********************** Add a new variable ******************/
    /* Get index of new variable, this should be 3 */
    if (r == MSK_RES_OK)
      r = MSK_getnumvar(task, &varidx);
    /* Append a new variable x_3 to the problem */
    if (r == MSK_RES_OK)
    {
      r = MSK_appendvars(task, 1);
      numvar++;
    }
    /* Set bounds on new variable */
    if (r == MSK_RES_OK)
      r = MSK_putvarbound(task,
```

(continues on next page)

(continued from previous page)

```
                           varidx,
                           MSK_BK_LO,
                           0,
                           +MSK_INFINITY);

    /* Change objective */
    if (r == MSK_RES_OK)
      r = MSK_putcj(task, varidx, 1.0);

    /* Put new values in the A matrix */
    if (r == MSK_RES_OK)
    {
      MSKint32t acolsub[] = {0,   2};
      double    acolval[] =  {4.0, 1.0};

      r = MSK_putacol(task,
                       varidx, /* column index */
                       2, /* num nz in column*/
                       acolsub,
                       acolval);
    }
```

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.

```
    /* **************** Add a new constraint ****************** */
    /* Get index of new constraint*/
    if (r == MSK_RES_OK)
      r = MSK_getnumcon(task, &conidx);
```

(continues on next page)

(continued from previous page)

```
    /* Append a new constraint */
    if (r == MSK_RES_OK)
    {
      r = MSK_appendcons(task, 1);
      numcon++;
    }

    /* Set bounds on new constraint */
    if (r == MSK_RES_OK)
      r = MSK_putconbound(task,
                          conidx,
                          MSK_BK_UP,
                          -MSK_INFINITY,
                          30000);

    /* Put new values in the A matrix */
    if (r == MSK_RES_OK)
    {
      MSKint32t arowsub[] = {0,   1,   2,   3  };
      double    arowval[] =  {1.0, 2.0, 1.0, 1.0};

      r = MSK_putarow(task,
                      conidx, /* row index */
                      4,      /* num nz in row*/
                      arowsub,
                      arowval);
    }
```

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:

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

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.

```
    /* **************** Change constraint bounds ******************* */
    if (r == MSK_RES_OK)
    {
      MSKboundkeye newbkc[]  = { MSK_BK_UP, MSK_BK_UP, MSK_BK_UP, MSK_BK_UP };
```

(continues on next page)

(continued from previous page)

```
    double       newblc[]  = { -MSK_INFINITY, -MSK_INFINITY, -MSK_INFINITY, -MSK_
↪INFINITY };
    double       newbuc[]  = { 80000, 40000, 50000, 22000 };

    r = MSK_putconboundslice(task, 0, numcon, newbkc, newblc, newbuc);
  }
```

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 *MSK_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.

```
/** Example of how to use MSK_optimizebatch().
    Optimizes tasks whose names were read from command line.
*/
int main(int argc, char **argv)
{
  MSKenv_t env;
  int n = argc - 1;
  MSKtask_t   *tasks = NULL;
  MSKrescodee *res   = NULL;
  MSKrescodee *trm   = NULL;
  MSKrescodee r = MSK_RES_OK;
  int i;
  /* Size of thread pool available for all tasks */
  int threadpoolsize = 6;

  tasks = calloc(n, sizeof(MSKtask_t));
  res   = calloc(n, sizeof(MSKrescodee));
  trm   = calloc(n, sizeof(MSKrescodee));

  MSK_makeenv(&env, NULL);
```

(continues on next page)

(continued from previous page)

```
  /* Create an example list of tasks to optimize */
  for (i = 0; i < n; i++) {
    MSK_makeemptytask(env, &(tasks[i]));
    MSK_readdata(tasks[i], argv[i+1]);
    /* We can set the number of threads for each task */
    MSK_putintparam(tasks[i], MSK_IPAR_NUM_THREADS, 2);
  }

  /* Optimize all the given tasks in parallel */
  r = MSK_optimizebatch(env,
                        0,                // No race
                        -1.0,             // No time limit
                        threadpoolsize,
                        n,
                        tasks,            // Array of tasks to optimize
                        trm,
                        res);

  for(i = 0; i < n; i++) {
    double obj, tm;
    MSK_getdouinf(tasks[i], MSK_DINF_INTPNT_PRIMAL_OBJ, &obj);
    MSK_getdouinf(tasks[i], MSK_DINF_OPTIMIZER_TIME, &tm);

    printf("Task  %d  res %d   trm %d   obj_val  %.5f  time %.5f\n",
            i,
            res[i],
            trm[i],
            obj,
            tm);
  }

  for(i = 0; i < n; i++)
    MSK_deletetask(&(tasks[i]));
  free(tasks);
  free(trm);
  free(res);
  MSK_deleteenv(&env);
  return 0;
}
```

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 C, 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 C 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

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

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
  {
    MSKprostae prosta;
    if (r == MSK_RES_OK)
      r = MSK_getprosta(task, MSK_SOL_ITR, &prosta);

    if (r == MSK_RES_OK && prosta == MSK_PRO_STA_PRIM_INFEAS) {
      // Set the tolerance at which we consider a dual value as essential
      double 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
// n - length of arrays
// sl - dual values for lower bounds
// su - dual values for upper bounds
// eps - tolerance for when a nunzero dual value is significant
static void analyzeCertificate(MSKint32t n, MSKrealt *sl, MSKrealt *su, double eps) {
  MSKint32t i;
  for(i = 0; i < n; i++) {
    if (abs(sl[i]) > eps)
      printf("#%d, lower,  dual = %e\n", i, sl[i]);
    if (abs(su[i]) > eps)
      printf("#%d, upper,  dual = %e\n", i, su[i]);
  }
}
```

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.

```
#include <stdio.h>
#include "mosek.h"

/* This function prints log output from MOSEK to the terminal. */
static void MSKAPI printstr(void       *handle,
                           const char str[])
{
  printf("%s", str);
} /* printstr */

// Set up a simple linear problem from the manual for test purposes
static MSKrescodee testProblem(MSKtask_t *task) {
  MSKrescodee r = MSK_RES_OK;
  const char data[] = "Task ''\n\
Objective ''\n\
    Minimize + @x0 + 2 @x1 + 5 @x2 + 2 @x3 + @x4 + 2 @x5 + @x6\n\
Constraints\n\
    @c0 [-inf;200] + @x0 + @x1\n\
    @c1 [-inf;1000] + @x2 + @x3\n\
    @c2 [-inf;1000] + @x4 + @x5 + @x6\n\
    @c3 [1100] + @x0 + @x4\n\
    @c4 [200] + @x1\n\
    @c5 [500] + @x2 + @x5\n\
    @c6 [500] + @x3 + @x6\n\
Variables\n\
    @x0 [0;+inf]\n\
    @x1 [0;+inf]\n\
    @x2 [0;+inf]\n\
    @x3 [0;+inf]\n\
    @x4 [0;+inf]\n\
    @x5 [0;+inf]\n\
    @x6 [0;+inf]\n";

  r = MSK_makeemptytask(NULL, task);
  if (r == MSK_RES_OK)
    r = MSK_readptfstring(*task, data);
  return r;
}

// Analyzes and prints infeasibility contributing elements
// n - length of arrays
// sl - dual values for lower bounds
// su - dual values for upper bounds
// eps - tolerance for when a nunzero dual value is significant
static void analyzeCertificate(MSKint32t n, MSKrealt *sl, MSKrealt *su, double eps) {
  MSKint32t i;
  for(i = 0; i < n; i++) {
    if (abs(sl[i]) > eps)
      printf("#%d, lower,  dual = %e\n", i, sl[i]);
    if (abs(su[i]) > eps)
```

(continues on next page)

(continued from previous page)

```
      printf("#%d, upper,  dual = %e\n", i, su[i]);
  }
}

int main(int argc, const char *argv[])
{
  MSKtask_t           task;
  MSKrescodee         r = MSK_RES_OK;
  double              inf = 0.0;

  // In this example we set up a simple problem
  // One could use any task or a task read from a file
  r = testProblem(&task);

  // Useful for debugging
  if (r == MSK_RES_OK)
    r = MSK_writedata(task, "pinfeas.ptf");                          // Write file in␣
↪human-readable format
  // Directs the log task stream to the 'printstr' function.
  if (r == MSK_RES_OK)
    r = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

  // Perform the optimization.
  if (r == MSK_RES_OK)
    r = MSK_optimize(task);
  if (r == MSK_RES_OK)
    r = MSK_solutionsummary(task, MSK_STREAM_LOG);

  // Check problem status, we use the interior point solution
  {
    MSKprostae prosta;
    if (r == MSK_RES_OK)
      r = MSK_getprosta(task, MSK_SOL_ITR, &prosta);

    if (r == MSK_RES_OK && prosta == MSK_PRO_STA_PRIM_INFEAS) {
      // Set the tolerance at which we consider a dual value as essential
      double eps = 1e-7;

      printf("Variable bounds important for infeasibility: \n");
      if (r == MSK_RES_OK) {
        MSKint32t m;
        r = MSK_getnumvar(task, &m);
        if (r == MSK_RES_OK) {
          MSKrealt *slx = NULL, *sux = NULL;
          slx = (MSKrealt*) calloc(m, sizeof(MSKrealt));
          sux = (MSKrealt*) calloc(m, sizeof(MSKrealt));

          if (r == MSK_RES_OK) r = MSK_getslx(task, MSK_SOL_ITR, slx);
          if (r == MSK_RES_OK) r = MSK_getsux(task, MSK_SOL_ITR, sux);

          if (r == MSK_RES_OK)
            analyzeCertificate(m, slx, sux, eps);

          free(sux);
          free(slx);
        }
```

(continues on next page)

(continued from previous page)

```
      }

      printf("Constraint bounds important for infeasibility: \n");
      if (r == MSK_RES_OK) {
        MSKint32t n;
        r = MSK_getnumcon(task, &n);
        if (r == MSK_RES_OK) {
          MSKrealt *slc = NULL, *suc = NULL;
          slc = (MSKrealt*) calloc(n, sizeof(MSKrealt));
          suc = (MSKrealt*) calloc(n, sizeof(MSKrealt));

          if (r == MSK_RES_OK) r = MSK_getslc(task, MSK_SOL_ITR, slc);
          if (r == MSK_RES_OK) r = MSK_getsuc(task, MSK_SOL_ITR, suc);

          if (r == MSK_RES_OK)
            analyzeCertificate(n, slc, suc, eps);

          free(suc);
          free(slc);
        }
      }
    }
    else {
      printf("The problem is not primal infeasible, no certificate to show.\n");
    }
  }

  MSK_deletetask(&task);
  return r;
}
```

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 C 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:

```
MSKrescodee res = MSK_RES_OK;
MSKtask_t  task = NULL;

/* Create the task */
res = MSK_makeemptytask(NULL,     // NULL indicates we use the global environment
                        &task);

if (res == MSK_RES_OK) {
  /* Define and solve the optimization problem here
   * ...
   */
}
else printf("An error %d while creating the task\n", res);

/* Dispose of the task */
if (task) MSK_deletetask(&task);
```

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 passing `NULL` in any function call that takes an environment argument.

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

```
MSKrescodee res = MSK_RES_OK;
MSKenv_t    env = NULL;

/* Create the environment */
res = MSK_makeenv(&env, NULL);

if (res == MSK_RES_OK) {
  /* Now we create a task. We could create more tasks. */
  MSKtask_t task = NULL;
  res = MSK_makeemptytask(env, &task);

  if (res == MSK_RES_OK) {
    /* Define and solve the optimization problem here
     * ...
     */
  }
  else printf("An error %d while creating the task\n", res);

  /* Dispose of the task */
  if (task) MSK_deletetask(&task);
}
else printf("An error %d while creating the environment\n", res);

/* When we stop using MOSEK then dispose of the environment */
if (env) MSK_deleteenv(&env);
```

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 *MSKrescodee*. 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).

To obtain both codes separately call the function *MSK_optimizetrm* to optimize the problem. When using the simplified *MSK_optimize* the response code or termination code most relevant for the user will be returned.

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. There is one for every type of solution, as explained above.

Problem status

Problem status (*MSKprostae*) 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 (*MSKsolstae*) 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 *MSK_getprosta* and *MSK_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_FEAS*	*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:

- *MSK_getprimalobj*, *MSK_getdualobj* — the primal and dual objective value.
- *MSK_getxx* — solution values for the variables.
- *MSK_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.

```
#include <stdio.h>
#include "mosek.h"

/* Log handler */
void MSKAPI printlog(void *ptr,
                     const char s[])
{
  printf("%s", s);
}

int main(int argc, char const *argv[])
{
  MSKenv_t    env;
  MSKtask_t   task;
  MSKrescodee r;
  char        symname[MSK_MAX_STR_LEN];
  char        desc[MSK_MAX_STR_LEN];
```

(continues on next page)

(continued from previous page)

```
  int         i, numvar;
  double      *xx = NULL;
  const char  *filename;

  if (argc >= 2) filename = argv[1];
  else           filename = "../data/cqo1.mps";

  // Create the environment
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    // Create the task
    r = MSK_makeemptytask(env, &task);

    // (Optionally) attach the log handler to receive log information
    // if ( r == MSK_RES_OK ) MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL,␣
↪printlog);

    // (Optionally) uncomment this line to most likely see solution status Unknown
    // MSK_putintparam(task, MSK_IPAR_INTPNT_MAX_ITERATIONS, 1);

    // In this example we read an optimization problem from a file
    r = MSK_readdata(task, filename);

    if (r == MSK_RES_OK)
    {
      MSKrescodee trmcode;
      MSKsolstae  solsta;

      // Do the optimization, and exit in case of error
      r = MSK_optimizetrm(task, &trmcode);

      if (r != MSK_RES_OK) {
        MSK_getcodedesc(r, symname, desc);
        printf("Error during optimization: %s %s\n", symname, desc);
        exit(r);
      }

      MSK_solutionsummary(task, MSK_STREAM_LOG);

      /* Expected result: The solution status of the interiot-point solution is␣
↪optimal. */

      if (MSK_RES_OK == MSK_getsolsta(task, MSK_SOL_ITR, &solsta))
      {
        switch (solsta)
        {
          case MSK_SOL_STA_OPTIMAL:
            printf("An optimal interior-point solution is located.\n");

            /* Read and print the variable values in the solution */
            MSK_getnumvar(task, &numvar);
            xx = calloc(numvar, sizeof(double));
            MSK_getxx(task, MSK_SOL_ITR, xx);
            for (i = 0; i < numvar; i++)
```

(continues on next page)

(continued from previous page)

```
              printf("xx[%d] = %.4lf\n", i, xx[i]);
            free(xx);
            break;

          case MSK_SOL_STA_DUAL_INFEAS_CER:
            printf("Dual infeasibility certificate found.\n");
            break;

          case MSK_SOL_STA_PRIM_INFEAS_CER:
            printf("Primal infeasibility certificate found.\n");
            break;

          case MSK_SOL_STA_UNKNOWN:
            /* The solutions status is unknown. The termination code
               indicating why the optimizer terminated prematurely. */
            printf("The solution status is unknown.\n");
            /* No-error cause of termination e.g. an iteration limit is reached.  */
            MSK_getcodedesc(trmcode, symname, desc);
            printf("  Termination code: %s %s\n", symname, desc);
            break;

          default:
            MSK_solstatostr(task, solsta, desc);
            printf("An unexpected solution status %s with code %d is obtained.\n",␣
↪desc, solsta);
            break;
        }
      }
      else
        printf("Could not obtain the solution status for the requested solution.\n");
    }
    else {
      MSK_getcodedesc(r, symname, desc);
      printf("Optimization was not started because of error %s(%d): %s\n", symname, r,
↪ desc);
    }

    MSK_deletetask(&task);
  }

  MSK_deleteenv(&env);
  return r;
}
```

7.3 Errors and exceptions

Response codes

Almost every function in Optimizer API for C returns a **response code**, which is an integer (implemented as the enum *MSKrescodee*), informing if the requested operation was performed correctly, and if not, what error occurred. The expected response, indicating successful execution, is always *MSK_RES_OK*. It is a good idea to check the response code every time to avoid silent fails 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.

The one case where it is *extremely important* to check the response code is during optimization, when *MSK_optimizetrm* is invoked. We will say more about this in Sec. 7.2.

A numerical response code can be converted into a human-readable description using *MSK_getcodedesc*. A full list of response codes, error, warning and termination codes can be found in the *API reference*. For example, the following code

```
res = MSK_putdouparam(task, MSK_DPAR_INTPNT_CO_TOL_REL_GAP, -1.0e-7);
if (res != MSK_RES_OK) {
  MSK_getcodedesc(res, symb, str);
  printf("Error %s(%d): %s\n", symb, res, str);
}
```

will produce as output:

```
Error MSK_RES_ERR_PARAM_IS_TOO_SMALL(1216): A parameter value is too small.
```

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.

The user can also register a dedicated callback function to handle all errors and warnings. This is done with *MSK_putresponsefunc*.

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.

```
#include <stdio.h>
#include "mosek.h"

/* Log handler */
void MSKAPI printlog(void *ptr,
                     const char s[])
{
  printf("%s", s);
}

int main(int argc, char const *argv[])
{
  MSKenv_t    env;
  MSKtask_t   task;
  MSKrescodee r;
  char        symname[MSK_MAX_STR_LEN];
  char        desc[MSK_MAX_STR_LEN];
  int         i, numvar;
```

(continues on next page)

(continued from previous page)

```
  double      *xx = NULL;
  const char  *filename;

  if (argc >= 2) filename = argv[1];
  else           filename = "../data/cqo1.mps";

  // Create the environment
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    // Create the task
    r = MSK_makeemptytask(env, &task);

    // (Optionally) attach the log handler to receive log information
    // if ( r == MSK_RES_OK ) MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL,␣
↪printlog);

    // (Optionally) uncomment this line to most likely see solution status Unknown
    // MSK_putintparam(task, MSK_IPAR_INTPNT_MAX_ITERATIONS, 1);

    // In this example we read an optimization problem from a file
    r = MSK_readdata(task, filename);

    if (r == MSK_RES_OK)
    {
      MSKrescodee trmcode;
      MSKsolstae  solsta;

      // Do the optimization, and exit in case of error
      r = MSK_optimizetrm(task, &trmcode);

      if (r != MSK_RES_OK) {
        MSK_getcodedesc(r, symname, desc);
        printf("Error during optimization: %s %s\n", symname, desc);
        exit(r);
      }

      MSK_solutionsummary(task, MSK_STREAM_LOG);

      /* Expected result: The solution status of the interiot-point solution is␣
↪optimal. */

      if (MSK_RES_OK == MSK_getsolsta(task, MSK_SOL_ITR, &solsta))
      {
        switch (solsta)
        {
          case MSK_SOL_STA_OPTIMAL:
            printf("An optimal interior-point solution is located.\n");

            /* Read and print the variable values in the solution */
            MSK_getnumvar(task, &numvar);
            xx = calloc(numvar, sizeof(double));
            MSK_getxx(task, MSK_SOL_ITR, xx);
            for (i = 0; i < numvar; i++)
              printf("xx[%d] = %.4lf\n", i, xx[i]);
```

(continues on next page)

(continued from previous page)

```
            free(xx);
            break;

          case MSK_SOL_STA_DUAL_INFEAS_CER:
            printf("Dual infeasibility certificate found.\n");
            break;

          case MSK_SOL_STA_PRIM_INFEAS_CER:
            printf("Primal infeasibility certificate found.\n");
            break;

          case MSK_SOL_STA_UNKNOWN:
            /* The solutions status is unknown. The termination code
               indicating why the optimizer terminated prematurely. */
            printf("The solution status is unknown.\n");
            /* No-error cause of termination e.g. an iteration limit is reached.  */
            MSK_getcodedesc(trmcode, symname, desc);
            printf("  Termination code: %s %s\n", symname, desc);
            break;

          default:
            MSK_solstatostr(task, solsta, desc);
            printf("An unexpected solution status %s with code %d is obtained.\n",␣
↪desc, solsta);
            break;
        }
      }
      else
        printf("Could not obtain the solution status for the requested solution.\n");
    }
    else {
      MSK_getcodedesc(r, symname, desc);
      printf("Optimization was not started because of error %s(%d): %s\n", symname, r,
↪ desc);
    }

    MSK_deletetask(&task);
  }

  MSK_deleteenv(&env);
  return r;
}
```

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 user-defined function of type *MSKstreamfunc* that accepts a string, for example:

```
static void MSKAPI printstr(void       *handle,
                            const char *str)
{
  printf("%s", str);
  fflush(stdout);
}
```

It is attached to a stream as follows:

```
MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);
```

The stream can be detached by calling

```
MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, NULL);
```

A log stream can also be redirected to a file:

```
MSK_linkfiletotaskstream(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 *MSK_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*, and
- *MSK_IPAR_LOG_SIM_MINOR*.

Each parameter controls the output level of a specific functionality or algorithm. The main switch is *MSK_IPAR_LOG* which 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 *MSK_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

```
MSK_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

```
MSK_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 *MSK_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 *MSK_readsolution*). If the file contains parameters, they will be set accordingly.

```
res = MSK_maketask(env, 0,0, &task);
if (res == MSK_RES_OK)
  res = MSK_readdata(task, "file.task.gz");
if (res == MSK_RES_OK)
  res = MSK_optimize(task);
```

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 *MSK_putintparam*.
- Double (floating point) parameters. Set with *MSK_putdouparam*.
- String parameters. Set with *MSK_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 parameters which choose and tune the interior point optimizer before solving a problem.

Listing 7.3: Parameter setting example.

```
    // Set log level (integer parameter)
    res = MSK_putintparam(task, MSK_IPAR_LOG, 1);
    // Select interior-point optimizer... (integer parameter)
    res = MSK_putintparam(task, MSK_IPAR_OPTIMIZER, MSK_OPTIMIZER_INTPNT);
    // ... without basis identification (integer parameter)
    res = MSK_putintparam(task, MSK_IPAR_INTPNT_BASIS, MSK_BI_NEVER);
    // Set relative gap tolerance (double parameter)
    res = MSK_putdouparam(task, MSK_DPAR_INTPNT_CO_TOL_REL_GAP, 1.0e-7);

    // The same using explicit string names
    res = MSK_putparam(task, "MSK_DPAR_INTPNT_CO_TOL_REL_GAP", "1.0e-7");
    res = MSK_putnadouparam(task, "MSK_DPAR_INTPNT_CO_TOL_REL_GAP",  1.0e-7);

    // Incorrect value
    res = MSK_putdouparam(task, MSK_DPAR_INTPNT_CO_TOL_REL_GAP,  -1.0);
    if (res != MSK_RES_OK)
      printf("Wrong parameter value\n");
```

Reading parameter values

The functions *MSK_getintparam*, *MSK_getdouparam*, *MSK_getstrparam* can be used to inspect the current value of a parameter, for example:

```
    res = MSK_getdouparam(task, MSK_DPAR_INTPNT_CO_TOL_REL_GAP, &param);
    printf("Current value for parameter MSK_DPAR_INTPNT_CO_TOL_REL_GAP = %e\n",␣
↪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

- *MSK_getdouinf* for a double information item,
- *MSK_getintinf* for an integer information item,
- *MSK_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.

```
    res = MSK_getdouinf(task, MSK_DINF_OPTIMIZER_TIME,     &tm);
    res = MSK_getintinf(task, MSK_IINF_INTPNT_ITER,        &iter);

    printf("Time: %f\nIterations: %d\n", tm, 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. The example in Listing 7.5 shows how to do it by handling the callback code *MSK_CALLBACK_NEW_INT_MIO*.

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 function *MSK_putcallbackfunc* and providing a handle to a user-defined function *MSKcallbackfunc*.

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.

```
static int MSKAPI usercallback(MSKtask_t            task,
                               MSKuserhandle_t      handle,
                               MSKcallbackcodee     caller,
                               const MSKrealt   * douinf,
                               const MSKint32t * intinf,
                               const MSKint64t * lintinf)
{
  cbdata_t data = (cbdata_t) handle;
  double maxtime = data->maxtime;
  MSKrescodee r;

  switch (caller)
  {
    case MSK_CALLBACK_BEGIN_INTPNT:
      printf("Starting interior-point optimizer\n");
      break;
    case MSK_CALLBACK_INTPNT:
      printf("Iterations: %-3d  Time: %6.2f(%.2f)   ",
             intinf[MSK_IINF_INTPNT_ITER],
             douinf[MSK_DINF_OPTIMIZER_TIME],
             douinf[MSK_DINF_INTPNT_TIME]);

      printf("Primal obj.: %-18.6e  Dual obj.: %-18.6e\n",
             douinf[MSK_DINF_INTPNT_PRIMAL_OBJ],
             douinf[MSK_DINF_INTPNT_DUAL_OBJ]);
      break;
    case MSK_CALLBACK_END_INTPNT:
      printf("Interior-point optimizer finished.\n");
      break;
    case MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX:
      printf("Primal simplex optimizer started.\n");
      break;
    case MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX:
```

(continues on next page)

(continued from previous page)

```
      printf("Iterations: %-3d  ",
             intinf[MSK_IINF_SIM_PRIMAL_ITER]);
      printf("  Elapsed time: %6.2f(%.2f)\n",
             douinf[MSK_DINF_OPTIMIZER_TIME],
             douinf[MSK_DINF_SIM_TIME]);
      printf("Obj.: %-18.6e\n",
             douinf[MSK_DINF_SIM_OBJ]);
      break;
    case MSK_CALLBACK_END_PRIMAL_SIMPLEX:
      printf("Primal simplex optimizer finished.\n");
      break;
    case MSK_CALLBACK_BEGIN_DUAL_SIMPLEX:
      printf("Dual simplex optimizer started.\n");
      break;
    case MSK_CALLBACK_UPDATE_DUAL_SIMPLEX:
      printf("Iterations: %-3d  ", intinf[MSK_IINF_SIM_DUAL_ITER]);
      printf("  Elapsed time: %6.2f(%.2f)\n",
             douinf[MSK_DINF_OPTIMIZER_TIME],
             douinf[MSK_DINF_SIM_TIME]);
      printf("Obj.: %-18.6e\n", douinf[MSK_DINF_SIM_OBJ]);
      break;
    case MSK_CALLBACK_END_DUAL_SIMPLEX:
      printf("Dual simplex optimizer finished.\n");
      break;
    case MSK_CALLBACK_NEW_INT_MIO:
      printf("New integer solution has been located.\n");

      r = MSK_getxx(task, MSK_SOL_ITG, data->xx);
      if (r == MSK_RES_OK) {
        int i;
        printf("xx = ");
        for (i = 0; i < data->numvars; i++) printf("%lf ", data->xx[i]);
        printf("\nObj.: %f\n", douinf[MSK_DINF_MIO_OBJ_INT]);
      }
    default:
      break;
  }

  if (douinf[MSK_DINF_OPTIMIZER_TIME] >= maxtime)
  {
    /* mosek is spending too much time.
       Terminate it. */
    return (1);
  }

  return (0);
} /* usercallback */
```

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.

```
      data.maxtime = 0.05;
      MSK_getnumvar(task, &data.numvars);
      data.xx = MSK_callocenv(env, data.numvars, sizeof(double));

```

(continues on next page)

(continued from previous page)

```
    MSK_putcallbackfunc(task,
                        usercallback,
                        (void *) &data);
```

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.

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.

```
#include "mosek.h"

static void MSKAPI printstr(void *handle, const char str[])
{
  printf("%s", str);
}

int main(int argc, const char * argv[])
{
  MSKenv_t    env  = NULL;
  MSKtask_t   task = NULL;
  MSKrescodee res  = MSK_RES_OK;
  MSKrescodee trm  = MSK_RES_OK;

  if (argc <= 3)
  {
    fprintf(stderr, "Syntax: opt_server_sync inputfile addr [certpath]\n");
    return 0;
  }
  else
  {
    const char * taskfile = argv[1];
    const char * address  = argv[2];
    const char * certfile = argc > 3 ? argv[3]  : NULL;

    // Create the mosek environment.
    // The `NULL' arguments here, are used to specify customized
    // memory allocators and a memory debug file. These can
    // safely be ignored for now.
    res = MSK_makeenv(&env, NULL);

    // Create a task object linked with the environment env.
```

(continues on next page)

(continued from previous page)

```
    // We create it with 0 variables and 0 constraints initially,
    // since we do not know the size of the problem.
    if (res == MSK_RES_OK)
      res = MSK_maketask(env, 0, 0, &task);

    // Direct the task log stream to a user specified function
    if (res == MSK_RES_OK)
      res = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

    // We assume that a problem file was given as the first command
    // line argument (received in `argv')
    if (res == MSK_RES_OK)
      res = MSK_readdata(task, taskfile);

    // Set OptServer URL
    if (res == MSK_RES_OK)
      res = MSK_putoptserverhost(task, address);

    // Path to certificate, if any
    if (MSK_RES_OK == res && certfile)
      res = MSK_putstrparam(task, MSK_SPAR_REMOTE_TLS_CERT_PATH,certfile);

    // Solve the problem remotely
    if (res == MSK_RES_OK)
      res = MSK_optimizetrm(task, &trm);

    // Print a summary of the solution.
    if (res == MSK_RES_OK)
      res = MSK_solutionsummary(task, MSK_STREAM_LOG);

    // Delete task and environment
    MSK_deletetask(&task);
    MSK_deleteenv(&env);
  }
  return res;
}
```

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:

- *MSK_asyncoptimize* : Offload the optimization task to a solver server.
- *MSK_asyncpoll* : Request information about the status of the remote job.
- *MSK_asyncgetresult* : Request the results from a completed remote job.
- *MSK_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.

```
#include "mosek.h"
#ifdef _WIN32
#include "windows.h"
#else
#include "unistd.h"
#endif

static void MSKAPI printstr(void *handle, const char str[])
{
  printf("%s", str);
}

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

  char token[33];

  int         numpolls = 10;
  int         i = 0;

  MSKbooleant respavailable;

  MSKenv_t    env   = NULL;
  MSKtask_t   task  = NULL;

  MSKrescodee res   = MSK_RES_OK;
  MSKrescodee trm;
  MSKrescodee resp;

  const char * filename = "../data/25fv47.mps";
  const char * addr     = "solve.mosek.com:30080";
  const char * cert = NULL;

  if (argc < 4)
  {
    fprintf(stderr, "Syntax: opt_server_async filename host:port numpolls [cert]\n");
    return 0;
  }

  filename = argv[1];
  addr     = argv[2];
  numpolls = atoi(argv[3]);
  cert     = argc < 5 ? NULL : argv[4];

  res = MSK_makeenv(&env, NULL);

  if (res == MSK_RES_OK)
    res = MSK_maketask(env, 0, 0, &task);

  if (res == MSK_RES_OK)
    res = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);
```

(continues on next page)

(continued from previous page)

```
  if (res == MSK_RES_OK)
    res = MSK_readdata(task, filename);

  if (MSK_RES_OK == res && cert)
    res = MSK_putstrparam(task, MSK_SPAR_REMOTE_TLS_CERT_PATH,cert);

  res = MSK_asyncoptimize(task,
                          addr,
                          "",
                          token);
  MSK_deletetask(&task);
  printf("token = %s\n", token);

  if (res == MSK_RES_OK)
    res = MSK_maketask(env, 0, 0, &task);

  if (res == MSK_RES_OK)
    res = MSK_readdata(task, filename);

  if (MSK_RES_OK == res && cert)
    res = MSK_putstrparam(task, MSK_SPAR_REMOTE_TLS_CERT_PATH,cert);

  if (res == MSK_RES_OK)
    res = MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

  for (i = 0; i < numpolls &&  res == MSK_RES_OK ; i++)
  {
#if __linux__
    sleep(1);
#elif defined(_WIN32)
    Sleep(1000);
#endif

    puts("++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
    printf("poll %d\n ", i);


    res = MSK_asyncpoll(task,
                        addr,
                        "",
                        token,
                        &respavailable,
                        &resp,
                        &trm);

    puts("polling done\n");
    puts("++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");

    if (respavailable)
    {
      puts("solution available!");

      res = MSK_asyncgetresult(task,
                               addr,
                               "",
```

(continues on next page)

(continued from previous page)

```
                                token,
                                &respavailable,
                                &resp,
                                &trm);

      MSK_solutionsummary(task, MSK_STREAM_LOG);
      break;
    }

  }


  if (i == numpolls)
  {
    printf("max num polls reached, stopping %s", addr);
    MSK_asyncstop(task, addr, "", token);
  }

  MSK_deletetask(&task);
  MSK_deleteenv(&env);

  printf("%s:%d: Result = %d\n", __FILE__, __LINE__, res); fflush(stdout);

  return res;
}
```

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:

  Create a log handler function:

  ```
  static void MSKAPI printstr(void       *handle,
                              const char *str)
  {
    printf("%s", str);
    fflush(stdout);
  }
  ```

  attach it to the log stream:

  ```
  MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);
  ```

  and include solution summary after the call to `optimize`:

  ```
    res = MSK_optimize(task);
    MSK_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.

    ```
        MSK_writedata(task,"data.ptf");
    ```

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

    ```
        MSK_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. 13.4.4.

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
  Disjunctive cons.       : 0
  Cones                   : 0
  Scalar variables        : 20693
  Matrix variables        : 0
  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 *`MSK_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 *MSK_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.

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

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

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 C, 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.

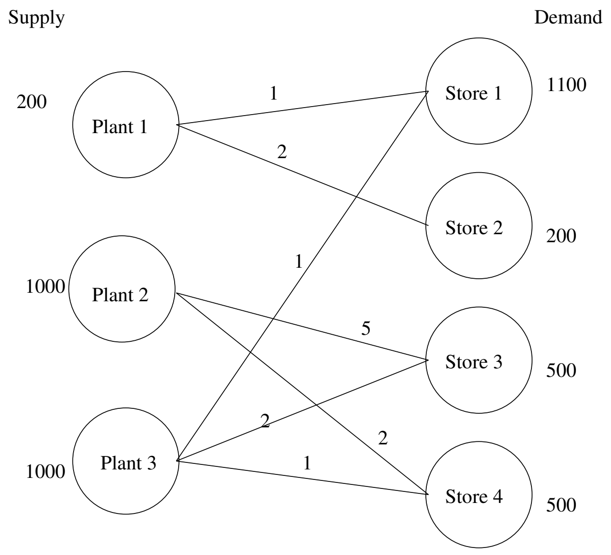

Fig. 8.1: Supply, demand and cost of transportation.

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:

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

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

```
MOSEK PRIMAL INFEASIBILITY REPORT.

Problem status: The problem is primal infeasible

The following constraints are involved in the primal infeasibility.

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

The following bound constraints are involved in the infeasibility.

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

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

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

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

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

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

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

8.3.3 Locating dual infeasibility

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

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

which is dual to (8.1) (and therefore is dual infeasible). The dual infeasibility report may look as follows:

```
MOSEK DUAL INFEASIBILITY REPORT.

Problem status: The problem is dual infeasible

The following constraints are involved in the infeasibility.

Index     Name              Activity          Objective         Lower bound       Upper␣
↪bound
5         x33               -1.000000e+00                       NONE              2.
↪000000e+00
6         x34               -1.000000e+00                       NONE              1.
↪000000e+00

The following variables are involved in the infeasibility.

Index     Name              Activity          Objective         Lower bound       Upper␣
↪bound
0         y1                -1.000000e+00     2.000000e+02      NONE              0.
↪000000e+00
2         y3                -1.000000e+00     1.000000e+03      NONE              0.
↪000000e+00
3         y4                1.000000e+00      1.100000e+03      NONE              NONE
4         y5                1.000000e+00      2.000000e+02      NONE              NONE

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

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

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

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
`info [name]`	Get an information item
`anapro`	Analyze problem data
`hist`	Plot a histogram of problem data
`histsol`	Plot a histogram of the solutions
`spy`	Plot the sparsity pattern of the data matrices
`truncate epsilon`	Truncate small coefficients down to 0
`resobj [fac]`	Rescale objective by a factor
`anasol`	Analyze solutions
`removeitg`	Remove integrality constraints
`removecones`	Remove all cones and leave just the linear part
`infsub`	Replace current problem with its infeasible subproblem
`writesol basename`	Write solution(s) to file(s) with given basename
`delsol`	Remove all solutions from the task
`optserver [url]`	Use an OptServer to optimize
`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

$$\begin{array}{lccccl} \text{maximize} & & & c^T x & & \\ \text{subject to} & & & Ax - x^c & = & 0, \\ & l^x & \leq & x & \leq & u^x, \\ & l^c & \leq & x^c & \leq & u^c. \end{array} \tag{9.3}$$

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 *MSK_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:

$$\begin{array}{lccl}\text{minimize} & x_0 + x_1 & & \\ \text{subject to} & x_0 + 2x_1 & \leq & 2, \\ & x_0 + x_1 & \leq & 6, \\ & x_0, x_1 \geq 0. & & \end{array} \tag{9.4}$$

Suppose a call to *MSK_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

$$\left[\begin{array}{cc} 0 & 1 \\ -1 & 1 \end{array}\right].$$

Please note the ordering of the columns in B .

Listing 9.1: A program showing how to identify the basis.

```
#include "mosek.h"

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char *argv[])
```

(continues on next page)

(continued from previous page)

```
{
  MSKenv_t     env;
  MSKtask_t    task;
  MSKint32t    numcon = 2, numvar = 2;
  double       c[]    = {1.0, 1.0};
  MSKint32t    ptrb[] = {0, 2},
                        ptre[] = {2, 3};
  MSKint32t    asub[] = {0, 1, 0, 1};
  double       aval[] = {1.0, 1.0, 2.0, 1.0};
  MSKboundkeye bkc[]  = { MSK_BK_UP, MSK_BK_UP };
  double       blc[]  = { -MSK_INFINITY, -MSK_INFINITY };
  double       buc[]  = {2.0, 6.0};

  MSKboundkeye bkx[]  = { MSK_BK_LO, MSK_BK_LO };
  double       blx[]  = {0.0, 0.0};
  double       bux[]  = { +MSK_INFINITY, +MSK_INFINITY};
  MSKrescodee  r      = MSK_RES_OK;
  MSKint32t    i, nz;
  double       w[]    = {2.0, 6.0};
  MSKint32t    sub[]  = {0, 1};
  MSKint32t    *basis;

  if (r == MSK_RES_OK)
    r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
    r = MSK_makeemptytask(env, &task);

  if (r == MSK_RES_OK)
    MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

  if (r == MSK_RES_OK)
    r = MSK_inputdata(task, numcon, numvar, numcon, numvar,
                      c, 0.0,
                      ptrb, ptre, asub, aval, bkc, blc, buc, bkx, blx, bux);

  if (r == MSK_RES_OK)
    r = MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE);

  if (r == MSK_RES_OK)
    r = MSK_optimizetrm(task, NULL);

  if (r == MSK_RES_OK)
    basis = MSK_calloctask(task, numcon, sizeof(MSKint32t));

  if (r == MSK_RES_OK)
    r = MSK_initbasissolve(task, basis);

  /* List basis variables corresponding to columns of B */
  for (i = 0; i < numcon && r == MSK_RES_OK; ++i)
  {
    printf("basis[%d] = %d\n", i, basis[i]);
    if (basis[sub[i]] < numcon)
      printf("Basis variable no %d is xc%d.\n", i, basis[i]);
    else
      printf("Basis variable no %d is x%d.\n", i, basis[i] - numcon);
```

(continues on next page)

(continued from previous page)

```
  }

  nz = 2;
  /* solve Bx = w */
  /* sub contains index of non-zeros in w.
     On return w contains the solution x and sub
     the index of the non-zeros in x.
   */
  if (r == MSK_RES_OK)
    r = MSK_solvewithbasis(task, 0, nz, sub, w, &nz);

  if (r == MSK_RES_OK)
  {
    printf("\nSolution to Bx = w:\n\n");

    /* Print solution and b. */

    for (i = 0; i < nz; ++i)
    {
      if (basis[sub[i]] < numcon)
        printf("xc%d = %e\n", basis[sub[i]] , w[sub[i]]);
      else
        printf("x%d = %e\n", basis[sub[i]] - numcon , w[sub[i]]);
    }
  }

  /* Solve B^T y = w */
  nz     = 1;      /* Only one element in sub is nonzero. */
  sub[0] = 1;      /* Only w[1] is nonzero. */
  w[0]   = 0.0;
  w[1]   = 1.0;

  if (r == MSK_RES_OK)
    r = MSK_solvewithbasis(task, 1, nz, sub, w, &nz);

  if (r == MSK_RES_OK)
  {
    printf("\nSolution to B^T y = w:\n\n");
    /* Print solution and y. */
    for (i = 0; i < nz; ++i)
      printf("y%d = %e\n", sub[i], w[sub[i]]);
  }

  return (r);
}/* main */
```

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:

```

(continues on next page)

(continued from previous page)

```
x0 = 2.000000e+00
xc1 = -4.000000e+00

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 *MSK_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 *MSK_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 $b = (7, 0)$.

```
#include "mosek.h"

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */


static MSKrescodee setup(MSKtask_t task,
                         double    *aval,
                         MSKint32t *asub,
                         MSKint32t *ptrb,
```

(continues on next page)

(continued from previous page)

```
                          MSKint32t *ptre,
                          MSKint32t numvar,
                          MSKint32t *basis)

{
  MSKint32t   i, j;
  MSKrescodee r = MSK_RES_OK;
  MSKstakeye *skx = NULL , *skc = NULL;

  skx = (MSKstakeye *) calloc(numvar, sizeof(MSKstakeye));
  if (skx == NULL && numvar)
    r = MSK_RES_ERR_SPACE;

  skc = (MSKstakeye *) calloc(numvar, sizeof(MSKstakeye));
  if (skc == NULL && numvar)
    r = MSK_RES_ERR_SPACE;

  for (i = 0; i < numvar && r == MSK_RES_OK; ++i)
  {
    skx[i] = MSK_SK_BAS;
    skc[i] = MSK_SK_FIX;
  }

  /* Create a coefficient matrix and right hand
     side with the data from the linear system */
  if (r == MSK_RES_OK)
    r = MSK_appendvars(task, numvar);

  if (r == MSK_RES_OK)
    r = MSK_appendcons(task, numvar);

  for (i = 0; i < numvar && r == MSK_RES_OK; ++i)
    r = MSK_putacol(task, i, ptre[i] - ptrb[i], asub + ptrb[i], aval + ptrb[i]);

  for (i = 0; i < numvar && r == MSK_RES_OK; ++i)
    r = MSK_putconbound(task, i, MSK_BK_FX, 0, 0);

  for (i = 0; i < numvar && r == MSK_RES_OK; ++i)
    r = MSK_putvarbound(task, i, MSK_BK_FR, -MSK_INFINITY, MSK_INFINITY);

  /* Allocate space for the solution and set status to unknown */

  if (r == MSK_RES_OK)
    r = MSK_deletesolution(task, MSK_SOL_BAS);

  /* Setup status keys. That is all that is needed. */
  if (r == MSK_RES_OK)
    r = MSK_putskcslice(task, MSK_SOL_BAS, 0, numvar, skc);

  if (r == MSK_RES_OK)
    r = MSK_putskxslice(task, MSK_SOL_BAS, 0, numvar, skx);

  if (r == MSK_RES_OK)
    r = MSK_initbasissolve(task, basis);

  free(skx);
```

(continues on next page)

(continued from previous page)

```
  free(skc);

  return (r);

}

#define NUMCON 2
#define NUMVAR 2

int main(int argc, const char *argv[])
{
  const MSKint32t numvar = NUMCON,
                  numcon = NUMVAR;    /* we must have numvar == numcon */
  MSKenv_t        env;
  MSKtask_t       task;
  MSKrescodee     r = MSK_RES_OK;
  MSKint32t       i, nz;
  double          aval[] = { -1.0, 1.0, 1.0};
  MSKint32t       asub[] = {1, 0, 1};
  MSKint32t       ptrb[] = {0, 1};
  MSKint32t       ptre[] = {1, 3};
  MSKint32t       bsub[NUMCON];
  double          b[NUMCON];
  MSKint32t       *basis = NULL;

  if (r == MSK_RES_OK)
    r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
    r = MSK_makeemptytask(env, &task);

  if (r == MSK_RES_OK)
    MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

  basis = (MSKint32t *) calloc(numcon, sizeof(MSKint32t));
  if (basis == NULL && numvar)
    r = MSK_RES_ERR_SPACE;

  /* setup: Put A matrix and factor A.
            Call this function only once for a given task. */
  if (r == MSK_RES_OK)
    r = setup(task,
              aval,
              asub,
              ptrb,
              ptre,
              numvar,
              basis
             );

  /* now solve rhs */
  b[0] = 1;
  b[1] = -2;
  bsub[0] = 0;
  bsub[1] = 1;
  nz = 2;
```

(continues on next page)

(continued from previous page)

```
  if (r == MSK_RES_OK)
    r = MSK_solvewithbasis(task, 0, nz, bsub, b, &nz);

  if (r == MSK_RES_OK)
  {
    printf("\nSolution to Bx = b:\n\n");
    /* Print solution and show correspondents
       to original variables in the problem */
    for (i = 0; i < nz; ++i)
    {
      if (basis[bsub[i]] < numcon)
        printf("This should never happen\n");
      else
        printf("x%d = %e\n", basis[bsub[i]] - numcon , b[bsub[i]]);
    }
  }

  b[0] = 7;
  bsub[0] = 0;
  nz = 1;

  if (r == MSK_RES_OK)
    r = MSK_solvewithbasis(task, 0, nz, bsub, b, &nz);

  if (r == MSK_RES_OK)
  {
    printf("\nSolution to Bx = b:\n\n");
    /* Print solution and show correspondents
       to original variables in the problem */
    for (i = 0; i < nz; ++i)
    {
      if (basis[bsub[i]] < numcon)
        printf("This should never happen\n");
      else
        printf("x%d = %e\n", basis[bsub[i]] - numcon , b[bsub[i]]);
    }
  }

  free(basis);
  return r;
}
```

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:
```

(continues on next page)

(continued from previous page)

```
x1 = 7
x0 = 7
```

9.2 Calling BLAS/LAPACK Routines from MOSEK

Sometimes users need to perform linear algebra operations that involve dense matrices and vectors. Also **MOSEK** extensively uses high-performance linear algebra routines from the BLAS and LAPACK packages and some of these routines are included in the package shipped to the users.

The **MOSEK** versions of BLAS/LAPACK routines:

- use **MOSEK** data types and return value conventions,
- preserve the BLAS/LAPACK naming convention.

Therefore the user can leverage on efficient linear algebra routines, with a simplified interface, with no need for additional packages.

List of available routines

Table 9.1: BLAS routines available.

BLAS Name	**MOSEK** function	Math Expression
AXPY	*MSK_axpy*	$y = \alpha x + y$
DOT	*MSK_dot*	$x^T y$
GEMV	*MSK_gemv*	$y = \alpha Ax + \beta y$
GEMM	*MSK_gemm*	$C = \alpha AB + \beta C$
SYRK	*MSK_syrk*	$C = \alpha AA^T + \beta C$

Table 9.2: LAPACK routines available.

LAPACK Name	**MOSEK** function	Description
POTRF	*MSK_potrf*	Cholesky factorization of a semidefinite symmetric matrix
SYEVD	*MSK_syevd*	Eigenvalues and eigenvectors of a symmetric matrix
SYEIG	*MSK_syeig*	Eigenvalues of a symmetric matrix

Source code examples

In Listing 9.2 we provide a simple working example. It has no practical meaning except showing how to organize the input and call the methods.

Listing 9.2: Calling BLAS and LAPACK routines from Optimizer API for C.

```
#include "mosek.h"
void print_matrix(MSKrealt* x, MSKint32t r, MSKint32t c)
{
  MSKint32t i, j;
  for (i = 0; i < r; i++)
  {
    for (j = 0; j < c; j++)
      printf("%f ", x[j * r + i]);

    printf("\n");
  }

}
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char*  argv[])
{

  MSKrescodee r   = MSK_RES_OK;
  MSKenv_t    env = NULL;

  const MSKint32t n = 3, m = 2, k = 3;

  MSKrealt alpha = 2.0, beta = 0.5;
  MSKrealt x[]    = {1.0, 1.0, 1.0};
  MSKrealt y[]    = {1.0, 2.0, 3.0};
  MSKrealt z[]    = {1.0, 1.0};
  MSKrealt A[]    = {1.0, 1.0, 2.0, 2.0, 3.0, 3.0};
  MSKrealt B[]    = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
  MSKrealt C[]    = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
  MSKrealt D[]    = {1.0, 1.0, 1.0, 1.0};
  MSKrealt Q[]    = {1.0, 0.0, 0.0, 2.0};
  MSKrealt v[]    = {0.0, 0.0, 0.0};

  MSKrealt xy;

  /* BLAS routines*/
  r = MSK_makeenv(&env, NULL);
  printf("n=%d m=%d k=%d\n", m, n, k);
  printf("alpha=%f\n", alpha);
  printf("beta=%f\n", beta);

  r = MSK_dot(env, n, x, y, &xy);
  printf("dot results= %f r=%d\n", xy, r);


  print_matrix(x, 1, n);
  print_matrix(y, 1, n);

  r = MSK_axpy(env, n, alpha, x, y);
  puts("axpy results is");
  print_matrix(y, 1, n);


  r = MSK_gemv(env, MSK_TRANSPOSE_NO, m, n, alpha, A, x, beta, z);
  printf("gemv results is (r=%d) \n", r);
  print_matrix(z, 1, m);

  r = MSK_gemm(env, MSK_TRANSPOSE_NO, MSK_TRANSPOSE_NO, m, n, k, alpha, A, B, beta,␣
↪C);
  printf("gemm results is (r=%d) \n", r);
  print_matrix(C, m, n);

  r = MSK_syrk(env, MSK_UPLO_LO, MSK_TRANSPOSE_NO, m, k, 1., A, beta, D);
  printf("syrk results is (r=%d) \n", r);
  print_matrix(D, m, m);

  /* LAPACK routines*/

  r = MSK_potrf(env, MSK_UPLO_LO, m, Q);
```

(continues on next page)

(continued from previous page)

```
  printf("potrf results is (r=%d) \n", r);
  print_matrix(Q, m, m);

  r = MSK_syeig(env, MSK_UPLO_LO, m, Q, v);
  printf("syeig results is (r=%d) \n", r);
  print_matrix(v, 1, m);

  r = MSK_syevd(env, MSK_UPLO_LO, m, Q, v);
  printf("syevd results is (r=%d) \n", r);
  print_matrix(v, 1, m);
  print_matrix(Q, m, m);

  /* Delete the environment and the associated data. */
  MSK_deleteenv(&env);

  return r;
}
```

9.3 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 *MSK_computesparsecholesky* provided by **MOSEK** for computing a Cholesky factorization has a build in permutation aka. reordering heuristic. The following code illustrates the use of *MSK_computesparsecholesky* and *MSK_sparsetriangularsolvedense*.

Listing 9.3: How to use the sparse Cholesky factorization routine available in **MOSEK**.

```
    r = MSK_computesparsecholesky(env,
                                  0,          /* Mosek chooses number of threads */
                                  1,          /* Apply a reordering heuristic */
                                  1.0e-14,    /* Singularity tolerance */
                                  n, anzc, aptrc, asubc, avalc,
                                  &perm, &diag, &lnzc, &lptrc, &lensubnval, &lsubc, &
↪lvalc);

    if (r == MSK_RES_OK)
    {
      MSKint32t i, j;
      MSKrealt  *x;
      printsparse(n, perm, diag, lnzc, lptrc, lensubnval, lsubc, lvalc);

      x = MSK_callocenv(env, n, sizeof(MSKrealt));
      if (x)
      {
        /* Permuted b is stored as x. */
        for (i = 0; i < n; ++i) x[i] = b[perm[i]];

        /* Compute inv(L)*x. */
        r = MSK_sparsetriangularsolvedense(env, MSK_TRANSPOSE_NO, n,
                                           lnzc, lptrc, lensubnval, lsubc, lvalc, x);

        if (r == MSK_RES_OK) {
          /* Compute inv(L^T)*x. */
          r = MSK_sparsetriangularsolvedense(env, MSK_TRANSPOSE_YES, n,
                                             lnzc, lptrc, lensubnval, lsubc, lvalc,␣
↪x);
          printf("\nSolution A x = b, x = [ ");
          for (i = 0; i < n; i++)
            for (j = 0; j < n; j++) if (perm[j] == i) printf("%.2f ", x[j]);
          printf("]\n");
        }

        MSK_freeenv(env, x);
      }
      else
        printf("Out of space while creating x.\n");
    }
    else
      printf("Cholesky computation failed: %d\n", (int) r);
```

We can set up the data to recreate the matrix A from (9.6):

```
    const MSKint32t n       = 4;       // Data from the example in the text
    //Observe that anzc, aptrc, asubc and avalc only specify the lower triangular␣
↪part.
    const MSKint32t anzc[]  = {4, 1, 1, 1},
                    asubc[] = {0, 1, 2, 3, 1, 2, 3};
    const MSKint64t aptrc[] = {0, 4, 5, 6};
    const MSKrealt  avalc[] = {4.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0},
                    b[]     = {13.0, 3.0, 4.0, 5.0};
    MSKint32t       *perm = NULL, *lnzc = NULL, *lsubc = NULL;
    MSKint64t       lensubnval, *lptrc = NULL;
    MSKrealt        *diag = NULL, *lvalc = NULL;
```

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

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}^T \tag{9.7}$$

```
    const MSKint32t n       = 3;
    const MSKint32t anzc[]  = {3, 2, 1},
                    asubc[] = {0, 1, 2, 1, 2, 2};
    const MSKint64t aptrc[] = {0, 3, 5};
    const MSKrealt  avalc[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
    MSKint32t       *perm = NULL, *lnzc = NULL, *lsubc = NULL;
    MSKint64t       lensubnval, *lptrc = NULL;
    MSKrealt        *diag = NULL, *lvalc = NULL;
```

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 *MSK_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 C, not strictly necessary for basic use of **MOSEK**.

10.1 Memory management and garbage collection

Users who experience memory leaks, especially:

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

should make sure that the memory used by the task is released when the task is no longer needed. This is done with the method *MSK_deletetask*. The same applies to the environment when it is no longer needed.

```
MSK_deletetask(&task);
MSK_deleteenv(&env);
```

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 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 (*MSK_putaij*) define them all at once (*MSK_putaijlist*) or in convenient large chunks (*MSK_putacollist* etc.)

Use one environment only

If possible share the environment between several tasks. For most applications you need to create only a single environment.

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 *MSK_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:

- *MSK_putmaxnumvar*. Estimate for the number of variables.
- *MSK_putmaxnumcon*. Estimate for the number of constraints.
- *MSK_putmaxnumbarvar*. Estimate for the number of semidefinite matrix variables.
- *MSK_putmaxnumanz*. Estimate for the number of non-zeros in A.
- *MSK_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.

10.5 The license system

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

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

- a license token is checked out when *`MSK_optimize`* is first called, and
- it is returned when the **MOSEK** environment is deleted.

Calling *`MSK_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 *`MSK_putlicensewait`*.
- Additional license checkouts and checkins can be performed with the functions *`MSK_checkinlicense`* and *`MSK_checkoutlicense`*.
- 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 *`MSK_licensecleanup`* as the last function call to **MOSEK**.

10.6 Deployment

When redistributing a C application using the **MOSEK** Optimizer API for C 10.0.46, the following shared libraries from the **MOSEK** `bin` folder are required:

- Linux : `libmosek64`, `libtbb`,
- Windows : `mosek64`, `tbb`, `svml_dispmd`,
- OSX : `libmosek64`, `libtbb`.

10.7 Strings

MOSEK supports Unicode strings. All arguments of type `char *` are allowed to be UTF8 encoded strings (http://en.wikipedia.org/wiki/UTF-8). Please note that

- an ASCII string is always a valid UTF8 string, and
- an UTF8 string is stored in a `char` array.

It is possible to convert between `wchar_t` strings and UTF8 strings using the functions *MSK_wchartoutf8* and *MSK_utf8towchar*. A working example is provided in the example file `unicode.c`.

Chapter 11

Case Studies

In this section we present some case studies in which the Optimizer API for C is used to solve real-life applications. These examples involve some more advanced modeling skills and possibly some input data. The user is strongly recommended to first read the basic tutorials of Sec. 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 C.

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

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

The variables x denote the investment i.e. x_j is the amount invested in asset j and x_j^0 is the initial holding of asset j. Finally, w is the initial amount of cash available.

A popular choice is $x^0 = 0$ and $w = 1$ because then x_j may be interpreted as the relative amount of the total portfolio that is invested in asset j.

Since e is the vector of all ones then

$$e^T x = \sum_{j=1}^{n} x_j$$

is the total investment. Clearly, the total amount invested must be equal to the initial wealth, which is

$$w + e^T x^0.$$

This leads to the first constraint

$$e^T x = w + e^T x^0.$$

The second constraint

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

ensures that the variance, is bounded by the parameter γ^2. Therefore, γ specifies an upper bound of the standard deviation (risk) the investor is willing to undertake. Finally, the constraint

$$x_j \geq 0$$

excludes the possibility of short-selling. This constraint can of course be excluded if short-selling is allowed.

The covariance matrix Σ is positive semidefinite by definition and therefore there exist a matrix $G \in \mathbb{R}^{n \times k}$ 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

$$\begin{array}{rcl} x^T \Sigma x & = & x^T GG^T x \\ & = & \left\| G^T x \right\|^2. \end{array}$$

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 C. Subsequently we will use the example data

$$\mu = \left[\begin{array}{c} 0.0720, 0.1552, 0.1754, 0.0898, 0.4290, 0.3929, 0.3217, 0.1838 \end{array} \right]^T$$

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

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

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).

```
#include <math.h>
#include <stdio.h>
#include "mosek.h"

#define MOSEKCALL(_r,_call)  if ( (_r)==MSK_RES_OK ) (_r) = (_call)

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char **argv)
{
  char             buf[128];

  double           expret  = 0.0,
                   stddev  = 0.0,
                   xj;

  const MSKint32t n        = 8;
  const MSKrealt  gamma    = 36.0;
  const MSKrealt  mu[]     = {0.07197349, 0.15518171, 0.17535435, 0.0898094 , 0.
↪42895777, 0.39291844, 0.32170722, 0.18378628};
  // GT must have size n rows
  const MSKrealt  GT[][8] = {
    {0.30758, 0.12146, 0.11341, 0.11327, 0.17625, 0.11973, 0.10435, 0.10638},
    {0.0,     0.25042, 0.09946, 0.09164, 0.06692, 0.08706, 0.09173, 0.08506},
    {0.0,     0.0,     0.19914, 0.05867, 0.06453, 0.07367, 0.06468, 0.01914},
    {0.0,     0.0,     0.0,     0.20876, 0.04933, 0.03651, 0.09381, 0.07742},
    {0.0,     0.0,     0.0,     0.0,     0.36096, 0.12574, 0.10157, 0.0571 },
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.21552, 0.05663, 0.06187},
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.22514, 0.03327},
```

(continues on next page)

(continued from previous page)

```
  {0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.2202 }
};
const MSKint32t k       = sizeof(GT) / (n * sizeof(MSKrealt));
const MSKrealt  x0[]    = {8.0, 5.0, 3.0, 5.0, 2.0, 9.0, 3.0, 6.0};
const MSKrealt  w       = 59;
MSKrealt        totalBudget;

MSKenv_t        env;
MSKint32t       i, j, *sub;
MSKrescodee     res = MSK_RES_OK, trmcode;
MSKtask_t       task;

//Offset of variables into the API variable.
MSKint32t numvar = n;
MSKint32t voff_x = 0;

// Constraints offsets
MSKint32t numcon = 1;
MSKint32t coff_bud = 0;

/* Initial setup. */
env  = NULL;
task = NULL;
MOSEKCALL(res, MSK_makeenv(&env, NULL));
MOSEKCALL(res, MSK_maketask(env, 0, 0, &task));
MOSEKCALL(res, MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr));

// Holding variable x of length n
// No other auxiliary variables are needed in this formulation
MOSEKCALL(res, MSK_appendvars(task, numvar));
// Setting up variable x
for (j = 0; j < n; ++j)
{
  /* Optionally we can give the variables names */
  sprintf(buf, "x[%d]", 1 + j);
  MOSEKCALL(res, MSK_putvarname(task, voff_x + j, buf));
  /* No short-selling - x^l = 0, x^u = inf */
  MOSEKCALL(res, MSK_putvarbound(task, voff_x + j, MSK_BK_LO, 0.0, MSK_INFINITY));
}

// One linear constraint: total budget
MOSEKCALL(res, MSK_appendcons(task, 1));
sprintf(buf, "%s", "budget");
MOSEKCALL(res, MSK_putconname(task, coff_bud, buf));
for (j = 0; j < n; ++j)
{
  /* Coefficients in the first row of A */
  MOSEKCALL(res, MSK_putaij(task, coff_bud, voff_x + j, 1.0));
}
totalBudget = w;
for (i = 0; i < n; ++i)
{
  totalBudget += x0[i];
}
MOSEKCALL(res, MSK_putconbound(task, coff_bud, MSK_BK_FX, totalBudget,␣
↪totalBudget));
```

(continues on next page)

(continued from previous page)

```
  // Input (gamma, GTx) in the AFE (affine expression) storage
  // We need k+1 rows
  MOSEKCALL(res, MSK_appendafes(task, k + 1));
  // The first affine expression = gamma
  MOSEKCALL(res, MSK_putafeg(task, 0, 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
  MSKint32t* vslice_x = (MSKint32t*) malloc(n * sizeof(MSKint32t));
  for (i = 0; i < n; ++i)
  {
    vslice_x[i] = voff_x + i;
  }
  for (i = 0; i < k; ++i)
  {
    MOSEKCALL(res, MSK_putafefrow(task, i + 1, n, vslice_x, GT[i]));
  }
  free(vslice_x);

  // Input the affine conic constraint (gamma, GT*x) \in QCone
  // Add the quadratic domain of dimension k+1
  MSKint64t qdom;
  MOSEKCALL(res, MSK_appendquadraticconedomain(task, k + 1, &qdom));
  // Add the constraint
  MOSEKCALL(res, MSK_appendaccseq(task, qdom, k + 1, 0, NULL));
  sprintf(buf, "%s", "risk");
  MOSEKCALL(res, MSK_putaccname(task, 0, buf));

  // Objective: maximize expected return mu^T x
  for (j = 0; j < n; ++j)
  {
    MOSEKCALL(res, MSK_putcj(task, voff_x + j, mu[j]));
  }
  MOSEKCALL(res, MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE));

#if 0
  /* No log output */
  MOSEKCALL(res, MSK_putintparam(task, MSK_IPAR_LOG, 0));
#endif

#if 0
  /* Dump the problem to a human readable PTF file. */
  MOSEKCALL(res, MSK_writedata(task, "dump.ptf"));
#endif

  MOSEKCALL(res, MSK_optimizetrm(task, &trmcode));

# if 1
  /* Display the solution summary for quick inspection of results. */
  MSK_solutionsummary(task, MSK_STREAM_MSG);
# endif

  if ( res == MSK_RES_OK )
  {
    expret = 0.0;
```

(continues on next page)

(continued from previous page)

```
    stddev = 0.0;

    /* Read the x variables one by one and compute expected return. */
    /* Can also be obtained as value of the objective. */
    for (j = 0; j < n; ++j)
    {
      MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITR, voff_x + j, voff_x + j + 1, &
↪xj));
      expret += mu[j] * xj;
    }

    /* Read the value of s. This should be gamma. */
    printf("\nExpected return %e for gamma %e\n", expret, gamma);
  }

  if ( task != NULL )
    MSK_deletetask(&task);

  if ( env != NULL )
    MSK_deleteenv(&env);

  return ( 0 );
}
```

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
  MSKint64t qdom;
  MOSEKCALL(res, MSK_appendquadraticconedomain(task, k + 1, &qdom));
  // Add the constraint
  MOSEKCALL(res, MSK_appendaccseq(task, qdom, k + 1, 0, NULL));
  ```

  we use *MSK_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:

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

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).

```
#include <math.h>
#include <stdio.h>

#include "mosek.h"

#define MOSEKCALL(_r,_call)  if ( (_r)==MSK_RES_OK ) (_r) = (_call)
#define LOGLEVEL             0

static void MSKAPI printstr(void       *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */


int main(int argc, const char **argv)
{
  char buf[128];
  const MSKint32t n       = 8;
  const MSKrealt  mu[]    = {0.07197349, 0.15518171, 0.17535435, 0.0898094 , 0.
↪42895777, 0.39291844, 0.32170722, 0.18378628};
  // GT must have size n rows
  const MSKrealt  GT[][8] = {
    {0.30758, 0.12146, 0.11341, 0.11327, 0.17625, 0.11973, 0.10435, 0.10638},
    {0.0,     0.25042, 0.09946, 0.09164, 0.06692, 0.08706, 0.09173, 0.08506},
    {0.0,     0.0,     0.19914, 0.05867, 0.06453, 0.07367, 0.06468, 0.01914},
    {0.0,     0.0,     0.0,     0.20876, 0.04933, 0.03651, 0.09381, 0.07742},
```

(continues on next page)

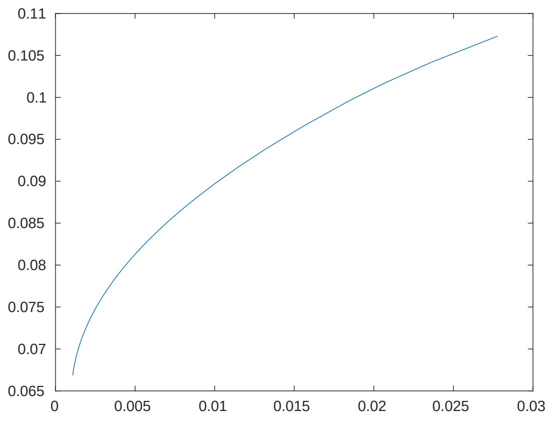

Fig. 11.1: The efficient frontier for the sample data.

(continued from previous page)

```
    {0.0,     0.0,     0.0,     0.0,     0.36096, 0.12574, 0.10157, 0.0571 },
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.21552, 0.05663, 0.06187},
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.22514, 0.03327},
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.2202 }
  };
  const MSKint32t k       = sizeof(GT) / (n * sizeof(MSKrealt));
  const MSKrealt  x0[]    = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
  const MSKrealt  w       = 1.0;
  const MSKrealt alphas[] = {0.0, 0.01, 0.1, 0.25, 0.30, 0.35, 0.4, 0.45, 0.5, 0.75,␣
↪1.0, 1.5, 2.0, 3.0, 10.0};
  const MSKint32t numalpha = 15;
  MSKrealt totalBudget;

  MSKenv_t        env;
  MSKint32t       i, j;
  MSKrescodee     res = MSK_RES_OK, lres;
  MSKtask_t       task;
  MSKrealt        xj;
  MSKsolstae      solsta;

  //Offset of variables into the API variable.
  MSKint32t numvar = n + 1;
  MSKint32t voff_x = 0;
  MSKint32t voff_s = n;

  // Constraints offsets
  MSKint32t numcon = 1;
  MSKint32t coff_bud = 0;

  /* Initial setup. */
  env  = NULL;
  task = NULL;

  /* Replace "" with NULL in production. */
  MOSEKCALL(res, MSK_makeenv(&env, ""));

  MOSEKCALL(res, MSK_maketask(env, 0, 0, &task));
  MOSEKCALL(res, MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr));

  // Holding variable x of length n
  MOSEKCALL(res, MSK_appendvars(task, numvar));
  // Setting up variable x
  for (j = 0; j < n; ++j)
  {
    /* Optionally we can give the variables names */
    sprintf(buf, "x[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, voff_x + j, buf));
    /* No short-selling - x^l = 0, x^u = inf */
    MOSEKCALL(res, MSK_putvarbound(task, voff_x + j, MSK_BK_LO, 0.0, MSK_INFINITY));
  }
  sprintf(buf, "s");
  MOSEKCALL(res, MSK_putvarname(task, voff_s, buf));
  MOSEKCALL(res, MSK_putvarbound(task, voff_s, MSK_BK_FR, -MSK_INFINITY, MSK_
↪INFINITY));

  // One linear constraint: total budget
```

(continues on next page)

(continued from previous page)

```
  MOSEKCALL(res, MSK_appendcons(task, 1));
  sprintf(buf, "%s", "budget");
  MOSEKCALL(res, MSK_putconname(task, coff_bud, buf));
  for (j = 0; j < n; ++j)
  {
    /* Coefficients in the first row of A */
    MOSEKCALL(res, MSK_putaij(task, coff_bud, voff_x + j, 1.0));
  }
  totalBudget = w;
  for (i = 0; i < n; ++i)
  {
    totalBudget += x0[i];
  }
  MOSEKCALL(res, MSK_putconbound(task, coff_bud, 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
  MOSEKCALL(res, MSK_appendafes(task, k + 2));
  // The first affine expression is variable s (last variable, index n)
  MOSEKCALL(res, MSK_putafefentry(task, 0, n, 1.0));
  // The second affine expression is constant 0.5
  MOSEKCALL(res, MSK_putafeg(task, 1, 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
  MSKint32t* vslice_x = (MSKint32t*) malloc(n * sizeof(MSKint32t));
  for (i = 0; i < n; ++i)
  {
    vslice_x[i] = voff_x + i;
  }
  for (i = 0; i < k; ++i)
  {
    MOSEKCALL(res, MSK_putafefrow(task, i + 2, n, vslice_x, GT[i]));
  }
  free(vslice_x);

  // Input the affine conic constraint (gamma, GT*x) \in QCone
  // Add the quadratic domain of dimension k+1
  MSKint64t rqdom;
  MOSEKCALL(res, MSK_appendrquadraticconedomain(task, k + 2, &rqdom));
  // Add the constraint
  MOSEKCALL(res, MSK_appendaccseq(task, rqdom, k + 2, 0, NULL));
  sprintf(buf, "%s", "risk");
  MOSEKCALL(res, MSK_putaccname(task, 0, buf));

  // Objective: maximize expected return mu^T x
  for (j = 0; j < n; ++j)
  {
    MOSEKCALL(res, MSK_putcj(task, voff_x + j, mu[j]));
  }
  MOSEKCALL(res, MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE));
```

(continues on next page)

(continued from previous page)

```
  /* Set the log level */
  MOSEKCALL(res, MSK_putintparam(task, MSK_IPAR_LOG, LOGLEVEL));

  printf("%-12s  %-12s  %-12s\n", "alpha", "exp ret", "std. dev");

  for (i = 0; i < numalpha && res==MSK_RES_OK; ++i)
  {
    const double alpha = alphas[i];
    MSKrescodee  trmcode;

    /* Sets the objective function coefficient for s. */
    MOSEKCALL(res, MSK_putcj(task, voff_s + 0, -alpha));

    MOSEKCALL(res, MSK_optimizetrm(task, &trmcode));

    MOSEKCALL(res, MSK_getsolsta(task, MSK_SOL_ITR, &solsta));

    if (solsta == MSK_SOL_STA_OPTIMAL)
    {
      double expret = 0.0,
             stddev;

      for (j = 0; j < n; ++j)
      {
        MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITR, voff_x + j, voff_x + j + 1, &
↪xj));
        expret += mu[j] * xj;
      }

      MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITR, voff_s + 0, voff_s + 1, &
↪stddev));

      printf("%-12.3e  %-12.3e  %-12.3e\n", alpha, expret, sqrt(stddev));
    }
    else
    {
      printf("An error occurred when solving for alpha=%e\n", alpha);
    }
  }
  lres = MSK_deletetask(&task);
  res  = res==MSK_RES_OK ? lres : res;

  lres = MSK_deleteenv(&env);
  res  = res==MSK_RES_OK ? lres : res;

  return ( res );
}
```

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

$$G = \left[\begin{array}{cc} D^{1/2} & V \end{array} \right]$$

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.

```
// NOTE: Here we specify matrices as vectors (row major order) to avoid having
// to initialize them as double(*)[] type, which is incompatible with double**.

// Factor exposure matrix
MSKrealt vecB[] =
{
  0.4256, 0.1869,
  0.2413, 0.3877,
  0.2235, 0.3697,
  0.1503, 0.4612,
  1.5325, -0.2633,
```

(continues on next page)

(continued from previous page)

```
    1.2741, -0.2613,
    0.6939, 0.2372,
    0.5425, 0.2116
  };
  matrix* B = vec_to_mat_r(vecB, n, 2);
  // Factor covariance matrix
  MSKrealt vecS_F[] =
  {
    0.0620, 0.0577,
    0.0577, 0.0908
  };
  matrix* S_F = vec_to_mat_r(vecS_F, 2, 2);

  // Specific risk components
  MSKrealt theta[] = {0.0720, 0.0508, 0.0377, 0.0394, 0.0663, 0.0224, 0.0417, 0.0459};
```

Then the matrix G_{factor} is obtained as:

```
  matrix* P = cholesky(&env, S_F);
  matrix* G_factor = matrix_mul(&env, B, P);
```

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 $G^T_{\text{factor}} x$ 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
  MOSEKCALL(res, MSK_appendafes(task, k + n + 1));
  // 1. The first affine expression = gamma, will be specified later
  // 2. The next k expressions comprise G_factor_T*x, we add them column by column␣
↪since
  //    G_factor is stored row-wise and we transpose on the fly
  MSKint64t* afeidxs = (MSKint64t*) malloc(k * sizeof(MSKint64t));
  for (i = 0; i < k; ++i)
  {
    afeidxs[i] = i + 1;
  }
  for (i = 0; i < n; ++i)
  {
    MOSEKCALL(res, MSK_putafefcol(task, i, k, afeidxs, G_factor->m[i])); //i-th row␣
↪of G_factor goes in i-th column of F
  }
  free(afeidxs);
  // 3. The remaining n rows contain sqrt(theta) on the diagonal
  for (i = 0; i < n; ++i)
  {
    MOSEKCALL(res, MSK_putafefentry(task, k + 1 + i, voff_x + i, sqrt(theta[i])));
  }
```

The full code is demonstrated below:

Listing 11.4: Implementation of portfolio optimization in the factor model.

```
int main(int argc, const char **argv)
{
  char              buf[128];
```

(continues on next page)

(continued from previous page)

```
MSKrealt        expret  = 0.0,
                xj;

MSKrealt        rtemp;
MSKenv_t        env;
MSKint32t       i, j, *sub;
MSKrescodee     res = MSK_RES_OK, trmcode;
MSKtask_t       task;

MSKint32t  n      = 8;
MSKrealt   w      = 1.0;
MSKrealt   mu[]   = {0.07197, 0.15518, 0.17535, 0.08981, 0.42896, 0.39292, 0.32171,␣
↪0.18379};
MSKrealt   x0[]   = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};

/* Initial setup. */
env  = NULL;
task = NULL;
MOSEKCALL(res, MSK_makeenv(&env, NULL));
MOSEKCALL(res, MSK_maketask(env, 0, 0, &task));
MOSEKCALL(res, MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr));

// NOTE: Here we specify matrices as vectors (row major order) to avoid having
// to initialize them as double(*)[] type, which is incompatible with double**.

// Factor exposure matrix
MSKrealt vecB[] =
{
  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
};
matrix* B = vec_to_mat_r(vecB, n, 2);
// Factor covariance matrix
MSKrealt vecS_F[] =
{
  0.0620, 0.0577,
  0.0577, 0.0908
};
matrix* S_F = vec_to_mat_r(vecS_F, 2, 2);

// Specific risk components
MSKrealt theta[] = {0.0720, 0.0508, 0.0377, 0.0394, 0.0663, 0.0224, 0.0417, 0.0459};

matrix* P = cholesky(&env, S_F);
matrix* G_factor = matrix_mul(&env, B, P);
matrix_free(B);
matrix_free(S_F);
matrix_free(P);
```

(continues on next page)

(continued from previous page)

```
MSKint32t k = G_factor->nc;
MSKrealt gammas[] = {0.24, 0.28, 0.32, 0.36, 0.4, 0.44, 0.48};
int num_gammas = 7;
MSKrealt   totalBudget;

//Offset of variables into the API variable.
MSKint32t numvar = n;
MSKint32t voff_x = 0;

// Constraint offset
MSKint32t coff_bud = 0;

// Holding variable x of length n
// No other auxiliary variables are needed in this formulation
MOSEKCALL(res, MSK_appendvars(task, numvar));
// Setting up variable x
for (j = 0; j < n; ++j)
{
  /* Optionally we can give the variables names */
  sprintf(buf, "x[%d]", 1 + j);
  MOSEKCALL(res, MSK_putvarname(task, voff_x + j, buf));
  /* No short-selling - x^l = 0, x^u = inf */
  MOSEKCALL(res, MSK_putvarbound(task, voff_x + j, MSK_BK_LO, 0.0, MSK_INFINITY));
}

// One linear constraint: total budget
MOSEKCALL(res, MSK_appendcons(task, 1));
sprintf(buf, "%s", "budget");
MOSEKCALL(res, MSK_putconname(task, coff_bud, buf));
for (j = 0; j < n; ++j)
{
  /* Coefficients in the first row of A */
  MOSEKCALL(res, MSK_putaij(task, coff_bud, voff_x + j, 1.0));
}
totalBudget = w;
for (i = 0; i < n; ++i)
{
  totalBudget += x0[i];
}
MOSEKCALL(res, MSK_putconbound(task, coff_bud, MSK_BK_FX, totalBudget,␣
↪totalBudget));

// 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
MOSEKCALL(res, MSK_appendafes(task, k + n + 1));
// 1. The first affine expression = gamma, will be specified later
// 2. The next k expressions comprise G_factor_T*x, we add them column by column␣
↪since
//    G_factor is stored row-wise and we transpose on the fly
MSKint64t* afeidxs = (MSKint64t*) malloc(k * sizeof(MSKint64t));
for (i = 0; i < k; ++i)
{
  afeidxs[i] = i + 1;
}
for (i = 0; i < n; ++i)
```

(continues on next page)

(continued from previous page)

```
  {
    MOSEKCALL(res, MSK_putafefcol(task, i, k, afeidxs, G_factor->m[i])); //i-th row␣
↪of G_factor goes in i-th column of F
  }
  free(afeidxs);
  // 3. The remaining n rows contain sqrt(theta) on the diagonal
  for (i = 0; i < n; ++i)
  {
    MOSEKCALL(res, MSK_putafefentry(task, k + 1 + i, voff_x + i, sqrt(theta[i])));
  }

  // Input the affine conic constraint (gamma, G_factor_T x, diag(sqrt(theta))*x) \in␣
↪QCone
  // Add the quadratic domain of dimension k+n+1
  MSKint64t qdom;
  MOSEKCALL(res, MSK_appendquadraticconedomain(task, k + n + 1, &qdom));
  // Add the constraint
  MOSEKCALL(res, MSK_appendaccseq(task, qdom, k + n + 1, 0, NULL));
  sprintf(buf, "%s", "risk");
  MOSEKCALL(res, MSK_putaccname(task, 0, buf));

  // Objective: maximize expected return mu^T x
  for (j = 0; j < n; ++j)
  {
    MOSEKCALL(res, MSK_putcj(task, voff_x + j, mu[j]));
  }
  MOSEKCALL(res, MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE));

#if 0
  /* No log output */
  MOSEKCALL(res, MSK_putintparam(task, MSK_IPAR_LOG, 0));
#endif

  for (i = 0; i < num_gammas; i++)
  {
    MSKrealt gamma = gammas[i];

    // Specify gamma in ACC
    MOSEKCALL(res, MSK_putafeg(task, 0, gamma));

#if 0
    /* Dump the problem to a human readable PTF file. */
    MOSEKCALL(res, MSK_writedata(task, "dump.ptf"));
#endif

    MOSEKCALL(res, MSK_optimizetrm(task, &trmcode));

# if 1
    /* Display the solution summary for quick inspection of results. */
    MSK_solutionsummary(task, MSK_STREAM_MSG);
# endif

    if ( res == MSK_RES_OK )
    {
      expret = 0.0;
```

(continues on next page)

(continued from previous page)

```
      /* Read the x variables one by one and compute expected return. */
      /* Can also be obtained as value of the objective. */
      for (j = 0; j < n; ++j)
      {
        MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITR, voff_x + j, voff_x + j + 1, &
↪xj));
        expret += mu[j] * xj;
      }

      printf("\nExpected return %e for gamma %e\n", expret, gamma);
    }
  }

  if ( task != NULL )
    MSK_deletetask(&task);

  if ( env != NULL )
    MSK_deleteenv(&env);

  matrix_free(G_factor);

  return ( 0 );
}
```

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).

```
#include <math.h>
#include <stdio.h>

#include "mosek.h"

#define MOSEKCALL(_r,_call)  if ( (_r)==MSK_RES_OK ) (_r) = (_call)

```

(continues on next page)

(continued from previous page)

```
static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char **argv)
{
  char              buf[128];
  const MSKint32t n       = 8;
  const MSKrealt  mu[]    = {0.07197, 0.15518, 0.17535, 0.08981, 0.42896, 0.39292, 0.
↪32171, 0.18379};
  // GT must have size n rows
  const MSKrealt  GT[][8] = {
    {0.30758, 0.12146, 0.11341, 0.11327, 0.17625, 0.11973, 0.10435, 0.10638},
    {0.0,     0.25042, 0.09946, 0.09164, 0.06692, 0.08706, 0.09173, 0.08506},
    {0.0,     0.0,     0.19914, 0.05867, 0.06453, 0.07367, 0.06468, 0.01914},
    {0.0,     0.0,     0.0,     0.20876, 0.04933, 0.03651, 0.09381, 0.07742},
    {0.0,     0.0,     0.0,     0.0,     0.36096, 0.12574, 0.10157, 0.0571 },
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.21552, 0.05663, 0.06187},
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.22514, 0.03327},
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.2202 }
  };
  const MSKint32t k       = sizeof(GT) / (n * sizeof(MSKrealt));
  const MSKrealt  x0[]    = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
  const MSKrealt  w       = 1.0;
  const MSKrealt  gamma   = 0.36;
  MSKrealt totalBudget;
  MSKint32t       i, j;

  MSKrealt* m = (MSKrealt*) malloc(n * sizeof(MSKrealt));
  for (i = 0; i < n; ++i)
  {
    m[i] = 0.01;
  }

  // Offset of variables into the API variable.
  MSKint32t numvar = 3 * n;
  MSKint32t voff_x = 0;
  MSKint32t voff_c = n;
  MSKint32t voff_z = 2 * n;

  // Offset of constraints.
  MSKint32t numcon = 2 * n + 1;
  MSKint32t coff_bud = 0;
  MSKint32t coff_abs1 = 1;
  MSKint32t coff_abs2 = 1 + n;

  double          expret,
                  stddev,
                  xj;
  MSKenv_t        env;
  MSKrescodee     res = MSK_RES_OK, trmcode;
  MSKtask_t       task;

  /* Initial setup. */
```

(continues on next page)

(continued from previous page)

```
  env  = NULL;
  task = NULL;
  MOSEKCALL(res, MSK_makeenv(&env, NULL));
  MOSEKCALL(res, MSK_maketask(env, 0, 0, &task));
  MOSEKCALL(res, MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr));

  // Variables (vector of x, c, z)
  MOSEKCALL(res, MSK_appendvars(task, numvar));
  for (j = 0; j < n; ++j)
  {
    /* Optionally we can give the variables names */
    sprintf(buf, "x[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, voff_x + j, buf));
    sprintf(buf, "c[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, voff_c + j, buf));
    sprintf(buf, "z[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, voff_z + j, buf));
    /* Apply variable bounds (x >= 0, c and z free) */
    MOSEKCALL(res, MSK_putvarbound(task, voff_x + j, MSK_BK_LO, 0.0, MSK_INFINITY));
    MOSEKCALL(res, MSK_putvarbound(task, voff_c + j, MSK_BK_FR, -MSK_INFINITY, MSK_
↪INFINITY));
    MOSEKCALL(res, MSK_putvarbound(task, voff_z + j, MSK_BK_FR, -MSK_INFINITY, MSK_
↪INFINITY));
  }

  // Linear constraints
  // - Total budget
  MOSEKCALL(res, MSK_appendcons(task, 1));
  sprintf(buf, "%s", "budget");
  MOSEKCALL(res, MSK_putconname(task, coff_bud, buf));
  for (j = 0; j < n; ++j)
  {
    /* Coefficients in the first row of A */
    MOSEKCALL(res, MSK_putaij(task, coff_bud, voff_x + j, 1.0));
    MOSEKCALL(res, MSK_putaij(task, coff_bud, voff_c + j, m[j]));
  }
  totalBudget = w;
  for (i = 0; i < n; ++i)
  {
    totalBudget += x0[i];
  }
  MOSEKCALL(res, MSK_putconbound(task, coff_bud, MSK_BK_FX, totalBudget,␣
↪totalBudget));

  // - Absolute value
  MOSEKCALL(res, MSK_appendcons(task, 2 * n));
  for (i = 0; i < n; ++i)
  {
    sprintf(buf, "zabs1[%d]", 1 + i);
    MOSEKCALL(res, MSK_putconname(task, coff_abs1 + i, buf));
    MOSEKCALL(res, MSK_putaij(task, coff_abs1 + i, voff_x + i, -1.0));
    MOSEKCALL(res, MSK_putaij(task, coff_abs1 + i, voff_z + i, 1.0));
    MOSEKCALL(res, MSK_putconbound(task, coff_abs1 + i, MSK_BK_LO, -x0[i], MSK_
↪INFINITY));
    sprintf(buf, "zabs2[%d]", 1 + i);
    MOSEKCALL(res, MSK_putconname(task, coff_abs2 + i, buf));
```

(continues on next page)

(continued from previous page)

```
    MOSEKCALL(res, MSK_putaij(task, coff_abs2 + i, voff_x + i, 1.0));
    MOSEKCALL(res, MSK_putaij(task, coff_abs2 + i, voff_z + i, 1.0));
    MOSEKCALL(res, MSK_putconbound(task, coff_abs2 + i, MSK_BK_LO, x0[i], MSK_
↪INFINITY));
  }

  // ACCs
  MSKint32t aoff_q = 0;
  MSKint32t aoff_pow = k + 1;
  // - (gamma, GTx) in Q(k+1)
  // The part of F and g for variable x:
  //     [0,  0, 0]      [gamma]
  // F = [GT, 0, 0], g = [0    ]
  MOSEKCALL(res, MSK_appendafes(task, k + 1));
  MOSEKCALL(res, MSK_putafeg(task, aoff_q, gamma));
  MSKint32t* vslice_x = (MSKint32t*) malloc(n * sizeof(MSKint32t));
  for (i = 0; i < n; ++i)
  {
    vslice_x[i] = voff_x + i;
  }
  for (i = 0; i < k; ++i)
  {
    MOSEKCALL(res, MSK_putafefrow(task, aoff_q + i + 1, n, vslice_x, GT[i]));
  }
  free(vslice_x);
  MSKint64t qdom;
  MOSEKCALL(res, MSK_appendquadraticconedomain(task, k + 1, &qdom));
  MOSEKCALL(res, MSK_appendaccseq(task, qdom, k + 1, aoff_q, NULL));
  sprintf(buf, "%s", "risk");
  MOSEKCALL(res, MSK_putaccname(task, aoff_q, buf));

  // - (c_j, 1, z_j) in P3(2/3, 1/3)
  // The part of F and g for variables [c, z]:
  //     [0, I, 0]      [0]
  // F = [0, 0, I], g = [0]
  //     [0, 0, 0]      [1]
  MOSEKCALL(res, MSK_appendafes(task, 2 * n + 1));
  for (i = 0; i < n; ++i)
  {
    MOSEKCALL(res, MSK_putafefentry(task, aoff_pow + i, voff_c + i, 1.0));
    MOSEKCALL(res, MSK_putafefentry(task, aoff_pow + n + i, voff_z + i, 1.0));
  }
  MOSEKCALL(res, MSK_putafeg(task, aoff_pow + 2 * n, 1.0));
  // We use one row from F and g for both c_j and z_j, and the last row of F and g␣
↪for the constant 1.
  // NOTE: Here we reuse the last AFE and the power cone n times, but we store them␣
↪only once.
  MSKrealt exponents[] = {2, 1};
  MSKint64t powdom;
  MOSEKCALL(res, MSK_appendprimalpowerconedomain(task, 3, 2, exponents, &powdom));
  MSKint64t* flat_afe_list = (MSKint64t*) malloc(3 * n * sizeof(MSKint64t));
  MSKint64t* dom_list = (MSKint64t*) malloc(n * sizeof(MSKint64t));
  for (i = 0; i < n; ++i)
  {
    flat_afe_list[3 * i + 0] = aoff_pow + i;
    flat_afe_list[3 * i + 1] = aoff_pow + 2 * n;
```

(continues on next page)

(continued from previous page)

```
    flat_afe_list[3 * i + 2] = aoff_pow + n + i;
    dom_list[i] = powdom;
  }
  MOSEKCALL(res, MSK_appendaccs(task, n, dom_list, 3 * n, flat_afe_list, NULL));
  free(flat_afe_list);
  free(dom_list);
  for (i = 0; i < n; ++i)
  {
    sprintf(buf, "market_impact[%d]", i);
    MOSEKCALL(res, MSK_putaccname(task, i + 1, buf));
  }

  // Objective: maximize expected return mu^T x
  for (j = 0; j < n; ++j)
  {
    MOSEKCALL(res, MSK_putcj(task, voff_x + j, mu[j]));
  }
  MOSEKCALL(res, MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE));

  #if 0
    /* no log output. */
    MOSEKCALL(res, MSK_putintparam(task, MSK_IPAR_LOG, 0));
  #endif

  #if 0
    /* Dump the problem to a human readable OPF file. */
    MOSEKCALL(res, MSK_writedata(task, "dump.ptf"));
  #endif

    MOSEKCALL(res, MSK_optimizetrm(task, &trmcode));

    /* Display the solution summary for quick inspection of results. */
  # if 1
    MSK_solutionsummary(task, MSK_STREAM_MSG);
  # endif

  if (res == MSK_RES_OK)
  {
    expret = 0.0;
    stddev = 0.0;

    for (j = 0; j < n; ++j)
    {
      MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITR, voff_x + j, voff_x + j + 1, &
↪xj));
      expret += mu[j] * xj;
    }

    printf("\nExpected return %e for gamma %e\n", expret, gamma);
  }

  MSK_deletetask(&task);
  MSK_deleteenv(&env);

  free(m);
```

(continues on next page)

(continued from previous page)

```
  return (0);
}
```

Note that in the following part of the code:

```
  MOSEKCALL(res, MSK_putafeg(task, aoff_pow + 2 * n, 1.0));
  // We use one row from F and g for both c_j and z_j, and the last row of F and g␣
↪for the constant 1.
  // NOTE: Here we reuse the last AFE and the power cone n times, but we store them␣
↪only once.
  MSKrealt exponents[] = {2, 1};
  MSKint64t powdom;
  MOSEKCALL(res, MSK_appendprimalpowerconedomain(task, 3, 2, exponents, &powdom));
  MSKint64t* flat_afe_list = (MSKint64t*) malloc(3 * n * sizeof(MSKint64t));
  MSKint64t* dom_list = (MSKint64t*) malloc(n * sizeof(MSKint64t));
  for (i = 0; i < n; ++i)
  {
    flat_afe_list[3 * i + 0] = aoff_pow + i;
    flat_afe_list[3 * i + 1] = aoff_pow + 2 * n;
    flat_afe_list[3 * i + 2] = aoff_pow + n + i;
    dom_list[i] = powdom;
  }
  MOSEKCALL(res, MSK_appendaccs(task, n, dom_list, 3 * n, flat_afe_list, NULL));
  free(flat_afe_list);
  free(dom_list);
  for (i = 0; i < n; ++i)
  {
    sprintf(buf, "market_impact[%d]", i);
    MOSEKCALL(res, MSK_putaccname(task, i + 1, buf));
  }
```

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).

```
#include <math.h>
#include <stdio.h>

#include "mosek.h"

#define MOSEKCALL(_r,_call)  if ( (_r)==MSK_RES_OK ) (_r) = (_call)

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, const char **argv)
{
  char              buf[128];
  const MSKint32t n       = 8;
  const MSKrealt  mu[]    = {0.07197, 0.15518, 0.17535, 0.08981, 0.42896, 0.39292, 0.
↪32171, 0.18379};
  // GT must have size n rows
  const MSKrealt  GT[][8] = {
    {0.30758, 0.12146, 0.11341, 0.11327, 0.17625, 0.11973, 0.10435, 0.10638},
    {0.0,     0.25042, 0.09946, 0.09164, 0.06692, 0.08706, 0.09173, 0.08506},
    {0.0,     0.0,     0.19914, 0.05867, 0.06453, 0.07367, 0.06468, 0.01914},
    {0.0,     0.0,     0.0,     0.20876, 0.04933, 0.03651, 0.09381, 0.07742},
    {0.0,     0.0,     0.0,     0.0,     0.36096, 0.12574, 0.10157, 0.0571 },
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.21552, 0.05663, 0.06187},
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.22514, 0.03327},
    {0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.0,     0.2202 }
  };
  const MSKint32t k       = sizeof(GT) / (n * sizeof(MSKrealt));
  const MSKrealt  x0[]    = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
  const MSKrealt  w       = 1.0;
  const MSKrealt  gamma   = 0.36;

  MSKint32t       i, j;
  MSKrealt* f = (MSKrealt*) malloc(n * sizeof(MSKrealt));
  MSKrealt* g = (MSKrealt*) malloc(n * sizeof(MSKrealt));
  for (i = 0; i < n; ++i)
  {
    f[i] = 0.01;
    g[i] = 0.001;
  }

  // Offset of variables.
  MSKint32t numvar = 3 * n;
```

(continues on next page)

(continued from previous page)

```
  MSKint32t voff_x = 0;
  MSKint32t voff_z = n;
  MSKint32t voff_y = 2 * n;

  // Offset of constraints.
  MSKint32t numcon = 3 * n + 1;
  MSKint32t coff_bud = 0;
  MSKint32t coff_abs1 = 1;
  MSKint32t coff_abs2 = 1 + n;
  MSKint32t coff_swi = 1 + 2 * n;

  double          expret,
                  xj;
  MSKenv_t        env;
  MSKrescodee     res = MSK_RES_OK, trmcode;
  MSKtask_t       task;

  /* Initial setup. */
  env  = NULL;
  task = NULL;
  MOSEKCALL(res, MSK_makeenv(&env, NULL));
  MOSEKCALL(res, MSK_maketask(env, 0, 0, &task));
  MOSEKCALL(res, MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr));

  // Variables (vector of x, z, y)
  MOSEKCALL(res, MSK_appendvars(task, numvar));
  for (j = 0; j < n; ++j)
  {
    /* Optionally we can give the variables names */
    sprintf(buf, "x[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, voff_x + j, buf));
    sprintf(buf, "z[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, voff_z + j, buf));
    sprintf(buf, "y[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, voff_y + j, buf));
    /* Apply variable bounds (x >= 0, z free, y binary) */
    MOSEKCALL(res, MSK_putvarbound(task, voff_x + j, MSK_BK_LO, 0.0, MSK_INFINITY));
    MOSEKCALL(res, MSK_putvarbound(task, voff_z + j, MSK_BK_FR, -MSK_INFINITY, MSK_
↪INFINITY));
    MOSEKCALL(res, MSK_putvarbound(task, voff_y + j, MSK_BK_RA, 0.0, 1.0));
    MOSEKCALL(res, MSK_putvartype(task, voff_y + j, MSK_VAR_TYPE_INT));
  }

  // Linear constraints
  // - Total budget
  MOSEKCALL(res, MSK_appendcons(task, 1));
  sprintf(buf, "%s", "budget");
  MOSEKCALL(res, MSK_putconname(task, coff_bud, buf));
  for (j = 0; j < n; ++j)
  {
    /* Coefficients in the first row of A */
    MOSEKCALL(res, MSK_putaij(task, coff_bud, voff_x + j, 1.0));
    MOSEKCALL(res, MSK_putaij(task, coff_bud, voff_z + j, g[j]));
    MOSEKCALL(res, MSK_putaij(task, coff_bud, voff_y + j, f[j]));
  }
  MSKrealt U = w;
```

(continues on next page)

(continued from previous page)

```
  for (i = 0; i < n; ++i)
  {
    U += x0[i];
  }
  MOSEKCALL(res, MSK_putconbound(task, coff_bud, MSK_BK_FX, U, U));

  // - Absolute value
  MOSEKCALL(res, MSK_appendcons(task, 2 * n));
  for (i = 0; i < n; ++i)
  {
    sprintf(buf, "zabs1[%d]", 1 + i);
    MOSEKCALL(res, MSK_putconname(task, coff_abs1 + i, buf));
    MOSEKCALL(res, MSK_putaij(task, coff_abs1 + i, voff_x + i, -1.0));
    MOSEKCALL(res, MSK_putaij(task, coff_abs1 + i, voff_z + i, 1.0));
    MOSEKCALL(res, MSK_putconbound(task, coff_abs1 + i, MSK_BK_LO, -x0[i], MSK_
↪INFINITY));
    sprintf(buf, "zabs2[%d]", 1 + i);
    MOSEKCALL(res, MSK_putconname(task, coff_abs2 + i, buf));
    MOSEKCALL(res, MSK_putaij(task, coff_abs2 + i, voff_x + i, 1.0));
    MOSEKCALL(res, MSK_putaij(task, coff_abs2 + i, voff_z + i, 1.0));
    MOSEKCALL(res, MSK_putconbound(task, coff_abs2 + i, MSK_BK_LO, x0[i], MSK_
↪INFINITY));
  }

  // - Switch
  MOSEKCALL(res, MSK_appendcons(task, n));
  for (i = 0; i < n; ++i)
  {
    sprintf(buf, "switch[%d]", 1 + i);
    MOSEKCALL(res, MSK_putconname(task, coff_swi + i, buf));
    MOSEKCALL(res, MSK_putaij(task, coff_swi + i, voff_z + i, 1.0));
    MOSEKCALL(res, MSK_putaij(task, coff_swi + i, voff_y + i, -U));
    MOSEKCALL(res, MSK_putconbound(task, coff_swi + i, MSK_BK_UP, -MSK_INFINITY, 0.
↪0));
  }

  // ACCs
  MSKint32t aoff_q = 0;
  // - (gamma, GTx) in Q(k+1)
  // The part of F and g for variable x:
  //     [0,  0, 0]      [gamma]
  // F = [GT, 0, 0], g = [0    ]
  MOSEKCALL(res, MSK_appendafes(task, k + 1));
  MOSEKCALL(res, MSK_putafeg(task, aoff_q, gamma));
  MSKint32t* vslice_x = (MSKint32t*) malloc(n * sizeof(MSKint32t));
  for (i = 0; i < n; ++i)
  {
    vslice_x[i] = voff_x + i;
  }
  for (i = 0; i < k; ++i)
  {
    MOSEKCALL(res, MSK_putafefrow(task, aoff_q + i + 1, n, vslice_x, GT[i]));
  }
  free(vslice_x);
  MSKint64t qdom;
  MOSEKCALL(res, MSK_appendquadraticconedomain(task, k + 1, &qdom));
```

(continues on next page)

(continued from previous page)

```
  MOSEKCALL(res, MSK_appendaccseq(task, qdom, k + 1, aoff_q, NULL));
  sprintf(buf, "%s", "risk");
  MOSEKCALL(res, MSK_putaccname(task, aoff_q, buf));

  // Objective: maximize expected return mu^T x
  for (j = 0; j < n; ++j)
  {
    MOSEKCALL(res, MSK_putcj(task, voff_x + j, mu[j]));
  }
  MOSEKCALL(res, MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE));

#if 0
  /* no log output. */
  MOSEKCALL(res, MSK_putintparam(task, MSK_IPAR_LOG, 0));
#endif

#if 0
  /* Dump the problem to a human readable OPF file. */
  MOSEKCALL(res, MSK_writedata(task, "dump.ptf"));
#endif

  MOSEKCALL(res, MSK_optimizetrm(task, &trmcode));

#if 1
  /* Display the solution summary for quick inspection of results. */
  MSK_solutionsummary(task, MSK_STREAM_MSG);
#endif

  if (res == MSK_RES_OK)
  {
    expret = 0.0;

    for (j = 0; j < n; ++j)
    {
      MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITG, voff_x + j, voff_x + j + 1, &
↪xj));
      expret += mu[j] * xj;
    }

    printf("\nExpected return %e for gamma %e\n", expret, gamma);
  }

  MSK_deletetask(&task);
  MSK_deleteenv(&env);

  free(f);
  free(g);

  return (0);
}
```

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).

```
MSKrescodee markowitz_with_card(const int         n,
                                const int         k,
                                const double      x0[],
                                const double      w,
                                const double      gamma,
                                const double      mu[],
                                const double      GT[][8],
                                const int         K,
                                      double      *xx)
{

  // Offset of variables.
  MSKint32t numvar = 3 * n;
  MSKint32t voff_x = 0;
  MSKint32t voff_z = n;
  MSKint32t voff_y = 2 * n;

  // Offset of constraints.
  MSKint32t numcon = 3 * n + 2;
  MSKint32t coff_bud = 0;
  MSKint32t coff_abs1 = 1;
  MSKint32t coff_abs2 = 1 + n;
  MSKint32t coff_swi = 1 + 2 * n;
  MSKint32t coff_card = 1 + 3 * n;

  char            buf[128];
  MSKenv_t        env;
  MSKint32t       i, j;
  MSKrescodee     res = MSK_RES_OK, trmcode;
  MSKtask_t       task;
```

(continues on next page)

(continued from previous page)

```
  /* Initial setup. */
  env  = NULL;
  task = NULL;
  MOSEKCALL(res, MSK_makeenv(&env, NULL));
  MOSEKCALL(res, MSK_maketask(env, 0, 0, &task));
  MOSEKCALL(res, MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr));

  // Variables (vector of x, z, y)
  MOSEKCALL(res, MSK_appendvars(task, numvar));
  for (j = 0; j < n; ++j)
  {
    /* Optionally we can give the variables names */
    sprintf(buf, "x[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, voff_x + j, buf));
    sprintf(buf, "z[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, voff_z + j, buf));
    sprintf(buf, "y[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, voff_y + j, buf));
    /* Apply variable bounds (x >= 0, z free, y binary) */
    MOSEKCALL(res, MSK_putvarbound(task, voff_x + j, MSK_BK_LO, 0.0, MSK_INFINITY));
    MOSEKCALL(res, MSK_putvarbound(task, voff_z + j, MSK_BK_FR, -MSK_INFINITY, MSK_
↪INFINITY));
    MOSEKCALL(res, MSK_putvarbound(task, voff_y + j, MSK_BK_RA, 0.0, 1.0));
    MOSEKCALL(res, MSK_putvartype(task, voff_y + j, MSK_VAR_TYPE_INT));
  }

  // Linear constraints
  // - Total budget
  MOSEKCALL(res, MSK_appendcons(task, 1));
  sprintf(buf, "%s", "budget");
  MOSEKCALL(res, MSK_putconname(task, coff_bud, buf));
  for (j = 0; j < n; ++j)
  {
    /* Coefficients in the first row of A */
    MOSEKCALL(res, MSK_putaij(task, coff_bud, voff_x + j, 1.0));
  }
  MSKrealt U = w;
  for (i = 0; i < n; ++i)
  {
    U += x0[i];
  }
  MOSEKCALL(res, MSK_putconbound(task, coff_bud, MSK_BK_FX, U, U));

  // - Absolute value
  MOSEKCALL(res, MSK_appendcons(task, 2 * n));
  for (i = 0; i < n; ++i)
  {
    sprintf(buf, "zabs1[%d]", 1 + i);
    MOSEKCALL(res, MSK_putconname(task, coff_abs1 + i, buf));
    MOSEKCALL(res, MSK_putaij(task, coff_abs1 + i, voff_x + i, -1.0));
    MOSEKCALL(res, MSK_putaij(task, coff_abs1 + i, voff_z + i, 1.0));
    MOSEKCALL(res, MSK_putconbound(task, coff_abs1 + i, MSK_BK_LO, -x0[i], MSK_
↪INFINITY));
    sprintf(buf, "zabs2[%d]", 1 + i);
    MOSEKCALL(res, MSK_putconname(task, coff_abs2 + i, buf));
    MOSEKCALL(res, MSK_putaij(task, coff_abs2 + i, voff_x + i, 1.0));
```

(continues on next page)

(continued from previous page)

```
    MOSEKCALL(res, MSK_putaij(task, coff_abs2 + i, voff_z + i, 1.0));
    MOSEKCALL(res, MSK_putconbound(task, coff_abs2 + i, MSK_BK_LO, x0[i], MSK_
↪INFINITY));
  }

  // - Switch
  MOSEKCALL(res, MSK_appendcons(task, n));
  for (i = 0; i < n; ++i)
  {
    sprintf(buf, "switch[%d]", 1 + i);
    MOSEKCALL(res, MSK_putconname(task, coff_swi + i, buf));
    MOSEKCALL(res, MSK_putaij(task, coff_swi + i, voff_z + i, 1.0));
    MOSEKCALL(res, MSK_putaij(task, coff_swi + i, voff_y + i, -U));
    MOSEKCALL(res, MSK_putconbound(task, coff_swi + i, MSK_BK_UP, -MSK_INFINITY, 0.
↪0));
  }

  // - Cardinality
  MOSEKCALL(res, MSK_appendcons(task, 1));
  sprintf(buf, "cardinality");
  MOSEKCALL(res, MSK_putconname(task, coff_card, buf));
  for (i = 0; i < n; ++i)
  {
    MOSEKCALL(res, MSK_putaij(task, coff_card, voff_y + i, 1.0));
    MOSEKCALL(res, MSK_putconbound(task, coff_card, MSK_BK_UP, -MSK_INFINITY, K));
  }

  // ACCs
  MSKint32t aoff_q = 0;
  // - (gamma, GTx) in Q(k+1)
  // The part of F and g for variable x:
  //     [0,   0, 0]       [gamma]
  // F = [GT, 0, 0], g = [0     ]
  MOSEKCALL(res, MSK_appendafes(task, k + 1));
  MOSEKCALL(res, MSK_putafeg(task, aoff_q, gamma));
  MSKint32t* vslice_x = (MSKint32t*) malloc(n * sizeof(MSKint32t));
  for (i = 0; i < n; ++i)
  {
    vslice_x[i] = voff_x + i;
  }
  for (i = 0; i < k; ++i)
  {
    MOSEKCALL(res, MSK_putafefrow(task, aoff_q + i + 1, n, vslice_x, GT[i]));
  }
  free(vslice_x);
  MSKint64t qdom;
  MOSEKCALL(res, MSK_appendquadraticconedomain(task, k + 1, &qdom));
  MOSEKCALL(res, MSK_appendaccseq(task, qdom, k + 1, aoff_q, NULL));
  sprintf(buf, "%s", "risk");
  MOSEKCALL(res, MSK_putaccname(task, aoff_q, buf));

  // Objective: maximize expected return mu^T x
  for (j = 0; j < n; ++j)
  {
    MOSEKCALL(res, MSK_putcj(task, voff_x + j, mu[j]));
  }
```

(continues on next page)

(continued from previous page)

```
  MOSEKCALL(res, MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE));

#if 0
  /* no log output. */
  MOSEKCALL(res, MSK_putintparam(task, MSK_IPAR_LOG, 0));
#endif

#if 0
  /* Dump the problem to a human readable OPF file. */
  MOSEKCALL(res, MSK_writedata(task, "dump.ptf"));
#endif

  MOSEKCALL(res, MSK_optimizetrm(task, &trmcode));

#if 1
  /* Display the solution summary for quick inspection of results. */
  MSK_solutionsummary(task, MSK_STREAM_MSG);
#endif

  if (res == MSK_RES_OK)
    MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITG, voff_x, voff_x + n, xx));

  MSK_deletetask(&task);
  MSK_deleteenv(&env);

  return res;
}
```

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

$$\begin{array}{lllll} \text{minimize} & \sum_i t_i + \lambda r & & & \\ \text{subject to} & t_i & \geq -\log(h_\theta(x)) & = \log(1+\exp(-\theta^T x_i)) & \text{if } y_i = 1, \\ & t_i & \geq -\log(1-h_\theta(x)) & = \log(1+\exp(\theta^T x_i)) & \text{if } y_i = 0, \\ & r & \geq \|\theta\|_2. & & \end{array} \tag{11.13}$$

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
MSKrescodee softplus(MSKtask_t task, int d, int n, MSKint32t theta, MSKint32t t,␣
↪double* X, int* y)
{
  MSKint32t nvar, ncon;
  MSKint64t nafe, thetaafe, tafe, z1afe, z2afe, oneafe, expdomain;
  MSKint32t z1, z2, zcon, v1con, v2con;
  MSKint32t  *subi = (MSKint32t*) calloc(2*n, sizeof(MSKint32t));
  MSKint32t  *subj = (MSKint32t*) calloc(3*n, sizeof(MSKint32t));
  MSKrealt   *aval = (MSKrealt*) calloc(2*n, sizeof(MSKrealt));
  MSKint64t  *afeidx = (MSKint64t*) calloc(d*n+4*n, sizeof(MSKint64t));
  MSKint32t  *varidx = (MSKint32t*) calloc(d*n+4*n, sizeof(MSKint32t));
  MSKrealt   *fval   = (MSKrealt*) calloc(d*n+4*n, sizeof(MSKrealt));
  MSKint64t  idx[3];
  int        k, i, j;
  MSKrescodee res = MSK_RES_OK;

  MSKCALL(MSK_getnumvar(task, &nvar));
  MSKCALL(MSK_getnumcon(task, &ncon));
  MSKCALL(MSK_getnumafe(task, &nafe));
  MSKCALL(MSK_appendvars(task, 2*n));   // z1, z2
  MSKCALL(MSK_appendcons(task, n));     // z1 + z2 = 1
  MSKCALL(MSK_appendafes(task, 4*n));   //theta * X[i] - t[i], -t[i], z1[i], z2[i]

  z1 = nvar, z2 = nvar+n;
  zcon = ncon;
  thetaafe = nafe, tafe = nafe+n, z1afe = nafe+2*n, z2afe = nafe+3*n;

  // Linear constraints
  k = 0;
  for(i = 0; i < n; i++)
  {
    // z1 + z2 = 1
    subi[k] = zcon+i;  subj[k] = z1+i;  aval[k] = 1;  k++;
    subi[k] = zcon+i;  subj[k] = z2+i;  aval[k] = 1;  k++;
  }
  MSKCALL(MSK_putaijlist(task, 2*n, subi, subj, aval));
  MSKCALL(MSK_putconboundsliceconst(task, zcon, zcon+n, MSK_BK_FX, 1, 1));
  MSKCALL(MSK_putvarboundsliceconst(task, nvar, nvar+2*n, MSK_BK_FR, -inf, inf));

  // Affine conic expressions
  k = 0;

  // Thetas
  for(i = 0; i < n; i++) {
    for(j = 0; j < d; j++) {
      afeidx[k] = thetaafe + i; varidx[k] = theta + j;
      fval[k] = ((y[i]) ? -1 : 1) * X[i*d+j];
      k++;
    }
  }

  // -t[i]
  for(i = 0; i < n; i++) {
```

(continues on next page)

(continued from previous page)

```
    afeidx[k] = thetaafe + i; varidx[k] = t + i; fval[k] = -1; k++;
    afeidx[k] = tafe + i;     varidx[k] = t + i; fval[k] = -1; k++;
  }

  // z1, z2
  for(i = 0; i < n; i++) {
    afeidx[k] = z1afe + i; varidx[k] = z1 + i; fval[k] = 1; k++;
    afeidx[k] = z2afe + i; varidx[k] = z2 + i; fval[k] = 1; k++;
  }

  // Add the expressions
  MSKCALL(MSK_putafefentrylist(task, d*n+4*n, afeidx, varidx, fval));

  // Add a single row with the constant expression "1.0"
  MSKCALL(MSK_getnumafe(task, &oneafe));
  MSKCALL(MSK_appendafes(task,1));
  MSKCALL(MSK_putafeg(task, oneafe, 1.0));

  // Add an exponential cone domain
  MSKCALL(MSK_appendprimalexpconedomain(task, &expdomain));

  // Conic constraints
  for(i = 0; i < n; i++)
  {
    idx[0] = z1afe+i, idx[1] = oneafe, idx[2] = thetaafe+i;
    MSKCALL(MSK_appendacc(task, expdomain, 3, idx, NULL));
    idx[0] = z2afe+i, idx[1] = oneafe, idx[2] = tafe+i;
    MSKCALL(MSK_appendacc(task, expdomain, 3, idx, NULL));
  }

  free(subi); free(subj); free(aval);
  free(afeidx); free(varidx); free(fval);
  return res;
}
```

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
MSKrescodee logisticRegression(MSKenv_t       env,
                               int            n,    // num samples
                               int            d,    // dimension
                               double        *X,
                               int           *y,
                               double         lamb,
                               double        *thetaVal)   // result
{
  MSKrescodee res = MSK_RES_OK;
  MSKrescodee trm = MSK_RES_OK;
  MSKtask_t task = NULL;
  MSKint32t nvar = 1+d+n;
  MSKint32t r = 0, theta = 1, t = 1+d;
```

(continues on next page)

(continued from previous page)

```
  MSKint64t numafe, quadDom;
  int i = 0;

  MSKCALL(MSK_maketask(env, 0, 0, &task));
  MSKCALL(MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr));

  // Variables [r; theta; t]
  MSKCALL(MSK_appendvars(task, nvar));
  MSKCALL(MSK_putvarboundsliceconst(task, 0, nvar, MSK_BK_FR, -inf, inf));

  // Objective lambda*r + sum(t)
  MSKCALL(MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MINIMIZE));
  MSKCALL(MSK_putcj(task, r, lamb));
  for(i = 0; i < n && res == MSK_RES_OK; i++)
    MSKCALL(MSK_putcj(task, t+i, 1.0));

  // Softplus function constraints
  MSKCALL(softplus(task, d, n, theta, t, X, y));

  // Regularization
  // Append a sequence of linear expressions (r, theta) to F
  MSKCALL(MSK_getnumafe(task, &numafe));
  MSKCALL(MSK_appendafes(task,1+d));
  MSKCALL(MSK_putafefentry(task, numafe, r, 1.0));
  for(i = 0; i < d; i++)
    MSKCALL(MSK_putafefentry(task, numafe + i + 1, theta + i, 1.0));

  // Add the constraint
  MSKCALL(MSK_appendquadraticconedomain(task, 1+d, &quadDom));
  MSKCALL(MSK_appendaccseq(task, quadDom, 1+d, numafe, NULL));

  // Solution
  MSKCALL(MSK_optimizetrm(task, &trm));
  MSKCALL(MSK_solutionsummary(task, MSK_STREAM_MSG));

  MSKCALL(MSK_getxxslice(task, MSK_SOL_ITR, theta, theta+d, thetaVal));

  return res;
}
```

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.

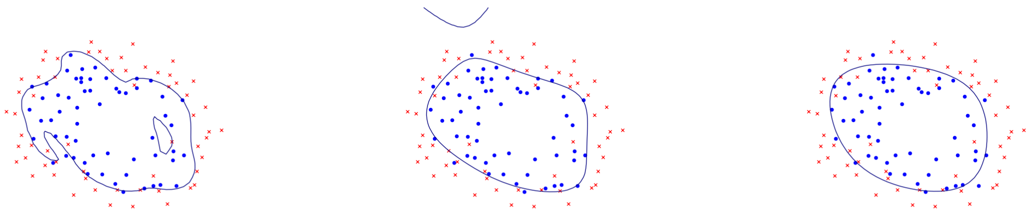

Fig. 11.2: Logistic regression example with none, medium and strong regularization (small, medium, large λ). Without regularization we get obvious overfitting.

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 (C++).

```
/**
   Defines a Mosek callback function whose only function
   is to indicate if the optimizer should be stopped.
 */
int stop = 0;
int firstStop = -1;
MSKint32t MSKAPI callback (MSKtask_t task, MSKuserhandle_t usrptr, MSKcallbackcodee␣
↪caller,
                                    const MSKrealt * douinf, const MSKint32t * intinf, const␣
↪MSKint64t * lintinf)
{
  return stop;
}
```

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 (C++).

```
void runTask(int         num,
             MSKtask_t   task,
             MSKrescodee *res,
             MSKrescodee *trm)
{
  *res = MSK_optimizetrm(task, trm);
  if (*res == MSK_RES_OK) {
    if (!stop) firstStop = num;
    stop = 1;
  }
}

int optimize(int         n,
             MSKtask_t   *tasks,
             MSKrescodee *res,
             MSKrescodee *trm)
{
  int i;
  std::thread * jobs = new std::thread[n];

  // Set a callback function and start optimization
  for(i = 0; i < n; ++i) {
    MSK_putcallbackfunc(tasks[i], callback, NULL);
    res[i] = trm[i] = MSK_RES_ERR_UNKNOWN;
    jobs[i] = std::thread(runTask, i, tasks[i], &(res[i]), &(trm[i]));
  }

  // Join all threads
  for(i = 0; i < n; ++i) jobs[i].join();
  delete[] jobs;

  // For debugging, print res and trm codes for all optimizers
  for(i = 0; i < n; ++i)
    printf("Optimizer  %d   res %d   trm %d\n", i, res[i], trm[i]);

  return firstStop;
}
```

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 (C++).

```
int optimizeconcurrent(MSKtask_t           task,
                       int                 n,
                       MSKoptimizertypee   *optimizers,
                       MSKtask_t           *winTask,
                       MSKrescodee         *winTrm,
                       MSKrescodee         *winRes)
{
  MSKtask_t   *tasks = new MSKtask_t[n];
```

(continues on next page)

(continued from previous page)

```
  MSKrescodee *res   = new MSKrescodee[n];
  MSKrescodee *trm   = new MSKrescodee[n];

  // Clone tasks and choose various optimizers
  for (int i = 0; i < n; ++i)
  {
    MSK_clonetask(task, &(tasks[i]));
    MSK_putintparam(tasks[i], MSK_IPAR_OPTIMIZER, optimizers[i]);
  }

  // Solve tasks in parallel
  int firstOK = optimize(n, tasks, res, trm);

  if (firstOK >= 0)
  {
    *winTask  = tasks[firstOK];
    *winTrm   = trm[firstOK];
    *winRes   = res[firstOK];
  }
  else
  {
    *winTask  = NULL;
    *winTrm   = MSK_RES_ERR_UNKNOWN;
    *winRes   = MSK_RES_ERR_UNKNOWN;
  }

  // Cleanup
  for (int i = 0; i < n; ++i)
    if (i != firstOK) MSK_deletetask(&(tasks[i]));

  delete[] tasks; delete[] res; delete[] trm;
  return firstOK;
}
```

It remains to call the method with a choice of optimizers, for example:

Listing 11.13: Calling concurrent linear optimization (C++).

```
    MSKoptimizertypee optimizers[3] = {
      MSK_OPTIMIZER_CONIC,
      MSK_OPTIMIZER_DUAL_SIMPLEX,
      MSK_OPTIMIZER_PRIMAL_SIMPLEX
    };

    idx = optimizeconcurrent(task, 3, optimizers, &t, &trm, &res);
```

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 (C++).

```
int optimizeconcurrentMIO(MSKtask_t            task,
                          int                  n,
                          int                  *seeds,
                          MSKtask_t            *winTask,
                          MSKrescodee          *winTrm,
                          MSKrescodee          *winRes)
{
  MSKtask_t   *tasks = new MSKtask_t[n];
  MSKrescodee *res   = new MSKrescodee[n];
  MSKrescodee *trm   = new MSKrescodee[n];
  *winTask  = NULL;
  *winTrm   = MSK_RES_ERR_UNKNOWN;
  *winRes   = MSK_RES_ERR_UNKNOWN;
  int bestPos = -1;

  // Clone tasks and choose various optimizers
  for (int i = 0; i < n; ++i)
  {
    MSK_clonetask(task, &(tasks[i]));
    MSK_putintparam(tasks[i], MSK_IPAR_MIO_SEED, seeds[i]);
  }

  // Solve tasks in parallel
  int firstOK = optimize(n, tasks, res, trm);

  if (firstOK >= 0)
  {
    // Pick the task that ended with res = ok
    // and contains an integer solution with best objective value
    MSKobjsensee sense;
    double bestObj;

    MSK_getobjsense(task, &sense);
    bestObj = (sense == MSK_OBJECTIVE_SENSE_MINIMIZE) ? 1.0e+10 : -1.0e+10;

    for (int i = 0; i < n; ++i) {
      double priObj;
      MSK_getprimalobj(tasks[i], MSK_SOL_ITG, &priObj);
      printf("%d      %f\n", i, priObj);
    }

    for (int i = 0; i < n; ++i) {
      double priObj;
      MSKsolstae solsta;
      MSK_getprimalobj(tasks[i], MSK_SOL_ITG, &priObj);
      MSK_getsolsta(tasks[i], MSK_SOL_ITG, &solsta);
```

(continues on next page)

(continued from previous page)

```
      if ((res[i] == MSK_RES_OK) &&
          (solsta == MSK_SOL_STA_PRIM_FEAS ||
           solsta == MSK_SOL_STA_INTEGER_OPTIMAL) &&
          ((sense == MSK_OBJECTIVE_SENSE_MINIMIZE) ?
              (priObj < bestObj) : (priObj > bestObj) ) )
      {
        bestObj = priObj;
        bestPos = i;
      }
    }

    if (bestPos != -1)
    {
      *winTask  = tasks[bestPos];
      *winTrm   = trm[bestPos];
      *winRes   = res[bestPos];
    }
  }

  // Cleanup
  for (int i = 0; i < n; ++i)
    if (i != bestPos) MSK_deletetask(&(tasks[i]));

  delete[] tasks; delete[] res; delete[] trm;
  return bestPos;
}
```

It remains to call the method with a choice of seeds, for example:

Listing 11.15: Calling concurrent integer optimization (C++).

```
    int seeds[3] = { 42, 13, 71749373 };

    idx = optimizeconcurrentMIO(task, 3, seeds, &t, &trm, &res);
```

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

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

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

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

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

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

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

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

to (12.4) so that

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

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

Dual Infeasible Problems

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

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

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

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

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

such that the objective value is strictly negative

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

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

12.2 Conic Optimization

Conic optimization is an extension of linear optimization (see Sec. 12.1) allowing conic domains to be specified for 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. 15.10.

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

$$
\begin{array}{ll}
\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x - g^T \dot{y} + c^f \\
\text{subject to} & \\
 & A^T y + s_l^x - s_u^x + F^T \dot{y} = c, \\
 & -y + s_l^c - s_u^c = 0, \\
 & s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \\
 & \dot{y} \in \mathcal{D}^*,
\end{array}
\tag{12.8}
$$

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

$$\begin{array}{lcccc} \text{minimize} & & c^T x & & \\ \text{subject to} & \hat{l}^c \leq & Ax & \leq & \hat{u}^c, \\ & \hat{l}^x \leq & x & \leq & \hat{u}^x, \\ & & Fx & \in \mathcal{D} & \end{array} \tag{12.12}$$

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

$$\begin{array}{c}
A^T y + s_l^x - s_u^x + F^T \dot{y} = 0, \\
-y + s_l^c - s_u^c = 0, \\
s_l^c, s_u^c, s_l^x, s_u^x \leq 0, \\
-\dot{y} \in \mathcal{D}^*
\end{array} \qquad (12.15)$$

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. 15.10.

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:

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x - g^T \dot{y} + c^f \\ \text{subject to} & \\ & \begin{array}{rl} A^T y + s_l^x - s_u^x + F^T \dot{y} = c, & \\ -y + s_l^c - s_u^c = 0, & \\ \overline{C}_j - \sum_{i=1}^m y_i \overline{A}_{ij} - \sum_{i=1}^k \dot{y}_i \overline{F}_{ij} = S_j, & j = 1, \ldots, s, \\ s_l^c, s_u^c, s_l^x, s_u^x \geq 0, & \\ \dot{y} \in \mathcal{D}^*, & \\ \overline{S}_j \in \mathcal{S}_+^{r_j}, & j = 1, \ldots, s. \end{array} \end{array} \tag{12.16}$$

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:

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x - g^T \dot{y} \\ \text{subject to} & \\ & \begin{array}{rl} A^T y + s_l^x - s_u^x + F^T \dot{y} = 0, & \\ -y + s_l^c - s_u^c = 0, & \\ -\sum_{i=1}^m y_i \overline{A}_{ij} - \sum_{i=1}^k \dot{y}_i \overline{F}_{ij} = S_j, & j = 1, \ldots, s, \\ s_l^c, s_u^c, s_l^x, s_u^x \geq 0, & \\ \dot{y} \in \mathcal{D}^*, & \\ \overline{S}_j \in \mathcal{S}_+^{r_j}, & j = 1, \ldots, s. \end{array} \end{array} \tag{12.18}$$

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

$$\begin{array}{lccccl} \text{minimize} & & & \frac{1}{2} x^T Q^o x + c^T x + c^f & & \\ \text{subject to} & l_k^c & \leq & \frac{1}{2} x^T Q^k x + \sum_{j=0}^{n-1} a_{kj} x_j & \leq & u_k^c, \quad k = 0, \ldots, m-1, \\ & l_j^x & \leq & x_j & \leq & u_j^x, \quad j = 0, \ldots, n-1, \end{array} \tag{12.20}$$

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

$$\begin{array}{lc}
\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + \frac{1}{2} x^T \left\{ \sum_{k=0}^{m-1} y_k Q^k - Q^o \right\} x + c^f \\
\text{subject to} & A^T y + s_l^x - s_u^x + \left\{ \sum_{k=0}^{m-1} y_k Q^k - Q^o \right\} x = c, \\
 & -y + s_l^c - s_u^c = 0, \\
 & s_l^c, s_u^c, s_l^x, s_u^x \geq 0.
\end{array} \tag{12.21}$$

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)**

$$\begin{array}{lrcccl} \text{minimize} & & & c^T x + c^f & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x. \end{array}$$

- **Conic optimization (CO)**

  Conic optimization extends linear optimization with *affine conic constraints* (ACC):

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

  where $\mathcal{D}$ is a product of domains from Sec. 15.10.

- **Semidefinite optimization (SDO)**

  A conic optimization problem can be further extended with *semidefinite variables*:

$$\begin{array}{lrcccl} \text{minimize} & & & c^T x + \langle \overline{C}, \overline{X} \rangle + c^f & & \\ \text{subject to} & l^c & \leq & Ax + \langle \overline{A}, \overline{X} \rangle & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \\ & & & Fx + \langle \overline{F}, \overline{X} \rangle + g & \in & \mathcal{D}, \\ & & & \overline{X} & \in & \mathcal{S}_+, \end{array}$$

  where $\mathcal{D}$ is a product of domains from Sec. 15.10 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

$$\begin{array}{lrcccl} \text{minimize} & & & \frac{1}{2} x^T Q^o x + c^T x + c^f & & \\ \text{subject to} & l^c & \leq & \frac{1}{2} x^T Q^c x + Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x. \end{array}$$

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. Available for problems of type LO, CO, QO and QCQO.

- **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. Available for problems of type LO and CO.

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

$$\begin{array}{lccc} \text{minimize} & c^T x & & \\ \text{subject to} & Ax & = & b, \\ & x \geq 0. & & \end{array} \tag{13.1}$$

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

$$\begin{array}{rcl} \left\| A\frac{x^k}{\tau^k} - b \right\|_\infty & \leq & \epsilon_p (1 + \|b\|_\infty), \\ \left\| A^T\frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c \right\|_\infty & \leq & \epsilon_d (1 + \|c\|_\infty), \text{ and} \\ \min\left(\frac{(x^k)^T s^k}{(\tau^k)^2}, \left|\frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k}\right|\right) & \leq & \epsilon_g \max\left(1, \frac{\min(|c^T x^k|, |b^T y^k|)}{\tau^k}\right), \end{array} \tag{13.5}$$

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

$$\begin{array}{lll} (x_1^*, y_1^*) & = & (1, 0), \\ (x_2^*, y_2^*) & = & (0, 1). \end{array}$$

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
4   1.3e-004 3.2e-004 2.0e-005 1.08e+000  -7.668323435e+003 -7.668207177e+003 3.2e-
↪004 0.01
```

(continues on next page)

(continued from previous page)

```
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                  ␣
↪     0.21      0.22
```

(continues on next page)

(continued from previous page)

```
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:

$$\begin{array}{lccc} \text{minimize} & c^T x & & \\ \text{subject to} & Ax & = & b, \\ & x \in \mathcal{K} & & \end{array} \tag{13.6}$$

where K is a convex cone. The corresponding dual problem is

$$\begin{array}{lccc} \text{maximize} & b^T y & & \\ \text{subject to} & A^T y + s & = & c, \\ & s \in \mathcal{K}^* & & \end{array} \tag{13.7}$$

where $\mathcal{K}^*$ is the dual cone of $\mathcal{K}$. See Sec. 12.2 for definitions.

Since it is not known beforehand whether problem (13.6) has an optimal solution, is primal infeasible or is dual infeasible, the optimization algorithm must deal with all three situations. This is the reason that **MOSEK** solves the so-called homogeneous model

$$\begin{array}{rcl} Ax - b\tau & = & 0, \\ A^T y + s - c\tau & = & 0, \\ -c^T x + b^T y - \kappa & = & 0, \\ x & \in & \mathcal{K}, \\ s & \in & \mathcal{K}^*, \\ \tau, \kappa & \geq & 0, \end{array} \tag{13.8}$$

where y and s correspond to the dual variables in (13.6), and τ and κ are two additional scalar variables. Note that the homogeneous model (13.8) always has a solution since

$$(x, y, s, \tau, \kappa) = (0, 0, 0, 0, 0)$$

is a solution, although not a very interesting one. Any solution

$$(x^*, y^*, s^*, \tau^*, \kappa^*)$$

to the homogeneous model (13.8) satisfies

$$(x^*)^T s^* + \tau^* \kappa^* = 0$$

i.e. complementarity. Observe that $x^* \in \mathcal{K}$ and $s^* \in \mathcal{K}^*$ implies

$$(x^*)^T s^* \geq 0$$

and therefore

$$\tau^* \kappa^* = 0.$$

since $\tau^*, \kappa^* \geq 0$. Hence, at least one of τ^* and κ^* is zero.

First, assume that $\tau^* > 0$ and hence $\kappa^* = 0$. It follows that

$$\begin{array}{rcl} A\frac{x^*}{\tau^*} & = & b, \\ A^T\frac{y^*}{\tau^*} + \frac{s^*}{\tau^*} & = & c, \\ -c^T\frac{x^*}{\tau^*} + b^T\frac{y^*}{\tau^*} & = & 0, \\ x^*/\tau^* & \in & \mathcal{K}, \\ s^*/\tau^* & \in & \mathcal{K}^*. \end{array}$$

This shows that $\frac{x^*}{\tau^*}$ is a primal optimal solution and $(\frac{y^*}{\tau^*}, \frac{s^*}{\tau^*})$ is a dual optimal solution; this is reported as the optimal interior-point solution since

$$(x, y, s) = \left(\frac{x^*}{\tau^*}, \frac{y^*}{\tau^*}, \frac{s^*}{\tau^*}\right)$$

is a primal-dual optimal solution.

On other hand, if $\kappa^* > 0$ then

$$\begin{array}{rcl} Ax^* & = & 0, \\ A^Ty^* + s^* & = & 0, \\ -c^Tx^* + b^Ty^* & = & \kappa^*, \\ x^* & \in & \mathcal{K}, \\ s^* & \in & \mathcal{K}^*. \end{array}$$

This implies that at least one of

$$c^T x^* < 0 \tag{13.9}$$

or

$$b^T y^* > 0 \tag{13.10}$$

holds. If (13.9) is satisfied, then x^* is a certificate of dual infeasibility, whereas if (13.10) holds then y^* is a certificate of primal infeasibility.

In summary, by computing an appropriate solution to the homogeneous model, all information required for a solution to the original problem is obtained. A solution to the homogeneous model can be computed using a primal-dual interior-point algorithm [And09].

13.3.2 Interior-point Termination Criterion

Since computations are performed in finite precision, and for efficiency reasons, it is not possible to solve the homogeneous model exactly in general. Hence, an exact optimal solution or an exact infeasibility certificate cannot be computed and a reasonable termination criterion has to be employed.

In every iteration k of the interior-point algorithm a trial solution

$$(x^k, y^k, s^k, \tau^k, \kappa^k)$$

to the homogeneous model is generated, where

$$x^k \in \mathcal{K}, s^k \in \mathcal{K}^*, \tau^k, \kappa^k > 0.$$

Therefore, it is possible to compute the values:

$$\begin{array}{rcl}
\rho_p^k & = & \arg\min_\rho \left\{\rho \mid \left\|A\frac{x^k}{\tau^k} - b\right\|_\infty \leq \rho\varepsilon_p(1 + \|b\|_\infty)\right\}, \\
\rho_d^k & = & \arg\min_\rho \left\{\rho \mid \left\|A^T\frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c\right\|_\infty \leq \rho\varepsilon_d(1 + \|c\|_\infty)\right\}, \\
\rho_g^k & = & \arg\min_\rho \left\{\rho \mid \left(\frac{(x^k)^T s^k}{(\tau^k)^2}, \left|\frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k}\right|\right) \leq \rho\varepsilon_g \max\left(1, \frac{\min(|c^T x^k|, |b^T y^k|)}{\tau^k}\right)\right\}, \\
\rho_{pi}^k & = & \arg\min_\rho \left\{\rho \mid \left\|A^T y^k + s^k\right\|_\infty \leq \rho\varepsilon_i b^T y^k,\, b^T y^k > 0\right\} \text{ and} \\
\rho_{di}^k & = & \arg\min_\rho \left\{\rho \mid \left\|Ax^k\right\|_\infty \leq -\rho\varepsilon_i c^T x^k,\, c^T x^k < 0\right\}.
\end{array}$$

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
Factor     - ML order time          : 0.00              GP order time          : 0.00
```

(continues on next page)

(continued from previous page)

```
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, a predominant special case of integer variables are binary variables, taking values in $\{0, 1\}$. Mixed- or pure binary problems are important subclasses of mixed-integer optimization where all integer variables are of this type. 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**. 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

$$\begin{array}{rclccc} z^* & = & \text{minimize} & c^T x & & \\ & & \text{subject to} & Ax & = & b \\ & & & x \geq 0 & & \\ & & & x_j \in \mathbb{Z}, & & \forall j \in \mathcal{J}. \end{array} \tag{13.12}$$

It has the continuous relaxation

$$\begin{array}{rclccc} \underline{z} & = & \text{minimize} & c^T x & & \\ & & \text{subject to} & Ax & = & b \\ & & & x \geq 0, & & \end{array} \tag{13.13}$$

obtained simply by ignoring the integrality restrictions. The 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 push the variable away from the fractional value, closer towards integrality. If the variable was binary, say, 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 a relaxation solution $\hat{x}$ does not violate any integrality constraints, it is feasible to (13.12) and is called an *integer feasible solution*. Clearly, its solution value $\bar{z} := c^T \hat{x}$ is an upper bound on the optimal objective value,

$$z^* \leq \bar{z}.$$

Since refining a relaxation by adding constraints to it can only increase its solution value, 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. If as upper bound $\bar{z}$ one records the solution value of the best integer feasible solution encountered so far, the so-called *incumbent solution*, $\bar{z}$ 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.

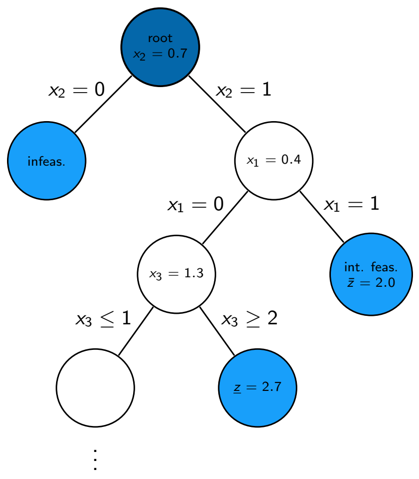

Fig. 13.1: An examplary Branch-and-Bound tree. Pruned nodes are shown in light blue.

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. Recalling the Branch-and-Bound scheme from the previous section, one may see that 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*

$$\epsilon_{\mathrm{rel}} = \frac{|\bar{z} - \underline{z}|}{\max(\delta_1, |\bar{z}|)}.$$

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

$$\bar{z} - \underline{z} \leq \max(\delta_2, \delta_3 \max(\delta_1, |\bar{z}|))$$

is checked. If satisfied, i.e., if either absolute or relative optimality gap are below the thresholds δ_2 or δ_3, respectively, the optimizer terminates and reports the incumbent as an optimal solution. The optimality gaps 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 most 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 does not terminate because the objective bound $\underline{z}$ is of poor quality. Instead, the vast majority of computational time is spent on trying to improve $\underline{z}$: 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 Additional components of the mixed-integer Optimizer

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 to the user. 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.

Presolve

Similar to the case of continuous problems, see Sec. 13.1, the mixed-integer optimizer applies various presolve reductions before the actual solution process is initiated. Just as in the continuous case, the use of presolve can be controlled with the parameter *MSK_IPAR_PRESOLVE_USE*.

Primal Heuristics

Solving relaxations in the Branch-and-bound 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 13.5.

MIO performance tweaks: primal heuristics

- If the mixed-integer optimizer struggles to improve the incumbent solution $\bar{z}$, see Sec. 13.4.4, 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 13.5. 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 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 13.5: 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_FEASPUMP_LEVEL	Way of using the Feasibility Pump heuristic.

Cutting Planes

Cutting planes (cuts) are simply constraints that are valid for a mixed-integer problem, for example in the form (13.12), meaning they do not remove any integer feasible solutions from the feasible set. Therefore they are also called valid inequalities. They do not have to be valid for the relaxation (13.13) though, and of interest and potentially useful are those cuts that do remove solutions from the feasible set of the relaxation. The latter is a superset of the feasible region of the mixed-integer problem, and the rationale behind cuts is thus to bring the integer problem and its relaxation closer together in terms of their feasible sets.

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. In fact, 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, but cuts can also be generated later on in the Branch-and-Bound tree. 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 13.6.

MIO performance tweaks: cutting planes

- If the mixed-integer optimizer struggles to improve the objective bound $\underline{z}$, see Sec. 13.4.4, 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 amount of time on cutting planes, see Sec. 13.4.4, 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 13.6: 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_ROUNDS	Maximum number of root cut rounds.
MSK_DPAR_MIO_TOL_REL_DUAL_BOUND_IMPROVEMENT	Minimum required objective bound improvement during root cut generation.

13.4.4 The Mixed-Integer Log

Below is a typical log output from the mixed-integer optimizer:

```
Presolved problem: 1176 variables, 1344 constraints, 4968 non-zeros
Presolved problem: 328 general integer, 392 binary, 456 continuous
Clique table size: 55
BRANCHES RELAXS   ACT_NDS  DEPTH    BEST_INT_OBJ         BEST_RELAX_OBJ       REL_GAP(
↪%)  TIME
0        0        1        0        8.3888091139e+07     NA                   NA      ␣
↪    0.0
0        1        1        0        8.3888091139e+07     2.5492512136e+07     69.61   ␣
↪    0.1
0        1        1        0        3.1273162420e+07     2.5492512136e+07     18.48   ␣
↪    0.1
0        1        1        0        2.6047699632e+07     2.5492512136e+07     2.13    ␣
↪    0.2
Cut generation started.
0        1        1        0        2.6047699632e+07     2.5492512136e+07     2.13    ␣
↪    0.2
0        2        1        0        2.6047699632e+07     2.5589986247e+07     1.76    ␣
↪    0.2
Cut generation terminated. Time = 0.05
0        4        1        0        2.5990071367e+07     2.5662741991e+07     1.26    ␣
↪    0.3
0        8        1        0        2.5971002767e+07     2.5662741991e+07     1.19    ␣
↪    0.5
```

(continues on next page)

(continued from previous page)

```
0        11       1        0        2.5925040617e+07     2.5662741991e+07     1.01    ␣
↪    0.5
0        12       1        0        2.5915504014e+07     2.5662741991e+07     0.98    ␣
↪    0.5
2        23       1        0        2.5915504014e+07     2.5662741991e+07     0.98    ␣
↪    0.6
14       35       1        0        2.5915504014e+07     2.5662741991e+07     0.98    ␣
↪    0.6

[ ... ]

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     : 0.03
Time spend optimizing the root     : 0.07
Mixed integer optimizer terminated. Time: 6.46
```

The first lines contain a summary of the problem after mixed-integer presolve has been applied. This is followed by the iteration log, 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.

The beginning and the end of the root cut generation is highlighted as well, and the number of log lines in between reflects to the computational effort spent here.

Finally there is a summary of the optimization process, containing also information on the type of generated cuts and the total number of iterations needed to solve all occuring continuous relaxations.

When the solution time for a mixed-integer problem has to be cut down, it can sometimes be useful to examine the log in order to understand where time is spent and what might be improved. In particular, it might happen that the values in either of the colums `BEST_INT_OBJ` or `BEST_RELAX_OBJ` stall over a long period of log lines, an indication that the optimizer has a hard time improving either the incumbent solution, i.e., $\bar{z}$, or the objective bound $\underline{z}$, see also Sec. 13.4.3 and Sec. 13.4.3.

13.4.5 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.

13.4.6 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.

13.4.7 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-Bound 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*.

13.4.8 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

- When relaxations in the the Branch-and-Bound 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 *MSK_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

$$\begin{array}{lccc} \text{minimize} & x+z & & \\ \text{subject to} & x & = & -1, \\ & x & \geq & 0 \end{array}$$

is repaired then it will become unbounded. Hence, a repaired problem may not have an optimal solution.

Another and more important caveat is that only a minimal repair is performed i.e. the repair that barely makes the problem feasible. Hence, the repaired problem is barely feasible and that sometimes makes the repaired problem hard to solve.

Using the automatic repair tool

In this subsection we consider an infeasible linear optimization example:

$$\begin{array}{lccccc} \text{minimize} & -10x_1 & & -9x_2, & & \\ \text{subject to} & 7/10x_1 & + & 1x_2 & \leq & 630, \\ & 1/2x_1 & + & 5/6x_2 & \leq & 600, \\ & 1x_1 & + & 2/3x_2 & \leq & 708, \\ & 1/10x_1 & + & 1/4x_2 & \leq & 135, \\ & x_1, & & x_2 & \geq & 0, \\ & & & x_2 & \geq & 650. \end{array} \tag{14.2}$$

The function *MSK_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).

```
#include <math.h>
#include <stdio.h>

#include "mosek.h"

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  fputs(str, stdout);
} /* printstr */

static const char * feasrepair_lp =
"minimize                                  \n"
" obj: - 10 x1 - 9 x2                      \n"
"st                                        \n"
" c1: + 7e-01 x1 + x2 <= 630               \n"
" c2: + 5e-01 x1 + 8.333333333e-01 x2 <= 600 \n"
" c3: + x1 + 6.6666667e-01 x2 <= 708       \n"
" c4: + 1e-01 x1 + 2.5e-01 x2 <= 135       \n"
"bounds                                    \n"
"x2 >= 650                                 \n"
"end                                       \n";

int main(int argc, const char *argv[])
{
  MSKenv_t    env;
  MSKrescodee r, trmcode;
  MSKtask_t   task;

```

(continues on next page)

(continued from previous page)

```
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
    r = MSK_makeemptytask(env, &task);

  if (r == MSK_RES_OK)
    MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

  if (r == MSK_RES_OK)
    r = MSK_readlpstring(task,feasrepair_lp); /* Read problem from string */

  if (r == MSK_RES_OK)
    r = MSK_putintparam(task, MSK_IPAR_LOG_FEAS_REPAIR, 3);

  if (r == MSK_RES_OK)
  {
    /* Weights are NULL implying all weights are 1. */
    r = MSK_primalrepair(task, NULL, NULL, NULL, NULL);
  }

  if (r == MSK_RES_OK)
  {
    double sum_viol;

    r = MSK_getdouinf(task, MSK_DINF_PRIMAL_REPAIR_PENALTY_OBJ, &sum_viol);

    if (r == MSK_RES_OK)
    {
      printf("Minimized sum of violations = %e\n", sum_viol);

      r = MSK_optimizetrm(task, &trmcode); /* Optimize the repaired task. */

      MSK_solutionsummary(task, MSK_STREAM_MSG);
    }
  }

  printf("Return code: %d\n", r);

  return (r);
}
```

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
```

(continues on next page)

(continued from previous page)

```
  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.
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
```

(continues on next page)

(continued from previous page)

```
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.
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

$$
\begin{array}{rclccccc}
z(l^c, u^c, l^x, u^x, c) & = & \text{minimize} & & & c^T x & & \\
 & & \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\
 & & & l^x & \leq & x & \leq & u^x,
\end{array}
\tag{14.3}
$$

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

$$
\begin{array}{rclcccc}
f_{l_i^c}(\beta) & = & \text{minimize} & & & c^T x & \\
 & & \text{subject to} & l^c + \beta e_i & \leq & Ax & \leq u^c, \\
 & & & l^x & \leq & x \leq & u^x,
\end{array}
$$

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.

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*

$$
\beta \in [\beta_1, \beta_2]
$$

for which

$$
f'_{l_i^c}(\beta) = f'_{l_i^c}(0).
$$

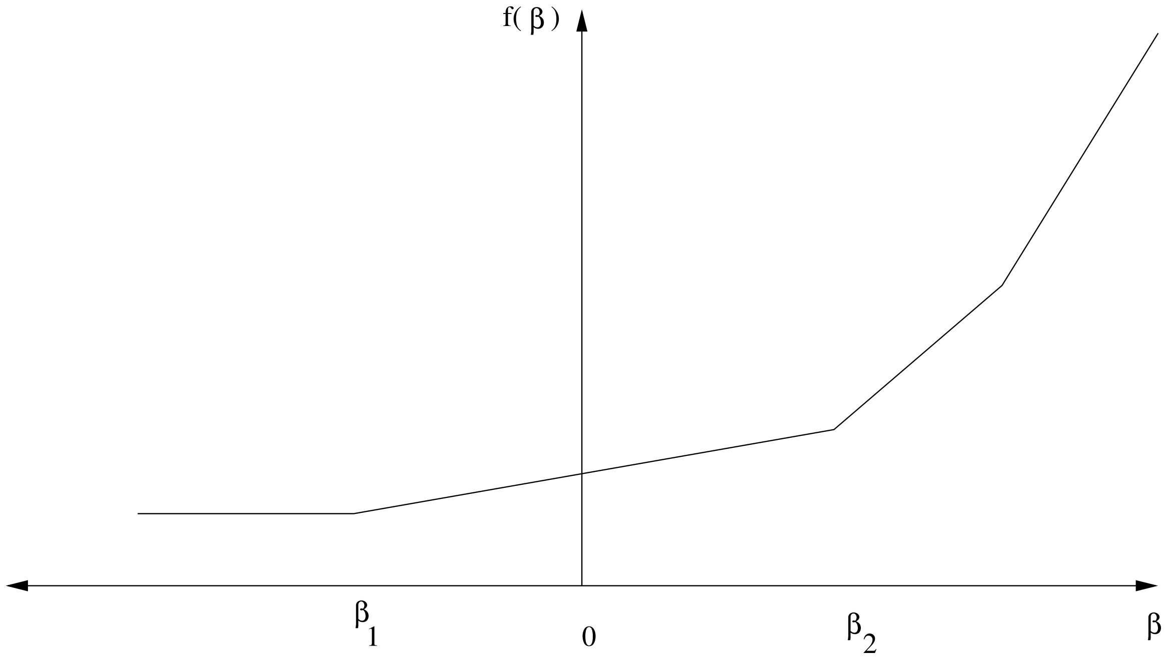

Fig. 14.1: $\beta = 0$ is in the interior of linearity interval.

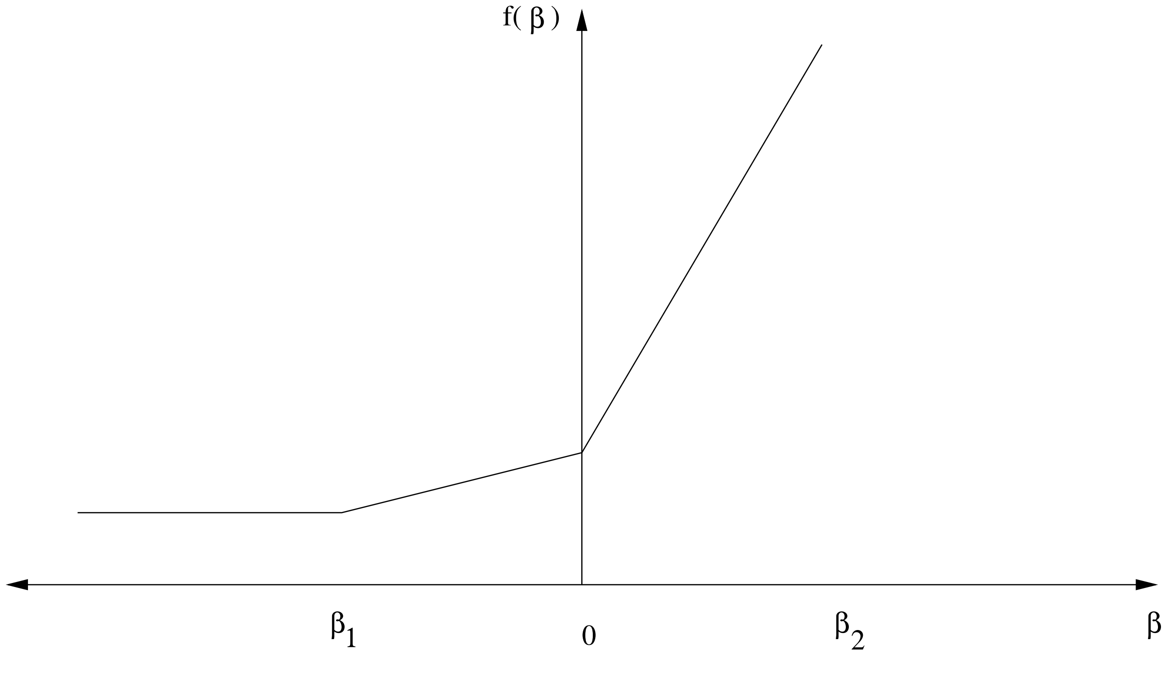

Fig. 14.2: $\beta = 0$ is a breakpoint.

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.

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

$$\begin{array}{ccccccccccccccc} x_{11} & + & x_{12} & & & & & & & & & & & \leq & 400, \\ & & & & x_{23} & + & x_{24} & & & & & & & \leq & 1200, \\ & & & & & & & & x_{31} & + & x_{33} & + & x_{34} & \leq & 1000, \\ x_{11} & & & & & & & + & x_{31} & & & & & = & 800, \\ & & x_{12} & & & & & & & & & & & = & 100, \\ & & & & x_{23} & + & & & & & x_{33} & & & = & 500, \\ & & & & & & x_{24} & + & & & & & x_{34} & = & 500, \\ x_{11}, & & x_{12}, & & x_{23}, & & x_{24}, & & x_{31}, & & x_{33}, & & x_{34} & \geq & 0. \end{array} \tag{14.5}$$

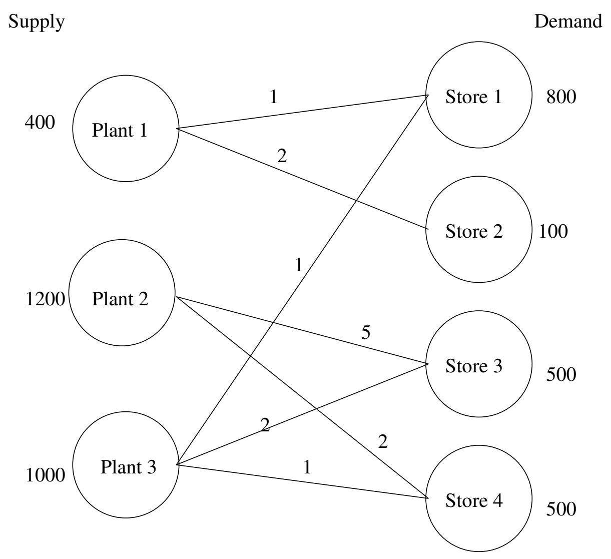

Fig. 14.3: Supply, demand and cost of transportation.

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 *MSK_primalsensitivity* and *MSK_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 C.

```
#include <stdio.h>

#include "mosek.h" /* Include the MOSEK definition file. */

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  printf("%s", str);
} /* printstr */

int main(int argc, char *argv[])
{
  const MSKint32t numcon = 7,
                  numvar = 7;
  MSKint32t       i, j;
  MSKboundkeye    bkc[] = {MSK_BK_UP, MSK_BK_UP, MSK_BK_UP, MSK_BK_FX,
                           MSK_BK_FX, MSK_BK_FX, MSK_BK_FX
                          };
  MSKboundkeye    bkx[] = {MSK_BK_LO, MSK_BK_LO, MSK_BK_LO,
                           MSK_BK_LO, MSK_BK_LO, MSK_BK_LO, MSK_BK_LO
                          };
  MSKint32t       ptrb[] = {0, 2, 4, 6, 8, 10, 12};
  MSKint32t       ptre[] = {2, 4, 6, 8, 10, 12, 14};
  MSKint32t       sub[] = {0, 3, 0, 4, 1, 5, 1, 6, 2, 3, 2, 5, 2, 6};
  MSKrealt        blc[] = { -MSK_INFINITY, -MSK_INFINITY, -MSK_INFINITY, 800, 100,␣
↪500, 500};
  MSKrealt        buc[] = {400,          1200,         1000,         800, 100, 500,␣
↪500};
  MSKrealt        c[]   = {1.0, 2.0, 5.0, 2.0, 1.0, 2.0, 1.0};
  MSKrealt        blx[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
  MSKrealt        bux[] = {MSK_INFINITY, MSK_INFINITY, MSK_INFINITY, MSK_INFINITY,
                           MSK_INFINITY, MSK_INFINITY, MSK_INFINITY
                          };
  MSKrealt        val[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
↪ 1.0, 1.0};
  MSKrescodee     r;

  MSKenv_t        env;
  MSKtask_t       task;
```

(continues on next page)

(continued from previous page)

```
  /* Create mosek environment. */
  r = MSK_makeenv(&env, NULL);

  if (r == MSK_RES_OK)
  {
    /* Make the optimization task. */
    r = MSK_makeemptytask(env, &task);

    if (r == MSK_RES_OK)
    {
      /* Directs the log task stream to the user
         specified procedure 'printstr'. */

      MSK_linkfunctotaskstream(task, MSK_STREAM_LOG, NULL, printstr);

      MSK_echotask(task,
                   MSK_STREAM_MSG,
                   "Defining the problem data.\n");
    }

    /* Append the constraints. */
    if (r == MSK_RES_OK)
      r = MSK_appendcons(task, numcon);

    /* Append the variables. */
    if (r == MSK_RES_OK)
      r = MSK_appendvars(task, numvar);

    /* Put C. */
    if (r == MSK_RES_OK)
      r = MSK_putcfix(task, 0.0);

    if (r == MSK_RES_OK)
      r = MSK_putcslice(task, 0, numvar, c);

    /* Put constraint bounds. */
    if (r == MSK_RES_OK)
      r = MSK_putconboundslice(task, 0, numcon, bkc, blc, buc);

    /* Put variable bounds. */
    if (r == MSK_RES_OK)
      r = MSK_putvarboundslice(task, 0, numvar, bkx, blx, bux);

    /* Put A. */
    if (r == MSK_RES_OK)
      r = MSK_putacolslice(task, 0, numvar, ptrb, ptre, sub, val);

    if (r == MSK_RES_OK)
      r = MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MINIMIZE);

    if (r == MSK_RES_OK)
      r = MSK_optimizetrm(task, NULL);

    if (r == MSK_RES_OK)
    {
      /* Analyze upper bound on c1 and the equality constraint on c4 */
```

(continues on next page)

(continued from previous page)

```
    MSKint32t subi[] = {0, 3};
    MSKmarke marki[] = {MSK_MARK_UP, MSK_MARK_UP};

    /* Analyze lower bound on the variables x12 and x31 */
    MSKint32t subj[] = {1, 4};
    MSKmarke markj[] = {MSK_MARK_LO, MSK_MARK_LO};

    MSKrealt leftpricei[2];
    MSKrealt rightpricei[2];
    MSKrealt leftrangei[2];
    MSKrealt rightrangei[2];
    MSKrealt leftpricej[2];
    MSKrealt rightpricej[2];
    MSKrealt leftrangej[2];
    MSKrealt rightrangej[2];

    r = MSK_primalsensitivity(task,
                              2,
                              subi,
                              marki,
                              2,
                              subj,
                              markj,
                              leftpricei,
                              rightpricei,
                              leftrangei,
                              rightrangei,
                              leftpricej,
                              rightpricej,
                              leftrangej,
                              rightrangej);

    printf("Results from sensitivity analysis on bounds:\n");

    printf("For constraints:\n");
    for (i = 0; i < 2; ++i)
      printf("leftprice = %e, rightprice = %e,leftrange = %e, rightrange =%e\n",
             leftpricei[i], rightpricei[i], leftrangei[i], rightrangei[i]);

    printf("For variables:\n");
    for (i = 0; i < 2; ++i)
      printf("leftprice = %e, rightprice = %e,leftrange = %e, rightrange =%e\n",
             leftpricej[i], rightpricej[i], leftrangej[i], rightrangej[i]);
  }

  if (r == MSK_RES_OK)
  {
    MSKint32t subj[]  = {2, 5};
    MSKrealt  leftprice[2];
    MSKrealt  rightprice[2];
    MSKrealt  leftrange[2];
    MSKrealt  rightrange[2];

    r = MSK_dualsensitivity(task,
                            2,
                            subj,
```

(continues on next page)

(continued from previous page)

```
                              leftprice,
                              rightprice,
                              leftrange,
                              rightrange
                             );

      printf("Results from sensitivity analysis on objective coefficients:\n");

      for (i = 0; i < 2; ++i)
        printf("leftprice = %e, rightprice = %e,leftrange = %e, rightrange =%e\n",
               leftprice[i], rightprice[i], leftrange[i], rightrange[i]);
    }

    MSK_deletetask(&task);
  }

  MSK_deleteenv(&env);

  printf("Return code: %d (0 means no error occured.)\n", r);
  return (r);
} /* main */
```

Chapter 15

API Reference

This section contains the complete reference of the **MOSEK** Optimizer API for C. 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*
- *User-defined function types*
- *Simple data types*
- *List of supported domains*

15.1 API Conventions

15.1.1 Function arguments

Naming Convention

In the definition of the **MOSEK** Optimizer API for C 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 C.

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	MSKboundkeye*	numcon	l_k^c and u_k^c
bkx	MSKboundkeye*	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: $q^o_{\texttt{qosubi[t]},\texttt{qosubj[t]}} = \texttt{qoval[t]}, \quad t = 0, \ldots, \texttt{numqonz} - 1.$
- The linear terms in the objective : $c_j = \texttt{c[j]}, \quad j = 0, \ldots, \texttt{numvar} - 1$
- The fixed term in the objective : $c^f = \texttt{cfix}.$
- The quadratic terms in the constraints: $q^{\texttt{qcsubk[t]}}_{\texttt{qcsubi[t]},\texttt{qcsubj[t]}} = \texttt{qcval[t]}, \quad t = 0, \ldots, \texttt{numqcnz} - 1$
- The linear terms in the constraints: $a_{\texttt{asub[t]},\texttt{j}} = \texttt{aval[t]}, \quad t = \texttt{ptrb[j]}, \ldots, \texttt{ptre[j]} - 1, \quad j = 0, \ldots, \texttt{numvar} - 1$

Passing arguments by reference

An argument described as **T** *by reference* indicates that the function interprets its given argument as a reference to a variable of type **T**. This usually means that the argument is used to output or update a value of type **T**. For example, suppose we have a function documented as

```
MSKrescodee MSK_foo (..., int * nzc, ...)
```

- nzc (**int** *by reference*) – The number of nonzero elements in the matrix. (output)

Then it could be called as follows.

```
int nzc;
MSK_foo (..., &nzc, ...)
printf("The number of nonzero elements: %d\n", nzc)
```

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

$$l_k^c = \texttt{blc}[\texttt{k}], \quad k = 0, \ldots, \texttt{numcon} - 1$$

$$u_k^c = \texttt{buc}[\texttt{k}], \quad k = 0, \ldots, \texttt{numcon} - 1.$$

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

$$l_j^x = \texttt{blx}[\texttt{j}], \quad j = 0, \ldots, \texttt{numvar} - 1.$$

$$u_j^x = \texttt{bux}[\texttt{j}], \quad j = 0, \ldots, \texttt{numvar} - 1.$$

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

```
MSKrealt * c = MSK_calloctask(task, numvar, sizeof(MSKrealt));

if ( c )
  res = MSK_getc(task,c);
else
  printf("Out of space\n");
```

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

```
    MSKrealt * upper_bound   = MSK_calloctask(task,8,sizeof(MSKrealt));
    MSKrealt * lower_bound   = MSK_calloctask(task,8,sizeof(MSKrealt));
    MSKboundkeye * bound_key = MSK_calloctask(task,8,sizeof(MSKboundkeye));
    res = MSK_getconboundslice(task, 2,10,
                               bound_key,lower_bound,upper_bound);
```

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

```
    MSKint32t    bound_index[] = {         1,         6,         3,         9 };
    MSKboundkeye bound_key[]   = { MSK_BK_FR, MSK_BK_LO, MSK_BK_UP, MSK_BK_FX };
    MSKrealt     lower_bound[] = {       0.0,     -10.0,       0.0,       5.0 };
    MSKrealt     upper_bound[] = {       0.0,       0.0,       6.0,       5.0 };

    res = MSK_putconboundlist(task, 4, 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.

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:

```
    MSKint32t subi[] = {   1,   3,   5 },
              subj[] = {   2,   3,   4 };
    MSKrealt  cof[]  = { 1.1, 4.3, 0.2 };

    res = MSK_putaijlist(task,3, subi,subj,cof);
```

Please note that in some cases (like *MSK_putaijlist*) *only* the specified indexes are modified — all other are unchanged. In other cases (such as *MSK_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:

$$a_{\texttt{asub}[k],j} = \texttt{aval}[k] \quad \text{for} \quad j = 0, \ldots, \texttt{numcol} - 1, \; k = \texttt{ptrb}[j], \ldots, \texttt{ptre}[j] - 1.$$

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

$$\begin{aligned} \texttt{ptrb} &= [0, 2, 3, 5, 7], \\ \texttt{ptre} &= [2, 3, 5, 7, 8], \\ \texttt{asub} &= [0, 2, 1, 0, 3, 0, 2, 1], \\ \texttt{aval} &= [1.1, 3.1, 2.2, 1.3, 4.4, 1.4, 3.4, 2.5]. \end{aligned}$$

Fig. 15.1 illustrates how the matrix A in (15.1) is represented in column ordered sparse matrix format.

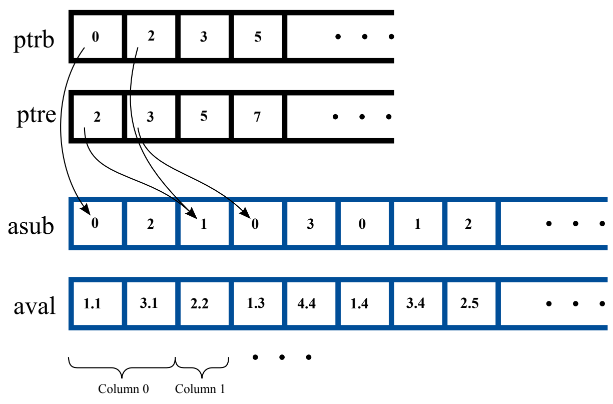

Fig. 15.1: The matrix A (15.1) represented in column ordered packed sparse matrix format.

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:

$$\begin{aligned} \texttt{ptrb} &= [0, 2, 3, 5, 7], \\ \texttt{nzc} &= [2, 1, 2, 2, 1], \\ \texttt{asub} &= [0, 2, 1, 0, 3, 0, 2, 1], \\ \texttt{aval} &= [1.1, 3.1, 2.2, 1.3, 4.4, 1.4, 3.4, 2.5]. \end{aligned}$$

Row ordered sparse matrix

The matrix A (15.1) can also be represented in the row ordered sparse matrix format as:

$$\begin{array}{lcl} \texttt{ptrb} & = & [0, 3, 5, 7], \\ \texttt{ptre} & = & [3, 5, 7, 8], \\ \texttt{asub} & = & [0, 2, 3, 1, 4, 0, 3, 2], \\ \texttt{aval} & = & [1.1, 1.3, 1.4, 2.2, 2.5, 3.1, 3.4, 4.4]. \end{array}$$

15.2 Functions grouped by topic

Callback

- *MSK_linkfunctotaskstream* – Connects a user-defined function to a task stream.
- *MSK_putcallbackfunc* – Input the progress callback function.
- *Infrequent:* *MSK_getcallbackfunc*, *MSK_linkfunctoenvstream*, *MSK_putexitfunc*, *MSK_putresponsefunc*, *MSK_unlinkfuncfromenvstream*, *MSK_unlinkfuncfromtaskstream*

Environment and task management

- *MSK_clonetask* – Creates a clone of an existing task.
- *MSK_deleteenv* – Delete a MOSEK environment.
- *MSK_deletetask* – Deletes a task.
- *MSK_makeenv* – Creates a new MOSEK environment.
- *MSK_maketask* – Creates a new task.
- *MSK_puttaskname* – Assigns a new name to the task.
- *Infrequent:* *MSK_commitchanges*, *MSK_deletesolution*, *MSK_getenv*, *MSK_makeemptytask*, *MSK_putmaxnumacc*, *MSK_putmaxnumafe*, *MSK_putmaxnumanz*, *MSK_putmaxnumbarvar*, *MSK_putmaxnumcon*, *MSK_putmaxnumdjc*, *MSK_putmaxnumdomain*, *MSK_putmaxnumqnz*, *MSK_putmaxnumvar*, *MSK_resizetask*
- *Deprecated:* ~~*MSK_putmaxnumcone*~~

Infeasibility diagnostic

- *MSK_getinfeasiblesubproblem* – Obtains an infeasible subproblem.
- *MSK_infeasibilityreport* – Prints the infeasibility report to an output stream.
- *MSK_primalrepair* – Repairs a primal infeasible optimization problem by adjusting the bounds on the constraints and variables.

Information items and statistics

- *MSK_getdouinf* – Obtains a double information item.
- *MSK_getintinf* – Obtains an integer information item.
- *MSK_getlintinf* – Obtains a long integer information item.
- *MSK_updatesolutioninfo* – Update the information items related to the solution.
- *Infrequent:* *MSK_getinfindex*, *MSK_getinfmax*, *MSK_getinfname*, *MSK_getnadouinf*, *MSK_getnaintinf*

Input/Output

- *MSK_readdata* – Reads problem data from a file.
- *MSK_writedata* – Writes problem data to a file.
- *MSK_writedatahandle* – Writes problem data to a user-defined handle.
- *MSK_writesolution* – Write a solution to a file.
- *Infrequent:* *MSK_echotask*, *MSK_readbsolution*, *MSK_readdataautoformat*, *MSK_readdatacb*, *MSK_readdataformat*, *MSK_readjsonsol*, *MSK_readjsonstring*, *MSK_readlpstring*, *MSK_readopfstring*, *MSK_readparamfile*, *MSK_readptfstring*, *MSK_readsolution*, *MSK_readsolutionfile*, *MSK_readsummary*, *MSK_readtask*, *MSK_writebsolution*, *MSK_writejsonsol*, *MSK_writeparamfile*, *MSK_writesolutionfile*, *MSK_writetask*

Inspecting the task

- *MSK_analyzeproblem* – Analyze the data of a task.
- *MSK_getnumcon* – Obtains the number of constraints.
- *MSK_getnumvar* – Obtains the number of variables.
- *Infrequent:* *MSK_analyzesolution*, *MSK_getaccafeidxlist*, *MSK_getaccb*, *MSK_getaccbarfnumblocktriplets*, *MSK_getaccdomain*, *MSK_getaccfnumnz*, *MSK_getaccftrip*, *MSK_getaccgvector*, *MSK_getaccn*, *MSK_getaccname*, *MSK_getaccnamelen*, *MSK_getaccntot*, *MSK_getaccs*, *MSK_getacol*, *MSK_getacolnumnz*, *MSK_getacolslice*, *MSK_getacolslice64*, *MSK_getacolslicenumnz*, *MSK_getacolslicenumnz64*, *MSK_getacolslicetrip*, *MSK_getafebarfnumblocktriplets*, *MSK_getafebarfnumrowentries*, *MSK_getafebarfrow*, *MSK_getafebarfrowinfo*, *MSK_getafefnumnz*, *MSK_getafefrow*, *MSK_getafefrownumnz*, *MSK_getafeftrip*, *MSK_getafeg*, *MSK_getafegslice*, *MSK_getaij*, *MSK_getapiecenumnz*, *MSK_getarow*, *MSK_getarownumnz*, *MSK_getarowslice*, *MSK_getarowslice64*, *MSK_getarowslicenumnz*, *MSK_getarowslicenumnz64*, *MSK_getarowslicetrip*, *MSK_getatrip*, *MSK_getbarablocktriplet*, *MSK_getbaraidx*, *MSK_getbaraidxij*, *MSK_getbaraidxinfo*, *MSK_getbarasparsity*, *MSK_getbarcblocktriplet*, *MSK_getbarcidx*, *MSK_getbarcidxinfo*, *MSK_getbarcidxj*, *MSK_getbarcsparsity*, *MSK_getbarvarname*, *MSK_getbarvarnameindex*, *MSK_getbarvarnamelen*, *MSK_getc*, *MSK_getcfix*, *MSK_getcj*, *MSK_getclist*, *MSK_getconbound*, *MSK_getconboundslice*, *MSK_getconname*, *MSK_getconnameindex*, *MSK_getconnamelen*, *MSK_getcslice*, *MSK_getdimbarvarj*, *MSK_getdjcafeidxlist*, *MSK_getdjcb*, *MSK_getdjcdomainidxlist*, *MSK_getdjcname*, *MSK_getdjcnamelen*, *MSK_getdjcnumafe*, *MSK_getdjcnumafetot*, *MSK_getdjcnumdomain*, *MSK_getdjcnumdomaintot*, *MSK_getdjcnumterm*, *MSK_getdjcnumtermtot*, *MSK_getdjcs*, *MSK_getdjctermsizelist*, *MSK_getdomainn*, *MSK_getdomainname*, *MSK_getdomainnamelen*, *MSK_getdomaintype*, *MSK_getlenbarvarj*, *MSK_getmaxnamelen*, *MSK_getmaxnumanz*, *MSK_getmaxnumanz64*, *MSK_getmaxnumbarvar*, *MSK_getmaxnumcon*, *MSK_getmaxnumqnz*, *MSK_getmaxnumqnz64*, *MSK_getmaxnumvar*, *MSK_getnumacc*, *MSK_getnumafe*, *MSK_getnumanz*, *MSK_getnumanz64*, *MSK_getnumbarablocktriplets*, *MSK_getnumbaranz*, *MSK_getnumbarcblocktriplets*, *MSK_getnumbarcnz*, *MSK_getnumbarvar*, *MSK_getnumdjc*, *MSK_getnumdomain*, *MSK_getnumintvar*, *MSK_getnumparam*, *MSK_getnumqconknz*, *MSK_getnumqconknz64*, *MSK_getnumqobjnz*, *MSK_getnumqobjnz64*, *MSK_getnumsymmat*, *MSK_getobjname*, *MSK_getobjnamelen*, *MSK_getpowerdomainalpha*, *MSK_getpowerdomaininfo*, *MSK_getprobtype*, *MSK_getqconk*, *MSK_getqconk64*, *MSK_getqobj*, *MSK_getqobj64*, *MSK_getqobjij*, *MSK_getsparsesymmat*, *MSK_getsymmatinfo*, *MSK_gettaskname*, *MSK_gettasknamelen*, *MSK_getvarbound*, *MSK_getvarboundslice*, *MSK_getvarname*, *MSK_getvarnameindex*, *MSK_getvarnamelen*, *MSK_getvartype*, *MSK_getvartypelist*, *MSK_printparam*, *MSK_probtypetostr*, *MSK_readsummary*
- *Deprecated:* ~~*MSK_getcone*~~, ~~*MSK_getconeinfo*~~, ~~*MSK_getconename*~~, ~~*MSK_getconenameindex*~~, ~~*MSK_getconenamelen*~~, ~~*MSK_getmaxnumcone*~~, ~~*MSK_getnumcone*~~, ~~*MSK_getnumconemem*~~

License system

- *MSK_checkoutlicense* – Check out a license feature from the license server ahead of time.
- *MSK_putlicensedebug* – Enables debug information for the license system.
- *MSK_putlicensepath* – Set the path to the license file.
- *MSK_putlicensewait* – Control whether mosek should wait for an available license if no license is available.
- *Infrequent:* *MSK_checkinall*, *MSK_checkinlicense*, *MSK_expirylicenses*, *MSK_licensecleanup*, *MSK_putlicensecode*, *MSK_resetexpirylicenses*

Linear algebra

- *Infrequent:* *MSK_axpy*, *MSK_computesparsecholesky*, *MSK_dot*, *MSK_gemm*, *MSK_gemv*, *MSK_potrf*, *MSK_sparsetriangularsolvedense*, *MSK_syeig*, *MSK_syevd*, *MSK_syrk*

Logging

- *MSK_linkfiletotaskstream* – Directs all output from a task stream to a file.
- *MSK_linkfunctotaskstream* – Connects a user-defined function to a task stream.
- *MSK_onesolutionsummary* – Prints a short summary of a specified solution.
- *MSK_optimizersummary* – Prints a short summary with optimizer statistics from last optimization.
- *MSK_solutionsummary* – Prints a short summary of the current solutions.
- *Infrequent:* *MSK_echoenv*, *MSK_echointro*, *MSK_linkfiletoenvstream*, *MSK_linkfunctoenvstream*, *MSK_printparam*, *MSK_putexitfunc*, *MSK_putresponsefunc*, *MSK_unlinkfuncfromenvstream*, *MSK_unlinkfuncfromtaskstream*

Names

- *MSK_getcodedesc* – Obtains a short description of a response code.
- *MSK_putaccname* – Sets the name of an affine conic constraint.
- *MSK_putbarvarname* – Sets the name of a semidefinite variable.
- *MSK_putconname* – Sets the name of a constraint.
- *MSK_putdjcname* – Sets the name of a disjunctive constraint.
- *MSK_putdomainname* – Sets the name of a domain.
- *MSK_putobjname* – Assigns a new name to the objective.
- *MSK_puttaskname* – Assigns a new name to the task.
- *MSK_putvarname* – Sets the name of a variable.
- *Infrequent:* *MSK_analyzenames*, *MSK_bktostr*, *MSK_callbackcodetostr*, *MSK_generateaccnames*, *MSK_generatebarvarnames*, *MSK_generateconnames*, *MSK_generatedjcnames*, *MSK_generatevarnames*, *MSK_getaccname*, *MSK_getaccnamelen*, *MSK_getbarvarname*, *MSK_getbarvarnameindex*, *MSK_getbarvarnamelen*, *MSK_getconname*, *MSK_getconnameindex*, *MSK_getconnamelen*, *MSK_getdjcname*, *MSK_getdjcnamelen*, *MSK_getdomainname*, *MSK_getdomainnamelen*, *MSK_getinfname*, *MSK_getmaxnamelen*, *MSK_getnastrparam*, *MSK_getobjname*, *MSK_getobjnamelen*, *MSK_getparamname*, *MSK_getstrparam*, *MSK_getstrparamlen*, *MSK_getsymbcon*, *MSK_gettaskname*, *MSK_gettasknamelen*, *MSK_getvarname*, *MSK_getvarnameindex*, *MSK_getvarnamelen*,

MSK_iparvaltosymnam, MSK_isdouparname, MSK_isintparname, MSK_isstrparname, MSK_probtypetostr, MSK_prostatostr, MSK_sktostr, MSK_solstatostr, MSK_strtosk, MSK_whichparam

- *Deprecated:* ~~MSK_conetypetostr~~, ~~MSK_generateconenames~~, ~~MSK_getconename~~, ~~MSK_getconenameindex~~, ~~MSK_getconenamelen~~, ~~MSK_putconename~~, ~~MSK_strtoconetype~~

Optimization

- MSK_optimize – Optimizes the problem.
- MSK_optimizebatch – Optimize a number of tasks in parallel using a specified number of threads.
- MSK_optimizetrm – Optimizes the problem.

Parameters

- MSK_putdouparam – Sets a double parameter.
- MSK_putintparam – Sets an integer parameter.
- MSK_putparam – Modifies the value of parameter.
- MSK_putstrparam – Sets a string parameter.
- MSK_setdefaults – Resets all parameter values.
- *Infrequent:* MSK_getatruncatetol, MSK_getdouparam, MSK_getintparam, MSK_getnadouparam, MSK_getnaintparam, MSK_getnastrparam, MSK_getnastrparamal, MSK_getnumparam, MSK_getparammax, MSK_getparamname, MSK_getstrparam, MSK_getstrparamal, MSK_getstrparamlen, MSK_getsymbcondim, MSK_iparvaltosymnam, MSK_isdouparname, MSK_isintparname, MSK_isstrparname, MSK_putnadouparam, MSK_putnaintparam, MSK_putnastrparam, MSK_readparamfile, MSK_symnamtovalue, MSK_whichparam, MSK_writeparamfile

Problem data - affine conic constraints

- MSK_appendacc – Appends an affine conic constraint to the task.
- MSK_getaccdoty – Obtains the doty vector for an affine conic constraint.
- MSK_putaccname – Sets the name of an affine conic constraint.
- *Infrequent:* MSK_appendaccs, MSK_appendaccseq, MSK_appendaccsseq, MSK_evaluateacc, MSK_evaluateaccs, MSK_getaccafeidxlist, MSK_getaccb, MSK_getaccbarfnumblocktriplets, MSK_getaccdomain, MSK_getaccdotys, MSK_getaccfnumnz, MSK_getaccftrip, MSK_getaccgvector, MSK_getaccn, MSK_getaccname, MSK_getaccnamelen, MSK_getaccntot, MSK_getaccs, MSK_getnumacc, MSK_putacc, MSK_putaccb, MSK_putaccbj, MSK_putaccdoty, MSK_putacclist, MSK_putmaxnumacc

Problem data - affine expressions

- MSK_appendafes – Appends a number of empty affine expressions to the optimization task.
- MSK_putafebarfentry – Inputs one entry in barF.
- MSK_putafebarfentrylist – Inputs a list of entries in barF.
- MSK_putafebarfrow – Inputs a row of barF.
- MSK_putafefcol – Replaces all elements in one column of the F matrix in the affine expressions.

- *MSK_putafefentry* – Replaces one entry in F.
- *MSK_putafefentrylist* – Replaces a list of entries in F.
- *MSK_putafefrow* – Replaces all elements in one row of the F matrix in the affine expressions.
- *MSK_putafefrowlist* – Replaces all elements in a number of rows of the F matrix in the affine expressions.
- *MSK_putafeg* – Replaces one element in the g vector in the affine expressions.
- *MSK_putafegslice* – Modifies a slice of the vector g.
- *Infrequent:* *MSK_emptyafebarfrow*, *MSK_emptyafebarfrowlist*, *MSK_emptyafefcol*, *MSK_emptyafefcollist*, *MSK_emptyafefrow*, *MSK_emptyafefrowlist*, *MSK_getaccbarfblocktriplet*, *MSK_getafebarfblocktriplet*, *MSK_getafebarfnumrowentries*, *MSK_getafebarfrow*, *MSK_getafebarfrowinfo*, *MSK_getafefnumnz*, *MSK_getafefrow*, *MSK_getafefrownumnz*, *MSK_getafeftrip*, *MSK_getafeg*, *MSK_getafegslice*, *MSK_getnumafe*, *MSK_putafebarfblocktriplet*, *MSK_putafeglist*, *MSK_putmaxnumafe*

Problem data - bounds

- *MSK_putconbound* – Changes the bound for one constraint.
- *MSK_putconboundslice* – Changes the bounds for a slice of the constraints.
- *MSK_putvarbound* – Changes the bounds for one variable.
- *MSK_putvarboundslice* – Changes the bounds for a slice of the variables.
- *Infrequent:* *MSK_chgconbound*, *MSK_chgvarbound*, *MSK_getconbound*, *MSK_getconboundslice*, *MSK_getvarbound*, *MSK_getvarboundslice*, *MSK_inputdata*, *MSK_inputdata64*, *MSK_putconboundlist*, *MSK_putconboundlistconst*, *MSK_putconboundsliceconst*, *MSK_putvarboundlist*, *MSK_putvarboundlistconst*, *MSK_putvarboundsliceconst*

Problem data - cones (deprecated)

- *Deprecated:* ~~*MSK_appendcone*~~, ~~*MSK_appendconeseq*~~, ~~*MSK_appendconesseq*~~, ~~*MSK_generateconenames*~~, ~~*MSK_getcone*~~, ~~*MSK_getconeinfo*~~, ~~*MSK_getconename*~~, ~~*MSK_getconenameindex*~~, ~~*MSK_getconenamelen*~~, ~~*MSK_getmaxnumcone*~~, ~~*MSK_getnumcone*~~, ~~*MSK_getnumconemem*~~, ~~*MSK_putcone*~~, ~~*MSK_putconename*~~, ~~*MSK_putmaxnumcone*~~, ~~*MSK_removecones*~~

Problem data - constraints

- *MSK_appendcons* – Appends a number of constraints to the optimization task.
- *MSK_getnumcon* – Obtains the number of constraints.
- *MSK_putconbound* – Changes the bound for one constraint.
- *MSK_putconboundslice* – Changes the bounds for a slice of the constraints.
- *MSK_putconname* – Sets the name of a constraint.
- *MSK_removecons* – Removes a number of constraints.
- *Infrequent:* *MSK_chgconbound*, *MSK_generateconnames*, *MSK_getconbound*, *MSK_getconboundslice*, *MSK_getconname*, *MSK_getconnameindex*, *MSK_getconnamelen*, *MSK_getmaxnumcon*, *MSK_getnumqconknz*, *MSK_getnumqconknz64*, *MSK_getqconk*, *MSK_getqconk64*, *MSK_inputdata*, *MSK_inputdata64*, *MSK_putconboundlist*, *MSK_putconboundlistconst*, *MSK_putconboundsliceconst*, *MSK_putmaxnumcon*

Problem data - disjunctive constraints

- *MSK_appenddjcs* – Appends a number of empty disjunctive constraints to the task.
- *MSK_putdjc* – Inputs a disjunctive constraint.
- *MSK_putdjcname* – Sets the name of a disjunctive constraint.
- *MSK_putdjcslice* – Inputs a slice of disjunctive constraints.
- *Infrequent:* *MSK_getdjcafeidxlist*, *MSK_getdjcb*, *MSK_getdjcdomainidxlist*, *MSK_getdjcname*, *MSK_getdjcnamelen*, *MSK_getdjcnumafe*, *MSK_getdjcnumafetot*, *MSK_getdjcnumdomain*, *MSK_getdjcnumdomaintot*, *MSK_getdjcnumterm*, *MSK_getdjcnumtermtot*, *MSK_getdjcs*, *MSK_getdjctermsizelist*, *MSK_getnumdjc*, *MSK_putmaxnumdjc*

Problem data - domain

- *MSK_appenddualexpconedomain* – Appends the dual exponential cone domain.
- *MSK_appenddualgeomeanconedomain* – Appends the dual geometric mean cone domain.
- *MSK_appenddualpowerconedomain* – Appends the dual power cone domain.
- *MSK_appendprimalexpconedomain* – Appends the primal exponential cone domain.
- *MSK_appendprimalgeomeanconedomain* – Appends the primal geometric mean cone domain.
- *MSK_appendprimalpowerconedomain* – Appends the primal power cone domain.
- *MSK_appendquadraticconedomain* – Appends the n dimensional quadratic cone domain.
- *MSK_appendrdomain* – Appends the n dimensional real number domain.
- *MSK_appendrminusdomain* – Appends the n dimensional negative orthant to the list of domains.
- *MSK_appendrplusdomain* – Appends the n dimensional positive orthant to the list of domains.
- *MSK_appendrquadraticconedomain* – Appends the n dimensional rotated quadratic cone domain.
- *MSK_appendrzerodomain* – Appends the n dimensional 0 domain.
- *MSK_appendsvecpsdconedomain* – Appends the vectorized SVEC PSD cone domain.
- *MSK_putdomainname* – Sets the name of a domain.
- *Infrequent:* *MSK_getdomainn*, *MSK_getdomainname*, *MSK_getdomainnamelen*, *MSK_getdomaintype*, *MSK_getnumdomain*, *MSK_getpowerdomainalpha*, *MSK_getpowerdomaininfo*, *MSK_putmaxnumdomain*

Problem data - linear part

- *MSK_appendcons* – Appends a number of constraints to the optimization task.
- *MSK_appendvars* – Appends a number of variables to the optimization task.
- *MSK_getnumcon* – Obtains the number of constraints.
- *MSK_putacol* – Replaces all elements in one column of the linear constraint matrix.
- *MSK_putacolslice* – Replaces all elements in a sequence of columns the linear constraint matrix.
- *MSK_putacolslice64* – Replaces all elements in a sequence of columns the linear constraint matrix.
- *MSK_putaij* – Changes a single value in the linear coefficient matrix.
- *MSK_putaijlist* – Changes one or more coefficients in the linear constraint matrix.

- *MSK_putaijlist64* – Changes one or more coefficients in the linear constraint matrix.
- *MSK_putarow* – Replaces all elements in one row of the linear constraint matrix.
- *MSK_putarowslice* – Replaces all elements in several rows the linear constraint matrix.
- *MSK_putarowslice64* – Replaces all elements in several rows the linear constraint matrix.
- *MSK_putcfix* – Replaces the fixed term in the objective.
- *MSK_putcj* – Modifies one linear coefficient in the objective.
- *MSK_putconbound* – Changes the bound for one constraint.
- *MSK_putconboundslice* – Changes the bounds for a slice of the constraints.
- *MSK_putconname* – Sets the name of a constraint.
- *MSK_putcslice* – Modifies a slice of the linear objective coefficients.
- *MSK_putobjname* – Assigns a new name to the objective.
- *MSK_putobjsense* – Sets the objective sense.
- *MSK_putvarbound* – Changes the bounds for one variable.
- *MSK_putvarboundslice* – Changes the bounds for a slice of the variables.
- *MSK_putvarname* – Sets the name of a variable.
- *MSK_removecons* – Removes a number of constraints.
- *MSK_removevars* – Removes a number of variables.
- *Infrequent:* *MSK_chgconbound*, *MSK_chgvarbound*, *MSK_generatebarvarnames*, *MSK_generateconnames*, *MSK_generatevarnames*, *MSK_getacol*, *MSK_getacolnumnz*, *MSK_getacolslice*, *MSK_getacolslice64*, *MSK_getacolslicenumnz*, *MSK_getacolslicenumnz64*, *MSK_getacolslicetrip*, *MSK_getaij*, *MSK_getapiecenumnz*, *MSK_getarow*, *MSK_getarownumnz*, *MSK_getarowslice*, *MSK_getarowslice64*, *MSK_getarowslicenumnz*, *MSK_getarowslicenumnz64*, *MSK_getarowslicetrip*, *MSK_getatrip*, *MSK_getatruncatetol*, *MSK_getc*, *MSK_getcfix*, *MSK_getcj*, *MSK_getclist*, *MSK_getconbound*, *MSK_getconboundslice*, *MSK_getconname*, *MSK_getconnameindex*, *MSK_getconnamelen*, *MSK_getcslice*, *MSK_getmaxnumanz*, *MSK_getmaxnumanz64*, *MSK_getmaxnumcon*, *MSK_getmaxnumvar*, *MSK_getnumanz*, *MSK_getnumanz64*, *MSK_getobjsense*, *MSK_getvarbound*, *MSK_getvarboundslice*, *MSK_getvarname*, *MSK_getvarnameindex*, *MSK_getvarnamelen*, *MSK_inputdata*, *MSK_inputdata64*, *MSK_putacollist*, *MSK_putacollist64*, *MSK_putarowlist*, *MSK_putarowlist64*, *MSK_putatruncatetol*, *MSK_putclist*, *MSK_putconboundlist*, *MSK_putconboundlistconst*, *MSK_putconboundsliceconst*, *MSK_putmaxnumanz*, *MSK_putvarboundlist*, *MSK_putvarboundlistconst*, *MSK_putvarboundsliceconst*

Problem data - objective

- *MSK_putbarcj* – Changes one element in barc.
- *MSK_putcfix* – Replaces the fixed term in the objective.
- *MSK_putcj* – Modifies one linear coefficient in the objective.
- *MSK_putcslice* – Modifies a slice of the linear objective coefficients.
- *MSK_putobjname* – Assigns a new name to the objective.
- *MSK_putobjsense* – Sets the objective sense.
- *MSK_putqobj* – Replaces all quadratic terms in the objective.
- *MSK_putqobjij* – Replaces one coefficient in the quadratic term in the objective.
- *Infrequent:* *MSK_putclist*

Problem data - quadratic part

- *MSK_putqcon* – Replaces all quadratic terms in constraints.
- *MSK_putqconk* – Replaces all quadratic terms in a single constraint.
- *MSK_putqobj* – Replaces all quadratic terms in the objective.
- *MSK_putqobjij* – Replaces one coefficient in the quadratic term in the objective.
- *Infrequent:* *MSK_getmaxnumqnz*, *MSK_getmaxnumqnz64*, *MSK_getnumqconknz*, *MSK_getnumqconknz64*, *MSK_getnumqobjnz*, *MSK_getnumqobjnz64*, *MSK_getqconk*, *MSK_getqconk64*, *MSK_getqobj*, *MSK_getqobj64*, *MSK_getqobjij*, *MSK_putmaxnumqnz*
- *Deprecated:* ~~*MSK_toconic*~~

Problem data - semidefinite

- *MSK_appendbarvars* – Appends semidefinite variables to the problem.
- *MSK_appendsparsesymmat* – Appends a general sparse symmetric matrix to the storage of symmetric matrices.
- *MSK_appendsparsesymmatlist* – Appends a general sparse symmetric matrix to the storage of symmetric matrices.
- *MSK_putafebarfentry* – Inputs one entry in barF.
- *MSK_putafebarfentrylist* – Inputs a list of entries in barF.
- *MSK_putafebarfrow* – Inputs a row of barF.
- *MSK_putbaraij* – Inputs an element of barA.
- *MSK_putbaraijlist* – Inputs list of elements of barA.
- *MSK_putbararowlist* – Replace a set of rows of barA
- *MSK_putbarcj* – Changes one element in barc.
- *MSK_putbarvarname* – Sets the name of a semidefinite variable.
- *Infrequent:* *MSK_emptyafebarfrow*, *MSK_emptyafebarfrowlist*, *MSK_getaccbarfblocktriplet*, *MSK_getaccbarfnumblocktriplets*, *MSK_getafebarfblocktriplet*, *MSK_getafebarfnumblocktriplets*, *MSK_getafebarfnumrowentries*, *MSK_getafebarfrow*, *MSK_getafebarfrowinfo*, *MSK_getbarablocktriplet*, *MSK_getbaraidx*, *MSK_getbaraidxij*, *MSK_getbaraidxinfo*, *MSK_getbarasparsity*, *MSK_getbarcblocktriplet*, *MSK_getbarcidx*, *MSK_getbarcidxinfo*, *MSK_getbarcidxj*, *MSK_getbarcsparsity*, *MSK_getdimbarvarj*, *MSK_getlenbarvarj*, *MSK_getmaxnumbarvar*, *MSK_getnumbarablocktriplets*, *MSK_getnumbaranz*, *MSK_getnumbarcblocktriplets*, *MSK_getnumbarcnz*, *MSK_getnumbarvar*, *MSK_getnumsymmat*, *MSK_getsparsesymmat*, *MSK_getsymmatinfo*, *MSK_putafebarfblocktriplet*, *MSK_putbarablocktriplet*, *MSK_putbarcblocktriplet*, *MSK_putmaxnumbarvar*, *MSK_removebarvars*

Problem data - variables

- *MSK_appendvars* – Appends a number of variables to the optimization task.
- *MSK_getnumvar* – Obtains the number of variables.
- *MSK_putvarbound* – Changes the bounds for one variable.
- *MSK_putvarboundslice* – Changes the bounds for a slice of the variables.
- *MSK_putvarname* – Sets the name of a variable.

- *MSK_putvartype* – Sets the variable type of one variable.
- *MSK_removevars* – Removes a number of variables.
- *Infrequent:* *MSK_chgvarbound*, *MSK_generatebarvarnames*, *MSK_generatevarnames*, *MSK_getc*, *MSK_getcj*, *MSK_getmaxnumvar*, *MSK_getnumintvar*, *MSK_getvarbound*, *MSK_getvarboundslice*, *MSK_getvarname*, *MSK_getvarnameindex*, *MSK_getvarnamelen*, *MSK_getvartype*, *MSK_getvartypelist*, *MSK_putclist*, *MSK_putmaxnumvar*, *MSK_putvarboundlist*, *MSK_putvarboundlistconst*, *MSK_putvarboundsliceconst*, *MSK_putvartypelist*

Remote optimization

- *MSK_asyncgetresult* – Request a solution from a remote job.
- *MSK_asyncoptimize* – Offload the optimization task to a solver server in asynchronous mode.
- *MSK_asyncpoll* – Requests information about the status of the remote job.
- *MSK_asyncstop* – Request that the job identified by the token is terminated.
- *MSK_optimizermt* – Offload the optimization task to a solver server and wait for the solution.
- *MSK_putoptserverhost* – Specify an OptServer for remote calls.

Responses, errors and warnings

- *MSK_getcodedesc* – Obtains a short description of a response code.
- *Infrequent:* *MSK_getlasterror*, *MSK_getlasterror64*, *MSK_getresponseclass*

Sensitivity analysis

- *MSK_dualsensitivity* – Performs sensitivity analysis on objective coefficients.
- *MSK_primalsensitivity* – Perform sensitivity analysis on bounds.
- *MSK_sensitivityreport* – Creates a sensitivity report.

Solution - dual

- *MSK_getaccdoty* – Obtains the doty vector for an affine conic constraint.
- *MSK_getdualobj* – Computes the dual objective value associated with the solution.
- *MSK_gety* – Obtains the y vector for a solution.
- *MSK_getyslice* – Obtains a slice of the y vector for a solution.
- *Infrequent:* *MSK_getaccdotys*, *MSK_getreducedcosts*, *MSK_getslc*, *MSK_getslcslice*, *MSK_getslx*, *MSK_getslxslice*, *MSK_getsnx*, *MSK_getsnxslice*, *MSK_getsolution*, *MSK_getsolutionnew*, *MSK_getsolutionslice*, *MSK_getsuc*, *MSK_getsucslice*, *MSK_getsux*, *MSK_getsuxslice*, *MSK_putaccdoty*, *MSK_putconsolutioni*, *MSK_putslc*, *MSK_putslcslice*, *MSK_putslx*, *MSK_putslxslice*, *MSK_putsnx*, *MSK_putsnxslice*, *MSK_putsolution*, *MSK_putsolutionnew*, *MSK_putsolutionyi*, *MSK_putsuc*, *MSK_putsucslice*, *MSK_putsux*, *MSK_putsuxslice*, *MSK_putvarsolutionj*, *MSK_putyslice*

Solution - primal

- *MSK_getprimalobj* – Computes the primal objective value for the desired solution.
- *MSK_getxx* – Obtains the xx vector for a solution.
- *MSK_getxxslice* – Obtains a slice of the xx vector for a solution.
- *MSK_putxx* – Sets the xx vector for a solution.
- *MSK_putxxslice* – Sets a slice of the xx vector for a solution.
- *Infrequent:* *MSK_evaluateacc*, *MSK_evaluateaccs*, *MSK_getsolution*, *MSK_getsolutionnew*, *MSK_getsolutionslice*, *MSK_getxc*, *MSK_getxcslice*, *MSK_putconsolutioni*, *MSK_putsolution*, *MSK_putsolutionnew*, *MSK_putvarsolutionj*, *MSK_putxc*, *MSK_putxcslice*, *MSK_puty*

Solution - semidefinite

- *MSK_getbarsj* – Obtains the dual solution for a semidefinite variable.
- *MSK_getbarsslice* – Obtains the dual solution for a sequence of semidefinite variables.
- *MSK_getbarxj* – Obtains the primal solution for a semidefinite variable.
- *MSK_getbarxslice* – Obtains the primal solution for a sequence of semidefinite variables.
- *Infrequent:* *MSK_putbarsj*, *MSK_putbarxj*

Solution information

- *MSK_getdualobj* – Computes the dual objective value associated with the solution.
- *MSK_getprimalobj* – Computes the primal objective value for the desired solution.
- *MSK_getprosta* – Obtains the problem status.
- *MSK_getpviolcon* – Computes the violation of a primal solution associated to a constraint.
- *MSK_getpviolvar* – Computes the violation of a primal solution for a list of scalar variables.
- *MSK_getsolsta* – Obtains the solution status.
- *MSK_getsolutioninfo* – Obtains information about of a solution.
- *MSK_getsolutioninfonew* – Obtains information about of a solution.
- *MSK_onesolutionsummary* – Prints a short summary of a specified solution.
- *MSK_solutiondef* – Checks whether a solution is defined.
- *MSK_solutionsummary* – Prints a short summary of the current solutions.
- *Infrequent:* *MSK_analyzesolution*, *MSK_deletesolution*, *MSK_getdualsolutionnorms*, *MSK_getdviolacc*, *MSK_getdviolbarvar*, *MSK_getdviolcon*, *MSK_getdviolvar*, *MSK_getprimalsolutionnorms*, *MSK_getpviolacc*, *MSK_getpviolbarvar*, *MSK_getpvioldjc*, *MSK_getskc*, *MSK_getskcslice*, *MSK_getskn*, *MSK_getskx*, *MSK_getskxslice*, *MSK_getsolution*, *MSK_getsolutionnew*, *MSK_getsolutionslice*, *MSK_prostatostr*, *MSK_putconsolutioni*, *MSK_putskc*, *MSK_putskcslice*, *MSK_putskx*, *MSK_putskxslice*, *MSK_putsolution*, *MSK_putsolutionnew*, *MSK_putsolutionyi*, *MSK_putvarsolutionj*
- *Deprecated:* ~~*MSK_getdviolcones*~~, ~~*MSK_getpviolcones*~~

Solving systems with basis matrix

- *Infrequent:* *MSK_basiscond*, *MSK_initbasissolve*, *MSK_solvewithbasis*

System, memory and debugging

- *Infrequent:* *MSK_callocdbgenv*, *MSK_callocdbgtask*, *MSK_callocenv*, *MSK_calloctask*, *MSK_checkmemenv*, *MSK_checkmemtask*, *MSK_freedbgenv*, *MSK_freedbgtask*, *MSK_freeenv*, *MSK_freetask*, *MSK_getmemusagetask*, *MSK_utf8towchar*, *MSK_wchartoutf8*

Versions

- *MSK_getversion* – Obtains MOSEK version information.
- *Infrequent:* *MSK_checkversion*, *MSK_getbuildinfo*

Other

- *Infrequent:* *MSK_isinfinity*

15.3 Functions in alphabetical order

`MSK_analyzenames`

```
MSKrescodee (MSKAPI MSK_analyzenames) (
  MSKtask_t task,
  MSKstreamtypee whichstream,
  MSKnametypee nametype)
```

The function analyzes the names and issues an error if a name is invalid.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichstream` (*MSKstreamtypee*) – Index of the stream. (input)
- `nametype` (*MSKnametypee*) – The type of names e.g. valid in MPS or LP files. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

`MSK_analyzeproblem`

```
MSKrescodee (MSKAPI MSK_analyzeproblem) (
  MSKtask_t task,
  MSKstreamtypee whichstream)
```

The function analyzes the data of a task and writes out a report.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichstream` (*MSKstreamtypee*) – Index of the stream. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*

`MSK_analyzesolution`

```
MSKrescodee (MSKAPI MSK_analyzesolution) (
  MSKtask_t task,
  MSKstreamtypee whichstream,
  MSKsoltypee whichsol)
```

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_t*) – An optimization task. (input)
- `whichstream` (*MSKstreamtypee*) – Index of the stream. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Inspecting the task*

`MSK_appendacc`

```
MSKrescodee (MSKAPI MSK_appendacc) (
  MSKtask_t task,
  MSKint64t domidx,
  MSKint64t numafeidx,
  const MSKint64t * afeidxlist,
  const MSKrealt * b)
```

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:

$$F_{\mathrm{afeidxlist}[k],:}x + g_{\mathrm{afeidxlist}[k]} \quad \text{for } k = 0, \ldots, n-1.$$

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:

$$F_{\mathrm{afeidxlist}[k],:}x + g_{\mathrm{afeidxlist}[k]} - b_k \quad \text{for } k = 0, \ldots, n-1.$$

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `domidx` (*MSKint64t*) – Domain index. (input)
- `numafeidx` (*MSKint64t*) – Number of affine expressions in the affine expression list (must equal the dimension of the domain). (input)
- `afeidxlist` (*MSKint64t**) – List of affine expression indexes. (input)
- `b` (*MSKrealt**) – The vector of constant terms added to affine expressions. Optional, can be NULL. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Problem data - affine conic constraints*

MSK_appendaccs

```
MSKrescodee (MSKAPI MSK_appendaccs) (
  MSKtask_t task,
  MSKint64t numaccs,
  const MSKint64t * domidxs,
  MSKint64t numafeidx,
  const MSKint64t * afeidxlist,
  const MSKrealt * b)
```

Appends `numaccs` affine conic constraint to the task. Each single affine conic constraint should be specified as in *MSK_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_t*) – An optimization task. (input)
- `numaccs` (*MSKint64t*) – The number of affine conic constraints to append. (input)
- `domidxs` (*MSKint64t* *) – Domain indices. (input)
- `numafeidx` (*MSKint64t*) – Number of affine expressions in the affine expression list (must equal the sum of dimensions of the domains). (input)
- `afeidxlist` (*MSKint64t* *) – List of affine expression indexes. (input)
- `b` (*MSKrealt* *) – The vector of constant terms added to affine expressions. Optional, can be NULL. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Problem data - affine conic constraints*

MSK_appendaccseq

```
MSKrescodee (MSKAPI MSK_appendaccseq) (
  MSKtask_t task,
  MSKint64t domidx,
  MSKint64t numafeidx,
  MSKint64t afeidxfirst,
  const MSKrealt * b)
```

Appends an affine conic constraint to the task, as in *MSK_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:

$$F_{\mathrm{afeidxfirst}+k,:}x + g_{\mathrm{afeidxfirst}+k} \quad \text{for } k = 0, \ldots, n-1.$$

If an optional vector `b` of length `numafeidx` is specified then the expressions appearing in the affine constraint will instead be taken as

$$F_{\mathrm{afeidxfirst}+k,:}x + g_{\mathrm{afeidxfirst}+k} - b_k \quad \text{for } k = 0, \ldots, n-1.$$

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- domidx (*MSKint64t*) – Domain index. (input)
- numafeidx (*MSKint64t*) – Number of affine expressions in the affine expression list (must equal the dimension of the domain). (input)
- afeidxfirst (*MSKint64t*) – Index of the first affine expression. (input)
- b (*MSKrealt* *) – The vector of constant terms added to affine expressions. Optional, can be NULL. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*

MSK_appendaccsseq

```
MSKrescodee (MSKAPI MSK_appendaccsseq) (
  MSKtask_t task,
  MSKint64t numaccs,
  const MSKint64t * domidxs,
  MSKint64t numafeidx,
  MSKint64t afeidxfirst,
  const MSKrealt * b)
```

Appends numaccs affine conic constraint to the task. It is the block variant of *MSK_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_t*) – An optimization task. (input)
- numaccs (*MSKint64t*) – The number of affine conic constraints to append. (input)
- domidxs (*MSKint64t* *) – Domain indices. (input)
- numafeidx (*MSKint64t*) – Number of affine expressions in the affine expression list (must equal the sum of dimensions of the domains). (input)
- afeidxfirst (*MSKint64t*) – Index of the first affine expression. (input)
- b (*MSKrealt* *) – The vector of constant terms added to affine expressions. Optional, can be NULL. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*

MSK_appendafes

```
MSKrescodee (MSKAPI MSK_appendafes) (
  MSKtask_t task,
  MSKint64t num)
```

Appends a number of empty affine expressions to the task.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint64t*) – Number of empty affine expressions which should be appended. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

MSK_appendbarvars

```
MSKrescodee (MSKAPI MSK_appendbarvars) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * dim)
```

Appends positive semidefinite matrix variables of dimensions given by `dim` to the problem.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of symmetric matrix variables to be appended. (input)
- `dim` (*MSKint32t**) – Dimensions of symmetric matrix variables to be added. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*

~~`MSK_appendcone`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_appendcone) (
  MSKtask_t task,
  MSKconetypee ct,
  MSKrealt conepar,
  MSKint32t nummem,
  const MSKint32t * submem)
```

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

$$\hat{x} = x_{\mathtt{submem}[0]}, \ldots, x_{\mathtt{submem}[\mathtt{nummem}-1]}.$$

Depending on the value of `ct` this function appends one of the constraints:

- Quadratic cone (*MSK_CT_QUAD*, requires `nummem` ≥ 1):

$$\hat{x}_0 \geq \sqrt{\sum_{i=1}^{i<\mathtt{nummem}} \hat{x}_i^2}$$

- Rotated quadratic cone (*MSK_CT_RQUAD*, requires `nummem` ≥ 2):

$$2\hat{x}_0\hat{x}_1 \geq \sum_{i=2}^{i<\mathtt{nummem}} \hat{x}_i^2, \quad \hat{x}_0, \hat{x}_1 \geq 0$$

- Primal exponential cone (*MSK_CT_PEXP*, requires `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 nummem ≥ 2):

$$\hat{x}_0^{\alpha}\hat{x}_1^{1-\alpha} \geq \sqrt{\sum_{i=2}^{i<\mathtt{nummem}} \hat{x}_i^2}, \quad \hat{x}_0, \hat{x}_1 \geq 0$$

  where α is the cone parameter specified by conepar.
- Dual exponential cone (*MSK_CT_DEXP*, requires 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 nummem ≥ 2):

$$\left(\frac{\hat{x}_0}{\alpha}\right)^{\alpha}\left(\frac{\hat{x}_1}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{i=2}^{i<\mathtt{nummem}} \hat{x}_i^2}, \quad \hat{x}_0, \hat{x}_1 \geq 0$$

  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_t*) – An optimization task. (input)
- ct (*MSKconetypee*) – Specifies the type of the cone. (input)
- conepar (*MSKrealt*) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0. (input)
- nummem (*MSKint32t*) – Number of member variables in the cone. (input)
- submem (*MSKint32t* *) – Variable subscripts of the members in the cone. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - cones (deprecated)*

~~MSK_appendconeseq~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_appendconeseq) (
  MSKtask_t task,
  MSKconetypee ct,
  MSKrealt conepar,
  MSKint32t nummem,
  MSKint32t j)
```

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 *~~MSK_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_t*) – An optimization task. (input)
- ct (*MSKconetypee*) – Specifies the type of the cone. (input)
- conepar (*MSKrealt*) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0. (input)
- nummem (*MSKint32t*) – Number of member variables in the cone. (input)
- j (*MSKint32t*) – Index of the first variable in the conic constraint. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - cones (deprecated)*

~~MSK_appendconesseq~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_appendconesseq) (
  MSKtask_t task,
  MSKint32t num,
  const MSKconetypee * ct,
  const MSKrealt * conepar,
  const MSKint32t * nummem,
  MSKint32t j)
```

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 ~~*MSK_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_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of cones to be added. (input)
- `ct` (*MSKconetypee**) – Specifies the type of the cone. (input)
- `conepar` (*MSKrealt**) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0. (input)
- `nummem` (*MSKint32t**) – Numbers of member variables in the cones. (input)
- `j` (*MSKint32t*) – Index of the first variable in the first cone to be appended. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - cones (deprecated)*

`MSK_appendcons`

```
MSKrescodee (MSKAPI MSK_appendcons) (
  MSKtask_t task,
  MSKint32t num)
```

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_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of constraints which should be appended. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - constraints*

`MSK_appenddjcs`

```
MSKrescodee (MSKAPI MSK_appenddjcs) (
  MSKtask_t task,
  MSKint64t num)
```

Appends a number of empty disjunctive constraints to the task.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint64t*) – Number of empty disjunctive constraints which should be appended. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*

MSK_appenddualexpconedomain

```
MSKrescodee (MSKAPI MSK_appenddualexpconedomain) (
  MSKtask_t task,
  MSKint64t * domidx)
```

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_t*) – An optimization task. (input)
- domidx (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

MSK_appenddualgeomeanconedomain

```
MSKrescodee (MSKAPI MSK_appenddualgeomeanconedomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t * domidx)
```

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_t*) – An optimization task. (input)
- n (*MSKint64t*) – Dimmension of the domain. (input)
- domidx (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

MSK_appenddualpowerconedomain

```
MSKrescodee (MSKAPI MSK_appenddualpowerconedomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t nleft,
  const MSKrealt * alpha,
  MSKint64t * domidx)
```

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_t*) – An optimization task. (input)
- n (*MSKint64t*) – Dimension of the domain. (input)
- nleft (*MSKint64t*) – Number of variables on the left hand side. (input)
- alpha (*MSKrealt* *) – The sequence proportional to exponents. Must be positive. (input)
- domidx (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

MSK_appendprimalexpconedomain

```
MSKrescodee (MSKAPI MSK_appendprimalexpconedomain) (
  MSKtask_t task,
  MSKint64t * domidx)
```

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_t*) – An optimization task. (input)
- domidx (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

MSK_appendprimalgeomeanconedomain

```
MSKrescodee (MSKAPI MSK_appendprimalgeomeanconedomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t * domidx)
```

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_t*) – An optimization task. (input)
- n (*MSKint64t*) – Dimmension of the domain. (input)
- domidx (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

MSK_appendprimalpowerconedomain

```
MSKrescodee (MSKAPI MSK_appendprimalpowerconedomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t nleft,
  const MSKrealt * alpha,
  MSKint64t * domidx)
```

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_t*) – An optimization task. (input)
- n (*MSKint64t*) – Dimension of the domain. (input)
- nleft (*MSKint64t*) – Number of variables on the left hand side. (input)
- alpha (*MSKrealt* *) – The sequence proportional to exponents. Must be positive. (input)
- domidx (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

MSK_appendquadraticconedomain

```
MSKrescodee (MSKAPI MSK_appendquadraticconedomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t * domidx)
```

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_t*) – An optimization task. (input)
- n (*MSKint64t*) – Dimmension of the domain. (input)
- domidx (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

MSK_appendrdomain

```
MSKrescodee (MSKAPI MSK_appendrdomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t * domidx)
```

Appends the n-dimensional real space $\{x \in \mathbb{R}^n\}$ to the list of domains.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- n (*MSKint64t*) – Dimmension of the domain. (input)
- domidx (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

MSK_appendrminusdomain

```
MSKrescodee (MSKAPI MSK_appendrminusdomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t * domidx)
```

Appends the n-dimensional negative orthant $\{x \in \mathbb{R}^n : x \leq 0\}$ to the list of domains.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `n` (*MSKint64t*) – Dimmension of the domain. (input)
- `domidx` (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

`MSK_appendrplusdomain`

```
MSKrescodee (MSKAPI MSK_appendrplusdomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t * domidx)
```

Appends the n-dimensional positive orthant $\{x \in \mathbb{R}^n : x \geq 0\}$ to the list of domains.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `n` (*MSKint64t*) – Dimmension of the domain. (input)
- `domidx` (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

`MSK_appendrquadraticconedomain`

```
MSKrescodee (MSKAPI MSK_appendrquadraticconedomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t * domidx)
```

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_t*) – An optimization task. (input)
- `n` (*MSKint64t*) – Dimmension of the domain. (input)
- `domidx` (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

`MSK_appendrzerodomain`

```
MSKrescodee (MSKAPI MSK_appendrzerodomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t * domidx)
```

Appends the zero in n-dimensional real space $\{x \in \mathbb{R}^n : x = 0\}$ to the list of domains.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `n` (*MSKint64t*) – Dimmension of the domain. (input)
- `domidx` (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

MSK_appendsparsesymmat

```
MSKrescodee (MSKAPI MSK_appendsparsesymmat) (
  MSKtask_t task,
  MSKint32t dim,
  MSKint64t nz,
  const MSKint32t * subi,
  const MSKint32t * subj,
  const MSKrealt * valij,
  MSKint64t * idx)
```

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_t*) – An optimization task. (input)
- `dim` (*MSKint32t*) – Dimension of the symmetric matrix that is appended. (input)
- `nz` (*MSKint64t*) – Number of triplets. (input)
- `subi` (*MSKint32t* *) – Row subscript in the triplets. (input)
- `subj` (*MSKint32t* *) – Column subscripts in the triplets. (input)
- `valij` (*MSKrealt* *) – Values of each triplet. (input)
- `idx` (*MSKint64t by reference*) – Unique index assigned to the inputted matrix that can be used for later reference. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*

MSK_appendsparsesymmatlist

```
MSKrescodee (MSKAPI MSK_appendsparsesymmatlist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * dims,
  const MSKint64t * nz,
  const MSKint32t * subi,
  const MSKint32t * subj,
  const MSKrealt * valij,
  MSKint64t * idx)
```

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_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of matrixes to append. (input)
- `dims` (*MSKint32t* *) – Dimensions of the symmetric matrixes. (input)
- `nz` (*MSKint64t* *) – Number of nonzeros for each matrix. (input)
- `subi` (*MSKint32t* *) – Row subscript in the triplets. (input)
- `subj` (*MSKint32t* *) – Column subscripts in the triplets. (input)
- `valij` (*MSKrealt* *) – Values of each triplet. (input)
- `idx` (*MSKint64t* *) – Unique index assigned to the inputted matrix that can be used for later reference. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*

`MSK_appendsvecpsdconedomain`

```
MSKrescodee (MSKAPI MSK_appendsvecpsdconedomain) (
  MSKtask_t task,
  MSKint64t n,
  MSKint64t * domidx)
```

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_t*) – An optimization task. (input)
- `n` (*MSKint64t*) – Dimension of the domain, must be of the form $d(d+1)/2$. (input)
- `domidx` (*MSKint64t by reference*) – Index of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*

`MSK_appendvars`

```
MSKrescodee (MSKAPI MSK_appendvars) (
  MSKtask_t task,
  MSKint32t num)
```

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_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of variables which should be appended. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - variables*

`MSK_asyncgetresult`

```
MSKrescodee (MSKAPI MSK_asyncgetresult) (
  MSKtask_t task,
  const char * address,
  const char * accesstoken,
  const char * token,
  MSKbooleant * respavailable,
  MSKrescodee * resp,
  MSKrescodee * trm)
```

Request a solution from a remote job identified by the argument `token`. For other arguments see *MSK_asyncoptimize*. If the solution is available it will be retrieved and loaded into the local task.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `address` (`char*`) – Address of the OptServer. (input)
- `accesstoken` (`char*`) – Access token. (input)
- `token` (`char*`) – The task token. (input)
- `respavailable` (*MSKbooleant by reference*) – Indicates if a remote response is available. If this is not true, `resp` and `trm` should be ignored. (output)
- `resp` (*MSKrescodee by reference*) – Is the response code from the remote solver. (output)
- `trm` (*MSKrescodee by reference*) – Is either *MSK_RES_OK* or a termination response code. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Remote optimization*

`MSK_asyncoptimize`

```
MSKrescodee (MSKAPI MSK_asyncoptimize) (
  MSKtask_t task,
  const char * address,
  const char * accesstoken,
  char * token)
```

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_t*) – An optimization task. (input)

- `address` (`char*`) – Address of the OptServer. (input)
- `accesstoken` (`char*`) – Access token. (input)
- `token` (`char*`) – Returns the task token. (output)

Return (*`MSKrescodee`*) – The function response code.

Groups *Remote optimization*

`MSK_asyncpoll`

```
MSKrescodee (MSKAPI MSK_asyncpoll) (
  MSKtask_t task,
  const char * address,
  const char * accesstoken,
  const char * token,
  MSKbooleant * respavailable,
  MSKrescodee * resp,
  MSKrescodee * trm)
```

Requests information about the status of the remote job identified by the argument `token`. For other arguments see *`MSK_asyncoptimize`*.

Parameters

- `task` (*`MSKtask_t`*) – An optimization task. (input)
- `address` (`char*`) – Address of the OptServer. (input)
- `accesstoken` (`char*`) – Access token. (input)
- `token` (`char*`) – The task token. (input)
- `respavailable` (*`MSKbooleant` by reference*) – Indicates if a remote response is available. If this is not true, `resp` and `trm` should be ignored. (output)
- `resp` (*`MSKrescodee` by reference*) – Is the response code from the remote solver. (output)
- `trm` (*`MSKrescodee` by reference*) – Is either *`MSK_RES_OK`* or a termination response code. (output)

Return (*`MSKrescodee`*) – The function response code.

Groups *Remote optimization*

`MSK_asyncstop`

```
MSKrescodee (MSKAPI MSK_asyncstop) (
  MSKtask_t task,
  const char * address,
  const char * accesstoken,
  const char * token)
```

Request that the remote job identified by `token` is terminated. For other arguments see *`MSK_asyncoptimize`*.

Parameters

- `task` (*`MSKtask_t`*) – An optimization task. (input)
- `address` (`char*`) – Address of the OptServer. (input)
- `accesstoken` (`char*`) – Access token. (input)
- `token` (`char*`) – The task token. (input)

Return (*`MSKrescodee`*) – The function response code.

Groups *Remote optimization*

`MSK_axpy`

```
MSKrescodee (MSKAPI MSK_axpy) (
  MSKenv_t env,
  MSKint32t n,
  MSKrealt alpha,
  const MSKrealt * x,
  MSKrealt * y)
```

Adds αx to y, i.e. performs the update

$$y := \alpha x + y.$$

Note that the result is stored overwriting y. It must not overlap with the other input arrays.

Parameters

- env (*MSKenv_t*) – The MOSEK environment. (input)
- n (*MSKint32t*) – Length of the vectors. (input)
- alpha (*MSKrealt*) – The scalar that multiplies x. (input)
- x (*MSKrealt**) – The x vector. (input)
- y (*MSKrealt**) – The y vector. (input/output)

Return (*MSKrescodee*) – The function response code.

Groups *Linear algebra*

MSK_basiscond

```
MSKrescodee (MSKAPI MSK_basiscond) (
  MSKtask_t task,
  MSKrealt * nrmbasis,
  MSKrealt * nrminvbasis)
```

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_t*) – An optimization task. (input)
- nrmbasis (*MSKrealt by reference*) – An estimate for the 1-norm of the basis. (output)
- nrminvbasis (*MSKrealt by reference*) – An estimate for the 1-norm of the inverse of the basis. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solving systems with basis matrix*

MSK_bktostr

```
MSKrescodee (MSKAPI MSK_bktostr) (
  MSKtask_t task,
  MSKboundkeye bk,
  char * str)
```

Obtains an identifier string corresponding to a bound key.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- bk (*MSKboundkeye*) – Bound key. (input)
- str (char*) – String corresponding to the bound key code bk. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

MSK_callbackcodetostr

```
MSKrescodee (MSKAPI MSK_callbackcodetostr) (
  MSKcallbackcodee code,
  char * callbackcodestr)
```

Obtains the string representation of a callback code.

Parameters

- code (*MSKcallbackcodee*) – A callback code. (input)
- callbackcodestr (char*) – String corresponding to the callback code. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

MSK_callocdbgenv

```
void * (MSKAPI MSK_callocdbgenv) (
  MSKenv_t env,
  const size_t number,
  const size_t size,
  const char * file,
  const unsigned line)
```

Debug version of *MSK_callocenv*.

Parameters

- env (*MSKenv_t*) – The MOSEK environment. (input)
- number (size_t) – Number of elements. (input)
- size (size_t) – Size of each individual element. (input)
- file (char*) – File from which the function is called. (input)
- line (unsigned) – Line in the file from which the function is called. (input)

Return (void*) – A pointer to the memory allocated through the environment.

Groups *System, memory and debugging*

MSK_callocdbgtask

```
void * (MSKAPI MSK_callocdbgtask) (
  MSKtask_t task,
  const size_t number,
  const size_t size,
  const char * file,
  const unsigned line)
```

Debug version of *MSK_calloctask*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- number (size_t) – Number of elements. (input)

- `size` (`size_t`) – Size of each individual element. (input)
- `file` (`char*`) – File from which the function is called. (input)
- `line` (`unsigned`) – Line in the file from which the function is called. (input)

Return (`void*`) – A pointer to the memory allocated through the task.
Groups *System, memory and debugging*

`MSK_callocenv`

```
void * (MSKAPI MSK_callocenv) (
  MSKenv_t env,
  const size_t number,
  const size_t size)
```

Equivalent to `calloc` i.e. allocate space for an array of length `number` where each element is of size `size`.

Parameters
- `env` (*`MSKenv_t`*) – The MOSEK environment. (input)
- `number` (`size_t`) – Number of elements. (input)
- `size` (`size_t`) – Size of each individual element. (input)

Return (`void*`) – A pointer to the memory allocated through the environment.
Groups *System, memory and debugging*

`MSK_calloctask`

```
void * (MSKAPI MSK_calloctask) (
  MSKtask_t task,
  const size_t number,
  const size_t size)
```

Equivalent to `calloc` i.e. allocate space for an array of length `number` where each element is of size `size`.

Parameters
- `task` (*`MSKtask_t`*) – An optimization task. (input)
- `number` (`size_t`) – Number of elements. (input)
- `size` (`size_t`) – Size of each individual element. (input)

Return (`void*`) – A pointer to the memory allocated through the task.
Groups *System, memory and debugging*

`MSK_checkinall`

```
MSKrescodee (MSKAPI MSK_checkinall) (
  MSKenv_t env)
```

Check in all unused license features to the license token server.

Parameters `env` (*`MSKenv_t`*) – The MOSEK environment. (input)
Return (*`MSKrescodee`*) – The function response code.
Groups *License system*

`MSK_checkinlicense`

```
MSKrescodee (MSKAPI MSK_checkinlicense) (
  MSKenv_t env,
  MSKfeaturee 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 *MSK_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_t*) – The MOSEK environment. (input)
- feature (*MSKfeaturee*) – Feature to check in to the license system. (input)

Return (*MSKrescodee*) – The function response code.

Groups *License system*

MSK_checkmemenv

```
MSKrescodee (MSKAPI MSK_checkmemenv) (
  MSKenv_t env,
  const char * file,
  MSKint32t line)
```

Checks the memory allocated by the environment.

Parameters

- env (*MSKenv_t*) – The MOSEK environment. (input)
- file (char*) – File from which the function is called. (input)
- line (*MSKint32t*) – Line in the file from which the function is called. (input)

Return (*MSKrescodee*) – The function response code.

Groups *System, memory and debugging*

MSK_checkmemtask

```
MSKrescodee (MSKAPI MSK_checkmemtask) (
  MSKtask_t task,
  const char * file,
  MSKint32t line)
```

Checks the memory allocated by the task.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- file (char*) – File from which the function is called. (input)
- line (*MSKint32t*) – Line in the file from which the function is called. (input)

Return (*MSKrescodee*) – The function response code.

Groups *System, memory and debugging*

MSK_checkoutlicense

```
MSKrescodee (MSKAPI MSK_checkoutlicense) (
  MSKenv_t env,
  MSKfeaturee 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 *MSK_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 *MSK_checkinlicense* is called.

If a given feature is already checked out when this function is called, the call has no effect.

Parameters
- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `feature` (*MSKfeaturee*) – Feature to check out from the license system. (input)

Return (*MSKrescodee*) – The function response code.

Groups *License system*

`MSK_checkversion`

```
MSKrescodee (MSKAPI MSK_checkversion) (
  MSKenv_t env,
  MSKint32t major,
  MSKint32t minor,
  MSKint32t revision)
```

Compares the version of the **MOSEK** DLL with a specified version. Returns *MSK_RES_OK* if the versions match and one of *MSK_RES_ERR_NEWER_DLL*, *MSK_RES_ERR_OLDER_DLL* otherwise.

Parameters
- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `major` (*MSKint32t*) – Major version number. (input)
- `minor` (*MSKint32t*) – Minor version number. (input)
- `revision` (*MSKint32t*) – Revision number. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Versions*

`MSK_chgconbound`

```
MSKrescodee (MSKAPI MSK_chgconbound) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t lower,
  MSKint32t finite,
  MSKrealt value)
```

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_t*) – An optimization task. (input)
- `i` (*MSKint32t*) – Index of the constraint for which the bounds should be changed. (input)
- `lower` (*MSKint32t*) – If non-zero, then the lower bound is changed, otherwise the upper bound is changed. (input)

- `finite` (*MSKint32t*) – If non-zero, then `value` is assumed to be finite. (input)
- `value` (*MSKrealt*) – New value for the bound. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - bounds*, *Problem data - constraints*, *Problem data - linear part*

`MSK_chgvarbound`

```
MSKrescodee (MSKAPI MSK_chgvarbound) (
  MSKtask_t task,
  MSKint32t j,
  MSKint32t lower,
  MSKint32t finite,
  MSKrealt value)
```

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_t*) – An optimization task. (input)
- `j` (*MSKint32t*) – Index of the variable for which the bounds should be changed. (input)
- `lower` (*MSKint32t*) – If non-zero, then the lower bound is changed, otherwise the upper bound is changed. (input)
- `finite` (*MSKint32t*) – If non-zero, then `value` is assumed to be finite. (input)
- `value` (*MSKrealt*) – New value for the bound. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - bounds*, *Problem data - variables*, *Problem data - linear part*

`MSK_clonetask`

```
MSKrescodee (MSKAPI MSK_clonetask) (
  MSKtask_t task,
  MSKtask_t * clonedtask)
```

Creates a clone of an existing task copying all problem data and parameter settings to a new task. Callback functions are not copied.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `clonedtask` (*MSKtask_t by reference*) – The cloned task. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*

`MSK_commitchanges`

```
MSKrescodee (MSKAPI MSK_commitchanges) (
  MSKtask_t task)
```

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_t*) – An optimization task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*

MSK_computesparsecholesky

```
MSKrescodee (MSKAPI MSK_computesparsecholesky) (
  MSKenv_t env,
  MSKint32t numthreads,
  MSKint32t ordermethod,
  MSKrealt tolsingular,
  MSKint32t n,
  const MSKint32t * anzc,
  const MSKint64t * aptrc,
  const MSKint32t * asubc,
  const MSKrealt * avalc,
  MSKint32t ** perm,
  MSKrealt ** diag,
  MSKint32t ** lnzc,
  MSKint64t ** lptrc,
  MSKint64t * lensubnval,
  MSKint32t ** lsubc,
  MSKrealt ** lvalc)
```

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.

The function allocates memory for the output arrays. It must be freed by the user with *MSK_freeenv*.

Parameters

- env (*MSKenv_t*) – The MOSEK environment. (input)

- numthreads (*MSKint32t*) – 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 (*MSKint32t*) – If nonzero, then a sparsity preserving ordering will be employed. (input)
- tolsingular (*MSKrealt*) – A positive parameter controlling when a pivot is declared zero. (input)
- n (*MSKint32t*) – Specifies the order of A. (input)
- anzc (*MSKint32t* *) – anzc[j] is the number of nonzeros in the j-th column of A. (input)
- aptrc (*MSKint64t* *) – aptrc[j] is a pointer to the first element in column j of A. (input)
- asubc (*MSKint32t* *) – Row indexes for each column stored in increasing order. (input)
- avalc (*MSKrealt* *) – The value corresponding to row indexed stored in asubc. (input)
- perm (*MSKint32t* * *by reference*) – Permutation array used to specify the permutation matrix P computed by the function. (output)
- diag (*MSKrealt* * *by reference*) – The diagonal elements of matrix D. (output)
- lnzc (*MSKint32t* * *by reference*) – lnzc[j] is the number of non zero elements in column j of L. (output)
- lptrc (*MSKint64t* * *by reference*) – lptrc[j] is a pointer to the first row index and value in column j of L. (output)
- lensubnval (*MSKint64t by reference*) – Number of elements in lsubc and lvalc. (output)
- lsubc (*MSKint32t* * *by reference*) – Row indexes for each column stored in increasing order. (output)
- lvalc (*MSKrealt* * *by reference*) – The values corresponding to row indexed stored in lsubc. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Linear algebra*

~~MSK_conetypetostr~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_conetypetostr) (
  MSKtask_t task,
  MSKconetypee ct,
  char * str)
```

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_t*) – An optimization task. (input)
- ct (*MSKconetypee*) – Specifies the type of the cone. (input)
- str (char*) – String corresponding to the cone type code ct. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

MSK_deleteenv

```
MSKrescodee (MSKAPI MSK_deleteenv) (
  MSKenv_t * env)
```

Deletes a **MOSEK** environment and all the data associated with it.

Before calling this function it is a good idea to call the function *MSK_unlinkfuncfromenvstream* for each stream that has had a function linked to it.

Parameters env (*MSKenv_t by reference*) – The MOSEK environment. (input/output)
Return (*MSKrescodee*) – The function response code.
Groups *Environment and task management*

MSK_deletesolution

```
MSKrescodee (MSKAPI MSK_deletesolution) (
  MSKtask_t task,
  MSKsoltypee whichsol)
```

Undefine a solution and free the memory it uses.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Environment and task management*, *Solution information*

MSK_deletetask

```
MSKrescodee (MSKAPI MSK_deletetask) (
  MSKtask_t * task)
```

Deletes a task.

Parameters task (*MSKtask_t by reference*) – An optimization task. (input/output)
Return (*MSKrescodee*) – The function response code.
Groups *Environment and task management*

MSK_dot

```
MSKrescodee (MSKAPI MSK_dot) (
  MSKenv_t env,
  MSKint32t n,
  const MSKrealt * x,
  const MSKrealt * y,
  MSKrealt * xty)
```

Computes the inner product of two vectors x, y of length $n \geq 0$, i.e

$$x \cdot y = \sum_{i=1}^{n} x_i y_i.$$

Note that if $n = 0$, then the result of the operation is 0.

Parameters
- env (*MSKenv_t*) – The MOSEK environment. (input)
- n (*MSKint32t*) – Length of the vectors. (input)
- x (*MSKrealt**) – The x vector. (input)

- y (*MSKrealt* *) – The y vector. (input)
- xty (*MSKrealt by reference*) – The result of the inner product between x and y. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Linear algebra*

MSK_dualsensitivity

```
MSKrescodee (MSKAPI MSK_dualsensitivity) (
  MSKtask_t task,
  MSKint32t numj,
  const MSKint32t * subj,
  MSKrealt * leftpricej,
  MSKrealt * rightpricej,
  MSKrealt * leftrangej,
  MSKrealt * rightrangej)
```

Calculates sensitivity information for objective coefficients. The indexes of the coefficients to analyze are

$$\{\texttt{subj}[i] \mid i = 0, \ldots, \texttt{numj} - 1\}$$

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_t*) – An optimization task. (input)
- numj (*MSKint32t*) – Number of coefficients to be analyzed. Length of subj. (input)
- subj (*MSKint32t* *) – Indexes of objective coefficients to analyze. (input)
- leftpricej (*MSKrealt* *) – leftpricej$[j]$ is the left shadow price for the coefficient with index subj[j]. (output)
- rightpricej (*MSKrealt* *) – rightpricej$[j]$ is the right shadow price for the coefficient with index subj[j]. (output)
- leftrangej (*MSKrealt* *) – leftrangej$[j]$ is the left range β_1 for the coefficient with index subj[j]. (output)
- rightrangej (*MSKrealt* *) – rightrangej$[j]$ is the right range β_2 for the coefficient with index subj[j]. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Sensitivity analysis*

MSK_echoenv

```
MSKrescodee (MSKAPIVA MSK_echoenv) (
  MSKenv_t env,
  MSKstreamtypee whichstream,
  const char * format,
  ...)
```

Prints a formatted message to the environment stream.

Parameters

- env (*MSKenv_t*) – The MOSEK environment. (input)
- whichstream (*MSKstreamtypee*) – Index of the stream. (input)

- `format` (`char*`) – Is a valid C format string which matches the arguments in `...`. (input)
- `varnumarg` (`...`) – A variable argument list. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*

`MSK_echointro`

```
MSKrescodee (MSKAPI MSK_echointro) (
  MSKenv_t env,
  MSKint32t longver)
```

Prints an intro to message stream.

Parameters

- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `longver` (*MSKint32t*) – If non-zero, then the intro is slightly longer. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*

`MSK_echotask`

```
MSKrescodee (MSKAPIVA MSK_echotask) (
  MSKtask_t task,
  MSKstreamtypee whichstream,
  const char * format,
  ...)
```

Prints a formatted string to a task stream.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichstream` (*MSKstreamtypee*) – Index of the stream. (input)
- `format` (`char*`) – Is a valid C format string which matches the arguments in `...`. (input)
- `varnumarg` (`...`) – Additional arguments (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

`MSK_emptyafebarfrow`

```
MSKrescodee (MSKAPI MSK_emptyafebarfrow) (
  MSKtask_t task,
  MSKint64t afeidx)
```

Clears a row in $\overline{F}$ i.e. sets $\overline{F}_{\mathrm{afeidx},*} = 0$.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `afeidx` (*MSKint64t*) – Row index of $\overline{F}$. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*

`MSK_emptyafebarfrowlist`

```
MSKrescodee (MSKAPI MSK_emptyafebarfrowlist) (
  MSKtask_t task,
  MSKint64t numafeidx,
  const MSKint64t * afeidxlist)
```

Clears a number of rows in $\overline{F}$ i.e. sets $\overline{F}_{i,*} = 0$ for all indices i in `afeidxlist`.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `numafeidx` (*MSKint64t*) – The number of rows to zero, length of `afeidxlist`. (input)
- `afeidxlist` (*MSKint64t**) – Indices of rows in $\overline{F}$ to clear. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*

`MSK_emptyafefcol`

```
MSKrescodee (MSKAPI MSK_emptyafefcol) (
  MSKtask_t task,
  MSKint32t varidx)
```

Clears one column in the affine constraint matrix F, that is sets $F_{*,\mathrm{varidx}} = 0$.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `varidx` (*MSKint32t*) – Index of a variable (column in F). (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

`MSK_emptyafefcollist`

```
MSKrescodee (MSKAPI MSK_emptyafefcollist) (
  MSKtask_t task,
  MSKint64t numvaridx,
  const MSKint32t * varidx)
```

Clears a number of columns in F i.e. sets $F_{*,j} = 0$ for all indices j in `varidx`.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `numvaridx` (*MSKint64t*) – The number of columns to zero, length of `varidx` list. (input)
- `varidx` (*MSKint32t**) – Indices of variables (columns) in F to clear. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

`MSK_emptyafefrow`

```
MSKrescodee (MSKAPI MSK_emptyafefrow) (
  MSKtask_t task,
  MSKint64t afeidx)
```

Clears one row in the affine constraint matrix F, that is sets $F_{\mathrm{afeidx},*} = 0$.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)

- afeidx (*MSKint64t*) – Index of a row in F. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

MSK_emptyafefrowlist

```
MSKrescodee (MSKAPI MSK_emptyafefrowlist) (
  MSKtask_t task,
  MSKint64t numafeidx,
  const MSKint64t * afeidx)
```

Clears a number of rows in F i.e. sets $F_{i,*} = 0$ for all indices i in afeidx.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numafeidx (*MSKint64t*) – The number of rows to zero, length of afeidxlist. (input)
- afeidx (*MSKint64t**) – Indices of rows in F to clear. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

MSK_evaluateacc

```
MSKrescodee (MSKAPI MSK_evaluateacc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint64t accidx,
  MSKrealt * activity)
```

Evaluates the activity of an affine conic constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- accidx (*MSKint64t*) – The index of the affine conic constraint. (input)
- activity (*MSKrealt**) – The activity of the affine conic constraint. The array should have length equal to the dimension of the constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*, *Problem data - affine conic constraints*

MSK_evaluateaccs

```
MSKrescodee (MSKAPI MSK_evaluateaccs) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * activity)
```

Evaluates the activities of all affine conic constraints.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- activity (*MSKrealt**) – The activity of affine conic constraints. The array should have length equal to the sum of dimensions of all affine conic constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*, *Problem data - affine conic constraints*

`MSK_expirylicenses`

```
MSKrescodee (MSKAPI MSK_expirylicenses) (
  MSKenv_t env,
  MSKint64t * expiry)
```

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 *MSK_resetexpirylicenses*, until now.

Parameters

- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `expiry` (*MSKint64t by reference*) – If nonnegative, then it is the minimum number days to expiry of any feature that has been checked out. (output)

Return (*MSKrescodee*) – The function response code.

Groups *License system*

`MSK_freedbgenv`

```
void (MSKAPI MSK_freedbgenv) (
  MSKenv_t env,
  void * buffer,
  const char * file,
  const unsigned line)
```

Frees space allocated by **MOSEK**. Debug version of *MSK_freeenv*.

Parameters

- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `buffer` (`void*`) – A pointer. (input/output)
- `file` (`char*`) – File from which the function is called. (input)
- `line` (`unsigned`) – Line in the file from which the function is called. (input)

Return (`void`)

Groups *System, memory and debugging*

`MSK_freedbgtask`

```
void (MSKAPI MSK_freedbgtask) (
  MSKtask_t task,
  void * buffer,
  const char * file,
  const unsigned line)
```

Frees space allocated by **MOSEK**. Debug version of *MSK_freetask*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `buffer` (`void*`) – A pointer. (input/output)
- `file` (`char*`) – File from which the function is called. (input)
- `line` (`unsigned`) – Line in the file from which the function is called. (input)

Return (`void`)

Groups *System, memory and debugging*

`MSK_freeenv`

```
void (MSKAPI MSK_freeenv) (
  MSKenv_t env,
  void * buffer)
```

Frees space allocated by a **MOSEK** function. Must not be applied to the **MOSEK** environment and task.

Parameters
- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `buffer` (`void*`) – A pointer. (input/output)

Return (`void`)

Groups *System, memory and debugging*

MSK_freetask

```
void (MSKAPI MSK_freetask) (
  MSKtask_t task,
  void * buffer)
```

Frees space allocated by a **MOSEK** function. Must not be applied to the **MOSEK** environment and task.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `buffer` (`void*`) – A pointer. (input/output)

Return (`void`)

Groups *System, memory and debugging*

MSK_gemm

```
MSKrescodee (MSKAPI MSK_gemm) (
  MSKenv_t env,
  MSKtransposee transa,
  MSKtransposee transb,
  MSKint32t m,
  MSKint32t n,
  MSKint32t k,
  MSKrealt alpha,
  const MSKrealt * a,
  const MSKrealt * b,
  MSKrealt beta,
  MSKrealt * c)
```

Performs a matrix multiplication plus addition of dense matrices. Given A, B and C of compatible dimensions, this function computes

$$C := \alpha op(A) op(B) + \beta C$$

where α, β are two scalar values. The function $op(X)$ denotes X if transX is *MSK_TRANSPOSE_NO*, or X^T if set to *MSK_TRANSPOSE_YES*. The matrix C has m rows and n columns, and the other matrices must have compatible dimensions.

The result of this operation is stored in C. It must not overlap with the other input arrays.

Parameters
- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `transa` (*MSKtransposee*) – Indicates whether the matrix A must be transposed. (input)

- transsb (*MSKtransposee*) – Indicates whether the matrix B must be transposed. (input)
- m (*MSKint32t*) – Indicates the number of rows of matrix C. (input)
- n (*MSKint32t*) – Indicates the number of columns of matrix C. (input)
- k (*MSKint32t*) – Specifies the common dimension along which $op(A)$ and $op(B)$ are multiplied. For example, if neither A nor B are transposed, then this is the number of columns in A and also the number of rows in B. (input)
- alpha (*MSKrealt*) – A scalar value multiplying the result of the matrix multiplication. (input)
- a (*MSKrealt**) – The pointer to the array storing matrix A in a column-major format. (input)
- b (*MSKrealt**) – The pointer to the array storing matrix B in a column-major format. (input)
- beta (*MSKrealt*) – A scalar value that multiplies C. (input)
- c (*MSKrealt**) – The pointer to the array storing matrix C in a column-major format. (input/output)

Return (*MSKrescodee*) – The function response code.

Groups *Linear algebra*

MSK_gemv

```
MSKrescodee (MSKAPI MSK_gemv) (
  MSKenv_t env,
  MSKtransposee transa,
  MSKint32t m,
  MSKint32t n,
  MSKrealt alpha,
  const MSKrealt * a,
  const MSKrealt * x,
  MSKrealt beta,
  MSKrealt * y)
```

Computes the multiplication of a scaled dense matrix times a dense vector, plus a scaled dense vector. Precisely, if trans is *MSK_TRANSPOSE_NO* then the update is

$$y := \alpha A x + \beta y,$$

and if trans is *MSK_TRANSPOSE_YES* then

$$y := \alpha A^T x + \beta y,$$

where α, β are scalar values and A is a matrix with m rows and n columns.

Note that the result is stored overwriting y. It must not overlap with the other input arrays.

Parameters

- env (*MSKenv_t*) – The MOSEK environment. (input)
- transa (*MSKtransposee*) – Indicates whether the matrix A must be transposed. (input)
- m (*MSKint32t*) – Specifies the number of rows of the matrix A. (input)
- n (*MSKint32t*) – Specifies the number of columns of the matrix A. (input)
- alpha (*MSKrealt*) – A scalar value multiplying the matrix A. (input)
- a (*MSKrealt**) – A pointer to the array storing matrix A in a column-major format. (input)
- x (*MSKrealt**) – A pointer to the array storing the vector x. (input)
- beta (*MSKrealt*) – A scalar value multiplying the vector y. (input)

- y (*MSKrealt* *) – A pointer to the array storing the vector y. (input/output)

Return (*MSKrescodee*) – The function response code.

Groups *Linear algebra*

MSK_generateaccnames

```
MSKrescodee (MSKAPI MSK_generateaccnames) (
  MSKtask_t task,
  MSKint64t num,
  const MSKint64t * sub,
  const char * fmt,
  MSKint32t ndims,
  const MSKint32t * dims,
  const MSKint64t * sp,
  MSKint32t numnamedaxis,
  const MSKint32t * namedaxisidxs,
  MSKint64t numnames,
  const char ** names)
```

Internal.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint64t*) – Number of variable indexes. (input)
- sub (*MSKint64t* *) – Indexes of the affine conic constraints. (input)
- fmt (char*) – The variable name formatting string. (input)
- ndims (*MSKint32t*) – Number of dimensions in the shape. (input)
- dims (*MSKint32t* *) – Dimensions in the shape. (input)
- sp (*MSKint64t* *) – Items that should be named. (input)
- numnamedaxis (*MSKint32t*) – Number of named axes (input)
- namedaxisidxs (*MSKint32t* *) – List if named index axes (input)
- numnames (*MSKint64t*) – Total number of names. (input)
- names (char**) – All axis names. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

MSK_generatebarvarnames

```
MSKrescodee (MSKAPI MSK_generatebarvarnames) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subj,
  const char * fmt,
  MSKint32t ndims,
  const MSKint32t * dims,
  const MSKint64t * sp,
  MSKint32t numnamedaxis,
  const MSKint32t * namedaxisidxs,
  MSKint64t numnames,
  const char ** names)
```

Generates systematic names for variables.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of variable indexes. (input)

- subj (*MSKint32t* *) – Indexes of the variables. (input)
- fmt (char*) – The variable name formatting string. (input)
- ndims (*MSKint32t*) – Number of dimensions in the shape. (input)
- dims (*MSKint32t* *) – Dimensions in the shape. (input)
- sp (*MSKint64t* *) – Items that should be named. (input)
- numnamedaxis (*MSKint32t*) – Number of named axes (input)
- namedaxisidxs (*MSKint32t* *) – List if named index axes (input)
- numnames (*MSKint64t*) – Total number of names. (input)
- names (char**) – All axis names. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - variables*, *Problem data - linear part*

~~MSK_generateconenames~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_generateconenames) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subk,
  const char * fmt,
  MSKint32t ndims,
  const MSKint32t * dims,
  const MSKint64t * sp,
  MSKint32t numnamedaxis,
  const MSKint32t * namedaxisidxs,
  MSKint64t numnames,
  const char ** names)
```

Internal, deprecated.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of cone indexes. (input)
- subk (*MSKint32t* *) – Indexes of the cone. (input)
- fmt (char*) – The cone name formatting string. (input)
- ndims (*MSKint32t*) – Number of dimensions in the shape. (input)
- dims (*MSKint32t* *) – Dimensions in the shape. (input)
- sp (*MSKint64t* *) – Items that should be named. (input)
- numnamedaxis (*MSKint32t*) – Number of named axes (input)
- namedaxisidxs (*MSKint32t* *) – List if named index axes (input)
- numnames (*MSKint64t*) – Total number of names. (input)
- names (char**) – All axis names. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - cones (deprecated)*

MSK_generateconnames

```
MSKrescodee (MSKAPI MSK_generateconnames) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subi,
  const char * fmt,
  MSKint32t ndims,
  const MSKint32t * dims,
```

(continues on next page)

(continued from previous page)

```
  const MSKint64t * sp,
  MSKint32t numnamedaxis,
  const MSKint32t * namedaxisidxs,
  MSKint64t numnames,
  const char ** names)
```

Generates systematic names for constraints.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of constraint indexes. (input)
- `subi` (*MSKint32t* *) – Indexes of the constraints. (input)
- `fmt` (`char*`) – The constraint name formatting string. (input)
- `ndims` (*MSKint32t*) – Number of dimensions in the shape. (input)
- `dims` (*MSKint32t* *) – Dimensions in the shape. (input)
- `sp` (*MSKint64t* *) – Items that should be named. (input)
- `numnamedaxis` (*MSKint32t*) – Number of named axes (input)
- `namedaxisidxs` (*MSKint32t* *) – List if named index axes (input)
- `numnames` (*MSKint64t*) – Total number of names. (input)
- `names` (`char**`) – All axis names. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - constraints*, *Problem data - linear part*

`MSK_generatedjcnames`

```
MSKrescodee (MSKAPI MSK_generatedjcnames) (
  MSKtask_t task,
  MSKint64t num,
  const MSKint64t * sub,
  const char * fmt,
  MSKint32t ndims,
  const MSKint32t * dims,
  const MSKint64t * sp,
  MSKint32t numnamedaxis,
  const MSKint32t * namedaxisidxs,
  MSKint64t numnames,
  const char ** names)
```

Internal.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint64t*) – Number of variable indexes. (input)
- `sub` (*MSKint64t* *) – Indexes of the disjunctive constraints. (input)
- `fmt` (`char*`) – The variable name formatting string. (input)
- `ndims` (*MSKint32t*) – Number of dimensions in the shape. (input)
- `dims` (*MSKint32t* *) – Dimensions in the shape. (input)
- `sp` (*MSKint64t* *) – Items that should be named. (input)
- `numnamedaxis` (*MSKint32t*) – Number of named axes (input)
- `namedaxisidxs` (*MSKint32t* *) – List if named index axes (input)
- `numnames` (*MSKint64t*) – Total number of names. (input)
- `names` (`char**`) – All axis names. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

MSK_generatevarnames

```
MSKrescodee (MSKAPI MSK_generatevarnames) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subj,
  const char * fmt,
  MSKint32t ndims,
  const MSKint32t * dims,
  const MSKint64t * sp,
  MSKint32t numnamedaxis,
  const MSKint32t * namedaxisidxs,
  MSKint64t numnames,
  const char ** names)
```

Generates systematic names for variables.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of variable indexes. (input)
- `subj` (*MSKint32t**) – Indexes of the variables. (input)
- `fmt` (`char*`) – The variable name formatting string. (input)
- `ndims` (*MSKint32t*) – Number of dimensions in the shape. (input)
- `dims` (*MSKint32t**) – Dimensions in the shape. (input)
- `sp` (*MSKint64t**) – Items that should be named. (input)
- `numnamedaxis` (*MSKint32t*) – Number of named axes (input)
- `namedaxisidxs` (*MSKint32t**) – List if named index axes (input)
- `numnames` (*MSKint64t*) – Total number of names. (input)
- `names` (`char**`) – All axis names. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - variables*, *Problem data - linear part*

MSK_getaccafeidxlist

```
MSKrescodee (MSKAPI MSK_getaccafeidxlist) (
  MSKtask_t task,
  MSKint64t accidx,
  MSKint64t * afeidxlist)
```

Obtains the list of affine expressions appearing in the affine conic constraint.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `accidx` (*MSKint64t*) – Index of the affine conic constraint. (input)
- `afeidxlist` (*MSKint64t**) – List of indexes of affine expressions appearing in the constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*, *Inspecting the task*

MSK_getaccb

```
MSKrescodee (MSKAPI MSK_getaccb) (
  MSKtask_t task,
  MSKint64t accidx,
  MSKrealt * b)
```

Obtains the additional constant term vector appearing in the affine conic constraint.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `accidx` (*MSKint64t*) – Index of the affine conic constraint. (input)
- `b` (*MSKrealt* *) – The vector b appearing in the constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*, *Inspecting the task*

`MSK_getaccbarfblocktriplet`

```
MSKrescodee (MSKAPI MSK_getaccbarfblocktriplet) (
  MSKtask_t task,
  MSKint64t maxnumtrip,
  MSKint64t * numtrip,
  MSKint64t * acc_afe,
  MSKint32t * bar_var,
  MSKint32t * blk_row,
  MSKint32t * blk_col,
  MSKrealt * blk_val)
```

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_t*) – An optimization task. (input)
- `maxnumtrip` (*MSKint64t*) – `acc_afe`, `bar_var`, `blk_row`, `blk_col` and `blk_val` must be `numtrip` long. (input)
- `numtrip` (*MSKint64t by reference*) – Number of elements in the block triplet form. (output)
- `acc_afe` (*MSKint64t* *) – Index of the AFE within the concatenated list of AFEs in ACCs. (output)
- `bar_var` (*MSKint32t* *) – Symmetric matrix variable index. (output)
- `blk_row` (*MSKint32t* *) – Block row index. (output)
- `blk_col` (*MSKint32t* *) – Block column index. (output)
- `blk_val` (*MSKrealt* *) – The numerical value associated with each block triplet. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*

`MSK_getaccbarfnumblocktriplets`

```
MSKrescodee (MSKAPI MSK_getaccbarfnumblocktriplets) (
  MSKtask_t task,
  MSKint64t * numtrip)
```

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_t*) – An optimization task. (input)

- `numtrip` (*MSKint64t by reference*) – An upper bound on the number of elements in the block triplet form of $\overline{F}$., as used within the ACCs. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Problem data - affine conic constraints*, *Inspecting the task*

`MSK_getaccdomain`

```
MSKrescodee (MSKAPI MSK_getaccdomain) (
  MSKtask_t task,
  MSKint64t accidx,
  MSKint64t * domidx)
```

Obtains the domain appearing in the affine conic constraint.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `accidx` (*MSKint64t*) – The index of the affine conic constraint. (input)
- `domidx` (*MSKint64t by reference*) – The index of domain in the affine conic constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*, *Inspecting the task*

`MSK_getaccdoty`

```
MSKrescodee (MSKAPI MSK_getaccdoty) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint64t accidx,
  MSKrealt * doty)
```

Obtains the $\dot{y}$ vector for a solution (the dual values of an affine conic constraint).

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `accidx` (*MSKint64t*) – The index of the affine conic constraint. (input)
- `doty` (*MSKrealt**) – The dual values for this affine conic constraint. The array should have length equal to the dimension of the constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*, *Problem data - affine conic constraints*

`MSK_getaccdotys`

```
MSKrescodee (MSKAPI MSK_getaccdotys) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * doty)
```

Obtains the $\dot{y}$ vector for a solution (the dual values of all affine conic constraint).

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `doty` (*MSKrealt**) – The dual values of affine conic constraints. The array should have length equal to the sum of dimensions of all affine conic constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*, *Problem data - affine conic constraints*

MSK_getaccfnumnz

```
MSKrescodee (MSKAPI MSK_getaccfnumnz) (
  MSKtask_t task,
  MSKint64t * accfnnz)
```

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_t*) – An optimization task. (input)
- accfnnz (*MSKint64t by reference*) – Number of non-zeros in F implied by ACCs. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*, *Inspecting the task*

MSK_getaccftrip

```
MSKrescodee (MSKAPI MSK_getaccftrip) (
  MSKtask_t task,
  MSKint64t * frow,
  MSKint32t * fcol,
  MSKrealt * fval)
```

Obtains the F (that would be implied by the ordering of the AFEs within the ACCs) in triplet format.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- frow (*MSKint64t* *) – Row indices of nonzeros in the implied F matrix. (output)
- fcol (*MSKint32t* *) – Column indices of nonzeros in the implied F matrix. (output)
- fval (*MSKrealt* *) – Values of nonzero entries in the implied F matrix. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*, *Inspecting the task*

MSK_getaccgvector

```
MSKrescodee (MSKAPI MSK_getaccgvector) (
  MSKtask_t task,
  MSKrealt * g)
```

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_t*) – An optimization task. (input)
- g (*MSKrealt* *) – The g used within the ACCs as a dense vector. The length is sum of the dimensions of the ACCs. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - affine conic constraints*

MSK_getaccn

```
MSKrescodee (MSKAPI MSK_getaccn) (
  MSKtask_t task,
  MSKint64t accidx,
  MSKint64t * n)
```

Obtains the dimension of the affine conic constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- accidx (*MSKint64t*) – The index of the affine conic constraint. (input)
- n (*MSKint64t by reference*) – The dimension of the affine conic constraint (equal to the dimension of its domain). (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*, *Inspecting the task*

MSK_getaccname

```
MSKrescodee (MSKAPI MSK_getaccname) (
  MSKtask_t task,
  MSKint64t accidx,
  MSKint32t sizename,
  char * name)
```

Obtains the name of an affine conic constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- accidx (*MSKint64t*) – Index of an affine conic constraint. (input)
- sizename (*MSKint32t*) – The length of the buffer pointed to by the name argument. (input)
- name (char*) – Returns the required name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - affine conic constraints*, *Inspecting the task*

MSK_getaccnamelen

```
MSKrescodee (MSKAPI MSK_getaccnamelen) (
  MSKtask_t task,
  MSKint64t accidx,
  MSKint32t * len)
```

Obtains the length of the name of an affine conic constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- accidx (*MSKint64t*) – Index of an affine conic constraint. (input)
- len (*MSKint32t by reference*) – Returns the length of the indicated name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - affine conic constraints*, *Inspecting the task*

MSK_getaccntot

```
MSKrescodee (MSKAPI MSK_getaccntot) (
  MSKtask_t task,
  MSKint64t * n)
```

Obtains the total dimension of all affine conic constraints (the sum of all their dimensions).

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `n` (*MSKint64t by reference*) – The total dimension of all affine conic constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*, *Inspecting the task*

`MSK_getaccs`

```
MSKrescodee (MSKAPI MSK_getaccs) (
  MSKtask_t task,
  MSKint64t * domidxlist,
  MSKint64t * afeidxlist,
  MSKrealt * b)
```

Obtains full data of all affine conic constraints. The output array `domainidxlist` must have at least length determined by *MSK_getnumacc*. The output arrays `afeidxlist` and `b` must have at least length determined by *MSK_getaccntot*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `domidxlist` (*MSKint64t* *) – The list of domains appearing in all affine conic constraints. (output)
- `afeidxlist` (*MSKint64t* *) – The concatenation of index lists of affine expressions appearing in all affine conic constraints. (output)
- `b` (*MSKrealt* *) – The concatenation of vectors b appearing in all affine conic constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*, *Inspecting the task*

`MSK_getacol`

```
MSKrescodee (MSKAPI MSK_getacol) (
  MSKtask_t task,
  MSKint32t j,
  MSKint32t * nzj,
  MSKint32t * subj,
  MSKrealt * valj)
```

Obtains one column of A in a sparse format.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `j` (*MSKint32t*) – Index of the column. (input)
- `nzj` (*MSKint32t by reference*) – Number of non-zeros in the column obtained. (output)
- `subj` (*MSKint32t* *) – Row indices of the non-zeros in the column obtained. (output)
- `valj` (*MSKrealt* *) – Numerical values in the column obtained. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

`MSK_getacolnumnz`

```
MSKrescodee (MSKAPI MSK_getacolnumnz) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t * nzj)
```

Obtains the number of non-zero elements in one column of A.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `i` (*MSKint32t*) – Index of the column. (input)
- `nzj` (*MSKint32t by reference*) – Number of non-zeros in the j-th column of A. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

`MSK_getacolslice`

```
MSKrescodee (MSKAPI MSK_getacolslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint32t maxnumnz,
  MSKint32t * ptrb,
  MSKint32t * ptre,
  MSKint32t * sub,
  MSKrealt * val)
```

Obtains a sequence of columns from A in sparse format.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `first` (*MSKint32t*) – Index of the first column in the sequence. (input)
- `last` (*MSKint32t*) – Index of the last column in the sequence **plus one**. (input)
- `maxnumnz` (*MSKint32t*) – Denotes the length of the arrays `sub` and `val`. (input)
- `ptrb` (*MSKint32t* *) – `ptrb[t]` is an index pointing to the first element in the t-th column obtained. (output)
- `ptre` (*MSKint32t* *) – `ptre[t]` is an index pointing to the last element plus one in the t-th column obtained. (output)
- `sub` (*MSKint32t* *) – Contains the row subscripts. (output)
- `val` (*MSKrealt* *) – Contains the coefficient values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

`MSK_getacolslice64`

```
MSKrescodee (MSKAPI MSK_getacolslice64) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint64t maxnumnz,
  MSKint64t * ptrb,
  MSKint64t * ptre,
  MSKint32t * sub,
  MSKrealt * val)
```

Obtains a sequence of columns from A in sparse format.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `first` (*MSKint32t*) – Index of the first column in the sequence. (input)
- `last` (*MSKint32t*) – Index of the last column in the sequence **plus one**. (input)
- `maxnumnz` (*MSKint64t*) – Denotes the length of the arrays `sub` and `val`. (input)
- `ptrb` (*MSKint64t* *) – `ptrb[t]` is an index pointing to the first element in the t-th column obtained. (output)
- `ptre` (*MSKint64t* *) – `ptre[t]` is an index pointing to the last element plus one in the t-th column obtained. (output)
- `sub` (*MSKint32t* *) – Contains the row subscripts. (output)
- `val` (*MSKrealt* *) – Contains the coefficient values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

`MSK_getacolslicenumnz`

```
MSKrescodee (MSKAPI MSK_getacolslicenumnz) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint32t * numnz)
```

Obtains the number of non-zeros in a slice of columns of A.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `first` (*MSKint32t*) – Index of the first column in the sequence. (input)
- `last` (*MSKint32t*) – Index of the last column **plus one** in the sequence. (input)
- `numnz` (*MSKint32t by reference*) – Number of non-zeros in the slice. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

`MSK_getacolslicenumnz64`

```
MSKrescodee (MSKAPI MSK_getacolslicenumnz64) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint64t * numnz)
```

Obtains the number of non-zeros in a slice of columns of A.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `first` (*MSKint32t*) – Index of the first column in the sequence. (input)
- `last` (*MSKint32t*) – Index of the last column **plus one** in the sequence. (input)
- `numnz` (*MSKint64t by reference*) – Number of non-zeros in the slice. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

`MSK_getacolslicetrip`

```
MSKrescodee (MSKAPI MSK_getacolslicetrip) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint64t maxnumnz,
  MSKint32t * subi,
  MSKint32t * subj,
  MSKrealt * val)
```

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_t*) – An optimization task. (input)
- `first` (*MSKint32t*) – Index of the first column in the sequence. (input)
- `last` (*MSKint32t*) – Index of the last column in the sequence **plus one**. (input)
- `maxnumnz` (*MSKint64t*) – Denotes the length of the arrays `subi`, `subj`, and `val`. (input)
- `subi` (*MSKint32t* *) – Constraint subscripts. (output)
- `subj` (*MSKint32t* *) – Column subscripts. (output)
- `val` (*MSKrealt* *) – Values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

`MSK_getafebarfblocktriplet`

```
MSKrescodee (MSKAPI MSK_getafebarfblocktriplet) (
  MSKtask_t task,
  MSKint64t maxnumtrip,
  MSKint64t * numtrip,
  MSKint64t * afeidx,
  MSKint32t * barvaridx,
  MSKint32t * subk,
  MSKint32t * subl,
  MSKrealt * valkl)
```

Obtains $\overline{F}$ in block triplet form.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `maxnumtrip` (*MSKint64t*) – `afeidx`, `barvaridx`, `subk`, `subl` and `valkl` must be `maxnumtrip` long. (input)
- `numtrip` (*MSKint64t by reference*) – Number of elements in the block triplet form. (output)
- `afeidx` (*MSKint64t* *) – Constraint index. (output)
- `barvaridx` (*MSKint32t* *) – Symmetric matrix variable index. (output)
- `subk` (*MSKint32t* *) – Block row index. (output)
- `subl` (*MSKint32t* *) – Block column index. (output)
- `valkl` (*MSKrealt* *) – The numerical value associated with each block triplet. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*

`MSK_getafebarfnumblocktriplets`

```
MSKrescodee (MSKAPI MSK_getafebarfnumblocktriplets) (
  MSKtask_t task,
  MSKint64t * numtrip)
```

Obtains an upper bound on the number of elements in the block triplet form of $\overline{F}$.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `numtrip` (*MSKint64t by reference*) – An upper bound on the number of elements in the block triplet form of $\overline{F}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

`MSK_getafebarfnumrowentries`

```
MSKrescodee (MSKAPI MSK_getafebarfnumrowentries) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKint32t * numentr)
```

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_t*) – An optimization task. (input)
- `afeidx` (*MSKint64t*) – Row index of $\overline{F}$. (input)
- `numentr` (*MSKint32t by reference*) – Number of nonzero entries in a row of $\overline{F}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*, *Inspecting the task*

`MSK_getafebarfrow`

```
MSKrescodee (MSKAPI MSK_getafebarfrow) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKint32t * barvaridx,
  MSKint64t * ptrterm,
  MSKint64t * numterm,
  MSKint64t * termidx,
  MSKrealt * termweight)
```

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 *MSK_getafebarfrowinfo*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `afeidx` (*MSKint64t*) – Row index of $\overline{F}$. (input)

- barvaridx (*MSKint32t* *) – Semidefinite variable indices of nonzero entries in the row of $\overline{F}$. (output)
- ptrterm (*MSKint64t* *) – Pointers to the start of each entry's description. (output)
- numterm (*MSKint64t* *) – Number of terms in the weighted sum representation of each entry. (output)
- termidx (*MSKint64t* *) – Indices of semidefinite matrices from the matrix store E. (output)
- termweight (*MSKrealt* *) – Weights appearing in the weighted sum representations of all entries. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*, *Inspecting the task*

MSK_getafebarfrowinfo

```
MSKrescodee (MSKAPI MSK_getafebarfrowinfo) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKint32t * numentr,
  MSKint64t * numterm)
```

Obtains information about one row of $\overline{F}$: the number of nonzero entries, that is the number of j such that $\overline{F}_{\mathrm{afeidx},j}$ 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 *MSK_getafebarfrow*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- afeidx (*MSKint64t*) – Row index of $\overline{F}$. (input)
- numentr (*MSKint32t by reference*) – Number of nonzero entries in a row of $\overline{F}$. (output)
- numterm (*MSKint64t by reference*) – Number of terms in the weighted sums representation of the row of $\overline{F}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*, *Inspecting the task*

MSK_getafefnumnz

```
MSKrescodee (MSKAPI MSK_getafefnumnz) (
  MSKtask_t task,
  MSKint64t * numnz)
```

Obtains the total number of nonzeros in F.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numnz (*MSKint64t by reference*) – Number of non-zeros in F. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Inspecting the task*

MSK_getafefrow

```
MSKrescodee (MSKAPI MSK_getafefrow) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKint32t * numnz,
  MSKint32t * varidx,
  MSKrealt * val)
```

Obtains one row of F in sparse format.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- afeidx (*MSKint64t*) – Index of a row in F. (input)
- numnz (*MSKint32t by reference*) – Number of non-zeros in the row obtained. (output)
- varidx (*MSKint32t* *) – Column indices of the non-zeros in the row obtained. (output)
- val (*MSKrealt* *) – Values of the non-zeros in the row obtained. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Inspecting the task*

MSK_getafefrownumnz

```
MSKrescodee (MSKAPI MSK_getafefrownumnz) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKint32t * numnz)
```

Obtains the number of nonzeros in one row of F.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- afeidx (*MSKint64t*) – Index of a row in F. (input)
- numnz (*MSKint32t by reference*) – Number of non-zeros in row afeidx of F. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Inspecting the task*

MSK_getafeftrip

```
MSKrescodee (MSKAPI MSK_getafeftrip) (
  MSKtask_t task,
  MSKint64t * afeidx,
  MSKint32t * varidx,
  MSKrealt * val)
```

Obtains the F in triplet format.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- afeidx (*MSKint64t* *) – Row indices of nonzeros. (output)
- varidx (*MSKint32t* *) – Column indices of nonzeros. (output)
- val (*MSKrealt* *) – Values of nonzero entries. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Inspecting the task*

MSK_getafeg

```
MSKrescodee (MSKAPI MSK_getafeg) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKrealt * g)
```

Obtains a single coefficient in g.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- afeidx (*MSKint64t*) – Index of an element in g. (input)
- g (*MSKrealt by reference*) – The value of g_{afeidx}. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Inspecting the task*

MSK_getafegslice

```
MSKrescodee (MSKAPI MSK_getafegslice) (
  MSKtask_t task,
  MSKint64t first,
  MSKint64t last,
  MSKrealt * g)
```

Obtains a sequence of elements from the vector g of constant terms in the affine expressions list.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint64t*) – First index in the sequence. (input)
- last (*MSKint64t*) – Last index plus 1 in the sequence. (input)
- g (*MSKrealt**) – The slice g as a dense vector. The length is last-first. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - affine expressions*

MSK_getaij

```
MSKrescodee (MSKAPI MSK_getaij) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t j,
  MSKrealt * aij)
```

Obtains a single coefficient in A.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Row index of the coefficient to be returned. (input)
- j (*MSKint32t*) – Column index of the coefficient to be returned. (input)
- aij (*MSKrealt by reference*) – The required coefficient $a_{i,j}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

MSK_getapiecenumnz

```
MSKrescodee (MSKAPI MSK_getapiecenumnz) (
  MSKtask_t task,
  MSKint32t firsti,
  MSKint32t lasti,
  MSKint32t firstj,
  MSKint32t lastj,
  MSKint32t * numnz)
```

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 *MSK_getarownumnz* or *MSK_getacolnumnz*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- firsti (*MSKint32t*) – Index of the first row in the rectangular piece. (input)
- lasti (*MSKint32t*) – Index of the last row plus one in the rectangular piece. (input)
- firstj (*MSKint32t*) – Index of the first column in the rectangular piece. (input)
- lastj (*MSKint32t*) – Index of the last column plus one in the rectangular piece. (input)
- numnz (*MSKint32t by reference*) – Number of non-zero A elements in the rectangular piece. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

MSK_getarow

```
MSKrescodee (MSKAPI MSK_getarow) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t * nzi,
  MSKint32t * subi,
  MSKrealt * vali)
```

Obtains one row of A in a sparse format.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Index of the row. (input)
- nzi (*MSKint32t by reference*) – Number of non-zeros in the row obtained. (output)
- subi (*MSKint32t**) – Column indices of the non-zeros in the row obtained. (output)
- vali (*MSKrealt**) – Numerical values of the row obtained. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

MSK_getarownumnz

```
MSKrescodee (MSKAPI MSK_getarownumnz) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t * nzi)
```

Obtains the number of non-zero elements in one row of A.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Index of the row. (input)
- nzi (*MSKint32t by reference*) – Number of non-zeros in the i-th row of A. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

MSK_getarowslice

```
MSKrescodee (MSKAPI MSK_getarowslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint32t maxnumnz,
  MSKint32t * ptrb,
  MSKint32t * ptre,
  MSKint32t * sub,
  MSKrealt * val)
```

Obtains a sequence of rows from A in sparse format.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – Index of the first row in the sequence. (input)
- last (*MSKint32t*) – Index of the last row in the sequence **plus one**. (input)
- maxnumnz (*MSKint32t*) – Denotes the length of the arrays sub and val. (input)
- ptrb (*MSKint32t* *) – ptrb[t] is an index pointing to the first element in the t-th row obtained. (output)
- ptre (*MSKint32t* *) – ptre[t] is an index pointing to the last element plus one in the t-th row obtained. (output)
- sub (*MSKint32t* *) – Contains the column subscripts. (output)
- val (*MSKrealt* *) – Contains the coefficient values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

MSK_getarowslice64

```
MSKrescodee (MSKAPI MSK_getarowslice64) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint64t maxnumnz,
  MSKint64t * ptrb,
  MSKint64t * ptre,
  MSKint32t * sub,
  MSKrealt * val)
```

Obtains a sequence of rows from A in sparse format.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – Index of the first row in the sequence. (input)
- last (*MSKint32t*) – Index of the last row in the sequence **plus one**. (input)

- maxnumnz (*MSKint64t*) – Denotes the length of the arrays sub and val. (input)
- ptrb (*MSKint64t* *) – ptrb[t] is an index pointing to the first element in the *t*-th row obtained. (output)
- ptre (*MSKint64t* *) – ptre[t] is an index pointing to the last element plus one in the *t*-th row obtained. (output)
- sub (*MSKint32t* *) – Contains the column subscripts. (output)
- val (*MSKrealt* *) – Contains the coefficient values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

MSK_getarowslicenumnz

```
MSKrescodee (MSKAPI MSK_getarowslicenumnz) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint32t * numnz)
```

Obtains the number of non-zeros in a slice of rows of A.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – Index of the first row in the sequence. (input)
- last (*MSKint32t*) – Index of the last row **plus one** in the sequence. (input)
- numnz (*MSKint32t by reference*) – Number of non-zeros in the slice. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

MSK_getarowslicenumnz64

```
MSKrescodee (MSKAPI MSK_getarowslicenumnz64) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint64t * numnz)
```

Obtains the number of non-zeros in a slice of rows of A.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – Index of the first row in the sequence. (input)
- last (*MSKint32t*) – Index of the last row **plus one** in the sequence. (input)
- numnz (*MSKint64t by reference*) – Number of non-zeros in the slice. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

MSK_getarowslicetrip

```
MSKrescodee (MSKAPI MSK_getarowslicetrip) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint64t maxnumnz,
  MSKint32t * subi,
  MSKint32t * subj,
  MSKrealt * val)
```

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_t*) – An optimization task. (input)
- first (*MSKint32t*) – Index of the first row in the sequence. (input)
- last (*MSKint32t*) – Index of the last row in the sequence **plus one**. (input)
- maxnumnz (*MSKint64t*) – Denotes the length of the arrays subi, subj, and val. (input)
- subi (*MSKint32t* *) – Constraint subscripts. (output)
- subj (*MSKint32t* *) – Column subscripts. (output)
- val (*MSKrealt* *) – Values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

MSK_getatrip

```
MSKrescodee (MSKAPI MSK_getatrip) (
  MSKtask_t task,
  MSKint64t maxnumnz,
  MSKint32t * subi,
  MSKint32t * subj,
  MSKrealt * val)
```

Obtains A in sparse triplet format. The triplets corresponding to nonzero entries are stored in the arrays subi, subj and val.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- maxnumnz (*MSKint64t*) – Denotes the length of the arrays subi, subj, and val. (input)
- subi (*MSKint32t* *) – Constraint subscripts. (output)
- subj (*MSKint32t* *) – Column subscripts. (output)
- val (*MSKrealt* *) – Values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

MSK_getatruncatetol

```
MSKrescodee (MSKAPI MSK_getatruncatetol) (
  MSKtask_t task,
  MSKrealt * tolzero)
```

Obtains the tolerance value set with *MSK_putatruncatetol*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- tolzero (*MSKrealt* *) – All elements $|a_{i,j}|$ less than this tolerance is truncated to zero. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*, *Problem data - linear part*

MSK_getbarablocktriplet

```
MSKrescodee (MSKAPI MSK_getbarablocktriplet) (
  MSKtask_t task,
  MSKint64t maxnum,
  MSKint64t * num,
  MSKint32t * subi,
  MSKint32t * subj,
  MSKint32t * subk,
  MSKint32t * subl,
  MSKrealt * valijkl)
```

Obtains $\overline{A}$ in block triplet form.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `maxnum` (*MSKint64t*) – `subi`, `subj`, `subk`, `subl` and `valijkl` must be `maxnum` long. (input)
- `num` (*MSKint64t by reference*) – Number of elements in the block triplet form. (output)
- `subi` (*MSKint32t* *) – Constraint index. (output)
- `subj` (*MSKint32t* *) – Symmetric matrix variable index. (output)
- `subk` (*MSKint32t* *) – Block row index. (output)
- `subl` (*MSKint32t* *) – Block column index. (output)
- `valijkl` (*MSKrealt* *) – The numerical value associated with each block triplet. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

`MSK_getbaraidx`

```
MSKrescodee (MSKAPI MSK_getbaraidx) (
  MSKtask_t task,
  MSKint64t idx,
  MSKint64t maxnum,
  MSKint32t * i,
  MSKint32t * j,
  MSKint64t * num,
  MSKint64t * sub,
  MSKrealt * weights)
```

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_t*) – An optimization task. (input)
- `idx` (*MSKint64t*) – Position of the element in the vectorized form. (input)
- `maxnum` (*MSKint64t*) – `sub` and `weights` must be at least `maxnum` long. (input)
- `i` (*MSKint32t by reference*) – Row index of the element at position `idx`. (output)
- `j` (*MSKint32t by reference*) – Column index of the element at position `idx`. (output)
- `num` (*MSKint64t by reference*) – Number of terms in weighted sum that forms the element. (output)

- sub (*MSKint64t* *) – A list indexes of the elements from symmetric matrix storage that appear in the weighted sum. (output)
- weights (*MSKrealt* *) – The weights associated with each term in the weighted sum. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getbaraidxij

```
MSKrescodee (MSKAPI MSK_getbaraidxij) (
  MSKtask_t task,
  MSKint64t idx,
  MSKint32t * i,
  MSKint32t * j)
```

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_t*) – An optimization task. (input)
- idx (*MSKint64t*) – Position of the element in the vectorized form. (input)
- i (*MSKint32t by reference*) – Row index of the element at position idx. (output)
- j (*MSKint32t by reference*) – Column index of the element at position idx. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getbaraidxinfo

```
MSKrescodee (MSKAPI MSK_getbaraidxinfo) (
  MSKtask_t task,
  MSKint64t idx,
  MSKint64t * num)
```

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 *MSK_appendsparsesymmat* for details about the weighted sum.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- idx (*MSKint64t*) – The internal position of the element for which information should be obtained. (input)
- num (*MSKint64t by reference*) – Number of terms in the weighted sum that form the specified element in $\overline{A}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getbarasparsity

```
MSKrescodee (MSKAPI MSK_getbarasparsity) (
  MSKtask_t task,
  MSKint64t maxnumnz,
  MSKint64t * numnz,
  MSKint64t * idxij)
```

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 *MSK_getbaraidxinfo* and *MSK_getbaraidx*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- maxnumnz (*MSKint64t*) – The array idxij must be at least maxnumnz long. (input)
- numnz (*MSKint64t by reference*) – Number of nonzero elements in $\overline{A}$. (output)
- idxij (*MSKint64t* *) – Position of each nonzero element in the vectorized form of $\overline{A}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getbarcblocktriplet

```
MSKrescodee (MSKAPI MSK_getbarcblocktriplet) (
  MSKtask_t task,
  MSKint64t maxnum,
  MSKint64t * num,
  MSKint32t * subj,
  MSKint32t * subk,
  MSKint32t * subl,
  MSKrealt * valjkl)
```

Obtains $\overline{C}$ in block triplet form.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- maxnum (*MSKint64t*) – subj, subk, subl and valjkl must be maxnum long. (input)
- num (*MSKint64t by reference*) – Number of elements in the block triplet form. (output)
- subj (*MSKint32t* *) – Symmetric matrix variable index. (output)
- subk (*MSKint32t* *) – Block row index. (output)
- subl (*MSKint32t* *) – Block column index. (output)
- valjkl (*MSKrealt* *) – The numerical value associated with each block triplet. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getbarcidx

```
MSKrescodee (MSKAPI MSK_getbarcidx) (
  MSKtask_t task,
  MSKint64t idx,
```

(continues on next page)

(continued from previous page)

```
  MSKint64t maxnum,
  MSKint32t * j,
  MSKint64t * num,
  MSKint64t * sub,
  MSKrealt * weights)
```

Obtains information about an element in $\overline{C}$.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `idx` (*MSKint64t*) – Index of the element for which information should be obtained. (input)
- `maxnum` (*MSKint64t*) – `sub` and `weights` must be at least `maxnum` long. (input)
- `j` (*MSKint32t by reference*) – Row index in $\overline{C}$. (output)
- `num` (*MSKint64t by reference*) – Number of terms in the weighted sum. (output)
- `sub` (*MSKint64t**) – Elements appearing the weighted sum. (output)
- `weights` (*MSKrealt**) – Weights of terms in the weighted sum. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

`MSK_getbarcidxinfo`

```
MSKrescodee (MSKAPI MSK_getbarcidxinfo) (
  MSKtask_t task,
  MSKint64t idx,
  MSKint64t * num)
```

Obtains the number of terms in the weighted sum that forms a particular element in $\overline{C}$.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `idx` (*MSKint64t*) – Index of the element for which information should be obtained. The value is an index of a symmetric sparse variable. (input)
- `num` (*MSKint64t by reference*) – Number of terms that appear in the weighted sum that forms the requested element. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

`MSK_getbarcidxj`

```
MSKrescodee (MSKAPI MSK_getbarcidxj) (
  MSKtask_t task,
  MSKint64t idx,
  MSKint32t * j)
```

Obtains the row index of an element in $\overline{C}$.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `idx` (*MSKint64t*) – Index of the element for which information should be obtained. (input)
- `j` (*MSKint32t by reference*) – Row index in $\overline{C}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getbarcsparsity

```
MSKrescodee (MSKAPI MSK_getbarcsparsity) (
  MSKtask_t task,
  MSKint64t maxnumnz,
  MSKint64t * numnz,
  MSKint64t * idxj)
```

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 *MSK_getbarcidxinfo* and *MSK_getbarcidx*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- maxnumnz (*MSKint64t*) – idxj must be at least maxnumnz long. (input)
- numnz (*MSKint64t by reference*) – Number of nonzero elements in $\overline{C}$. (output)
- idxj (*MSKint64t* *) – Internal positions of the nonzeros elements in $\overline{C}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getbarsj

```
MSKrescodee (MSKAPI MSK_getbarsj) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t j,
  MSKrealt * barsj)
```

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_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- j (*MSKint32t*) – Index of the semidefinite variable. (input)
- barsj (*MSKrealt* *) – Value of $\overline{S}_j$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - semidefinite*

MSK_getbarsslice

```
MSKrescodee (MSKAPI MSK_getbarsslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKint64t slicesize,
  MSKrealt * barsslice)
```

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 *MSK_getbarsj*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)

- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – Index of the first semidefinite variable in the slice. (input)
- `last` (*MSKint32t*) – Index of the last semidefinite variable in the slice plus one. (input)
- `slicesize` (*MSKint64t*) – Denotes the length of the array `barsslice`. (input)
- `barsslice` (*MSKrealt* *) – Dual solution values of symmetric matrix variables in the slice, stored sequentially. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - semidefinite*

`MSK_getbarvarname`

```
MSKrescodee (MSKAPI MSK_getbarvarname) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t sizename,
  char * name)
```

Obtains the name of a semidefinite variable.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `i` (*MSKint32t*) – Index of the variable. (input)
- `sizename` (*MSKint32t*) – Length of the name buffer. (input)
- `name` (`char*`) – The requested name is copied to this buffer. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Inspecting the task*

`MSK_getbarvarnameindex`

```
MSKrescodee (MSKAPI MSK_getbarvarnameindex) (
  MSKtask_t task,
  const char * somename,
  MSKint32t * asgn,
  MSKint32t * index)
```

Obtains the index of semidefinite variable from its name.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `somename` (`char*`) – The name of the variable. (input)
- `asgn` (*MSKint32t by reference*) – Non-zero if the name `somename` is assigned to some semidefinite variable. (output)
- `index` (*MSKint32t by reference*) – The index of a semidefinite variable with the name `somename` (if one exists). (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Inspecting the task*

`MSK_getbarvarnamelen`

```
MSKrescodee (MSKAPI MSK_getbarvarnamelen) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t * len)
```

Obtains the length of the name of a semidefinite variable.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `i` (*MSKint32t*) – Index of the variable. (input)
- `len` (*MSKint32t by reference*) – Returns the length of the indicated name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Inspecting the task*

`MSK_getbarxj`

```
MSKrescodee (MSKAPI MSK_getbarxj) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t j,
  MSKrealt * barxj)
```

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_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `j` (*MSKint32t*) – Index of the semidefinite variable. (input)
- `barxj` (*MSKrealt**) – Value of $\overline{X}_j$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - semidefinite*

`MSK_getbarxslice`

```
MSKrescodee (MSKAPI MSK_getbarxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKint64t slicesize,
  MSKrealt * barxslice)
```

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 *MSK_getbarxj*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – Index of the first semidefinite variable in the slice. (input)
- `last` (*MSKint32t*) – Index of the last semidefinite variable in the slice plus one. (input)
- `slicesize` (*MSKint64t*) – Denotes the length of the array `barxslice`. (input)
- `barxslice` (*MSKrealt**) – Solution values of symmetric matrix variables in the slice, stored sequentially. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - semidefinite*

MSK_getbuildinfo

```
MSKrescodee (MSKAPI MSK_getbuildinfo) (
  char * buildstate,
  char * builddate)
```

Obtains build information.

Parameters

- `buildstate` (`char*`) – State of binaries, i.e. a debug, release candidate or final release. (output)
- `builddate` (`char*`) – Date when the binaries were built. (output)

Return (*`MSKrescodee`*) – The function response code.

Groups *Versions*

MSK_getc

```
MSKrescodee (MSKAPI MSK_getc) (
  MSKtask_t task,
  MSKrealt * c)
```

Obtains all objective coefficients c.

Parameters

- `task` (*`MSKtask_t`*) – An optimization task. (input)
- `c` (*`MSKrealt`*`*`) – Linear terms of the objective as a dense vector. The length is the number of variables. (output)

Return (*`MSKrescodee`*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*, *Problem data - variables*

MSK_getcallbackfunc

```
MSKrescodee (MSKAPI MSK_getcallbackfunc) (
  MSKtask_t task,
  MSKcallbackfunc * func,
  MSKuserhandle_t * handle)
```

Obtains the current user-defined callback function and associated user handle.

Parameters

- `task` (*`MSKtask_t`*) – An optimization task. (input)
- `func` (*`MSKcallbackfunc` by reference*) – Get the user-defined progress callback function *`MSKcallbackfunc`* associated with `task`. If `func` is identical to NULL, then no callback function is associated with the `task`. (output)
- `handle` (*`MSKuserhandle_t` by reference*) – The user-defined pointer associated with the user-defined callback function. (output)

Return (*`MSKrescodee`*) – The function response code.

Groups *Callback*

MSK_getcfix

```
MSKrescodee (MSKAPI MSK_getcfix) (
  MSKtask_t task,
  MSKrealt * cfix)
```

Obtains the fixed term in the objective.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `cfix` (*MSKrealt by reference*) – Fixed term in the objective. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*

`MSK_getcj`

```
MSKrescodee (MSKAPI MSK_getcj) (
  MSKtask_t task,
  MSKint32t j,
  MSKrealt * cj)
```

Obtains one coefficient of c.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `j` (*MSKint32t*) – Index of the variable for which the c coefficient should be obtained. (input)
- `cj` (*MSKrealt by reference*) – The value of c_j. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*, *Problem data - variables*

`MSK_getclist`

```
MSKrescodee (MSKAPI MSK_getclist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subj,
  MSKrealt * c)
```

Obtains a sequence of elements in c.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of variables for which the c values should be obtained. (input)
- `subj` (*MSKint32t**) – A list of variable indexes. (input)
- `c` (*MSKrealt**) – Linear terms of the requested list of the objective as a dense vector. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - linear part*

`MSK_getcodedesc`

```
MSKrescodee (MSKAPI MSK_getcodedesc) (
  MSKrescodee code,
  char * symname,
  char * str)
```

Obtains a short description of the meaning of the response code given by `code`.

Parameters

- `code` (*MSKrescodee*) – A valid **MOSEK** response code. (input)
- `symname` (`char*`) – Symbolic name corresponding to `code`. (output)
- `str` (`char*`) – Obtains a short description of a response code. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Responses, errors and warnings*

MSK_getconbound

```
MSKrescodee (MSKAPI MSK_getconbound) (
  MSKtask_t task,
  MSKint32t i,
  MSKboundkeye * bk,
  MSKrealt * bl,
  MSKrealt * bu)
```

Obtains bound information for one constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Index of the constraint for which the bound information should be obtained. (input)
- bk (*MSKboundkeye by reference*) – Bound keys. (output)
- bl (*MSKrealt by reference*) – Values for lower bounds. (output)
- bu (*MSKrealt by reference*) – Values for upper bounds. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*, *Problem data - bounds*, *Problem data - constraints*

MSK_getconboundslice

```
MSKrescodee (MSKAPI MSK_getconboundslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKboundkeye * bk,
  MSKrealt * bl,
  MSKrealt * bu)
```

Obtains bounds information for a slice of the constraints.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- bk (*MSKboundkeye**) – Bound keys. (output)
- bl (*MSKrealt**) – Values for lower bounds. (output)
- bu (*MSKrealt**) – Values for upper bounds. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*, *Problem data - bounds*, *Problem data - constraints*

~~MSK_getcone~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getcone) (
  MSKtask_t task,
  MSKint32t k,
  MSKconetypee * ct,
```

(continues on next page)

(continued from previous page)

```
  MSKrealt * conepar,
  MSKint32t * nummem,
  MSKint32t * submem)
```

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_t*) – An optimization task. (input)
- `k` (*MSKint32t*) – Index of the cone. (input)
- `ct` (*MSKconetypee by reference*) – Specifies the type of the cone. (output)
- `conepar` (*MSKrealt by reference*) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0. (output)
- `nummem` (*MSKint32t by reference*) – Number of member variables in the cone. (output)
- `submem` (*MSKint32t **) – Variable subscripts of the members in the cone. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - cones (deprecated)*

~~`MSK_getconeinfo`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getconeinfo) (
  MSKtask_t task,
  MSKint32t k,
  MSKconetypee * ct,
  MSKrealt * conepar,
  MSKint32t * nummem)
```

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_t*) – An optimization task. (input)
- `k` (*MSKint32t*) – Index of the cone. (input)
- `ct` (*MSKconetypee by reference*) – Specifies the type of the cone. (output)
- `conepar` (*MSKrealt by reference*) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0. (output)
- `nummem` (*MSKint32t by reference*) – Number of member variables in the cone. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - cones (deprecated)*

~~`MSK_getconename`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getconename) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t sizename,
  char * name)
```

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_t*) – An optimization task. (input)
- i (*MSKint32t*) – Index of the cone. (input)
- sizename (*MSKint32t*) – Maximum length of a name that can be stored in name. (input)
- name (char*) – The required name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - cones (deprecated)*, *Inspecting the task*

~~MSK_getconenameindex~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getconenameindex) (
  MSKtask_t task,
  const char * somename,
  MSKint32t * asgn,
  MSKint32t * index)
```

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_t*) – An optimization task. (input)
- somename (char*) – The name which should be checked. (input)
- asgn (*MSKint32t by reference*) – Is non-zero if the name somename is assigned to some cone. (output)
- index (*MSKint32t by reference*) – If the name somename is assigned to some cone, then index is the index of the cone. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - cones (deprecated)*, *Inspecting the task*

~~MSK_getconenamelen~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getconenamelen) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t * len)
```

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_t*) – An optimization task. (input)
- i (*MSKint32t*) – Index of the cone. (input)
- len (*MSKint32t by reference*) – Returns the length of the indicated name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - cones (deprecated)*, *Inspecting the task*

MSK_getconname

```
MSKrescodee (MSKAPI MSK_getconname) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t sizename,
  char * name)
```

Obtains the name of a constraint.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `i` (*MSKint32t*) – Index of the constraint. (input)
- `sizename` (*MSKint32t*) – Maximum length of name that can be stored in `name`. (input)
- `name` (`char*`) – The required name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - linear part*, *Problem data - constraints*, *Inspecting the task*

`MSK_getconnameindex`

```
MSKrescodee (MSKAPI MSK_getconnameindex) (
  MSKtask_t task,
  const char * somename,
  MSKint32t * asgn,
  MSKint32t * index)
```

Checks whether the name `somename` has been assigned to any constraint. If so, the index of the constraint is reported.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `somename` (`char*`) – The name which should be checked. (input)
- `asgn` (*MSKint32t by reference*) – Is non-zero if the name `somename` is assigned to some constraint. (output)
- `index` (*MSKint32t by reference*) – If the name `somename` is assigned to a constraint, then `index` is the index of the constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - linear part*, *Problem data - constraints*, *Inspecting the task*

`MSK_getconnamelen`

```
MSKrescodee (MSKAPI MSK_getconnamelen) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t * len)
```

Obtains the length of the name of a constraint.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `i` (*MSKint32t*) – Index of the constraint. (input)
- `len` (*MSKint32t by reference*) – Returns the length of the indicated name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - linear part*, *Problem data - constraints*, *Inspecting the task*

MSK_getcslice

```
MSKrescodee (MSKAPI MSK_getcslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * c)
```

Obtains a sequence of elements in c.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- c (*MSKrealt**) – Linear terms of the requested slice of the objective as a dense vector. The length is last-first. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - linear part*

MSK_getdimbarvarj

```
MSKrescodee (MSKAPI MSK_getdimbarvarj) (
  MSKtask_t task,
  MSKint32t j,
  MSKint32t * dimbarvarj)
```

Obtains the dimension of a symmetric matrix variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- j (*MSKint32t*) – Index of the semidefinite variable whose dimension is requested. (input)
- dimbarvarj (*MSKint32t by reference*) – The dimension of the j-th semidefinite variable. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - semidefinite*

MSK_getdjcafeidxlist

```
MSKrescodee (MSKAPI MSK_getdjcafeidxlist) (
  MSKtask_t task,
  MSKint64t djcidx,
  MSKint64t * afeidxlist)
```

Obtains the list of affine expression indexes in a disjunctive constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- djcidx (*MSKint64t*) – Index of the disjunctive constraint. (input)
- afeidxlist (*MSKint64t**) – List of affine expression indexes. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getdjcb

```
MSKrescodee (MSKAPI MSK_getdjcb) (
  MSKtask_t task,
  MSKint64t djcidx,
  MSKrealt * b)
```

Obtains the optional constant term vector of a disjunctive constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- djcidx (*MSKint64t*) – Index of the disjunctive constraint. (input)
- b (*MSKrealt* *) – The vector b. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getdjcdomainidxlist

```
MSKrescodee (MSKAPI MSK_getdjcdomainidxlist) (
  MSKtask_t task,
  MSKint64t djcidx,
  MSKint64t * domidxlist)
```

Obtains the list of domain indexes in a disjunctive constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- djcidx (*MSKint64t*) – Index of the disjunctive constraint. (input)
- domidxlist (*MSKint64t* *) – List of term sizes. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getdjcname

```
MSKrescodee (MSKAPI MSK_getdjcname) (
  MSKtask_t task,
  MSKint64t djcidx,
  MSKint32t sizename,
  char * name)
```

Obtains the name of a disjunctive constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- djcidx (*MSKint64t*) – Index of a disjunctive constraint. (input)
- sizename (*MSKint32t*) – The length of the buffer pointed to by the name argument. (input)
- name (char*) – Returns the required name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getdjcnamelen

```
MSKrescodee (MSKAPI MSK_getdjcnamelen) (
  MSKtask_t task,
  MSKint64t djcidx,
  MSKint32t * len)
```

Obtains the length of the name of a disjunctive constraint.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `djcidx` (*MSKint64t*) – Index of a disjunctive constraint. (input)
- `len` (*MSKint32t by reference*) – Returns the length of the indicated name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - disjunctive constraints*, *Inspecting the task*

`MSK_getdjcnumafe`

```
MSKrescodee (MSKAPI MSK_getdjcnumafe) (
  MSKtask_t task,
  MSKint64t djcidx,
  MSKint64t * numafe)
```

Obtains the number of affine expressions in the disjunctive constraint.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `djcidx` (*MSKint64t*) – Index of the disjunctive constraint. (input)
- `numafe` (*MSKint64t by reference*) – Number of affine expressions in the disjunctive constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

`MSK_getdjcnumafetot`

```
MSKrescodee (MSKAPI MSK_getdjcnumafetot) (
  MSKtask_t task,
  MSKint64t * numafetot)
```

Obtains the total number of affine expressions in all disjunctive constraints.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `numafetot` (*MSKint64t by reference*) – Number of affine expressions in all disjunctive constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

`MSK_getdjcnumdomain`

```
MSKrescodee (MSKAPI MSK_getdjcnumdomain) (
  MSKtask_t task,
  MSKint64t djcidx,
  MSKint64t * numdomain)
```

Obtains the number of domains in the disjunctive constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- djcidx (*MSKint64t*) – Index of the disjunctive constraint. (input)
- numdomain (*MSKint64t by reference*) – Number of domains in the disjunctive constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getdjcnumdomaintot

```
MSKrescodee (MSKAPI MSK_getdjcnumdomaintot) (
  MSKtask_t task,
  MSKint64t * numdomaintot)
```

Obtains the total number of domains in all disjunctive constraints.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numdomaintot (*MSKint64t by reference*) – Number of domains in all disjunctive constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getdjcnumterm

```
MSKrescodee (MSKAPI MSK_getdjcnumterm) (
  MSKtask_t task,
  MSKint64t djcidx,
  MSKint64t * numterm)
```

Obtains the number terms in the disjunctive constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- djcidx (*MSKint64t*) – Index of the disjunctive constraint. (input)
- numterm (*MSKint64t by reference*) – Number of terms in the disjunctive constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getdjcnumtermtot

```
MSKrescodee (MSKAPI MSK_getdjcnumtermtot) (
  MSKtask_t task,
  MSKint64t * numtermtot)
```

Obtains the total number of terms in all disjunctive constraints.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numtermtot (*MSKint64t by reference*) – Total number of terms in all disjunctive constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getdjcs

```
MSKrescodee (MSKAPI MSK_getdjcs) (
  MSKtask_t task,
  MSKint64t * domidxlist,
  MSKint64t * afeidxlist,
  MSKrealt * b,
  MSKint64t * termsizelist,
  MSKint64t * numterms)
```

Obtains full data of all disjunctive constraints. The output arrays must have minimal lengths determined by the following methods: `domainidxlist` by *MSK_getdjcnumdomaintot*, `afeidxlist` and `b` by *MSK_getdjcnumafetot*, `termsizelist` by *MSK_getdjcnumtermtot* and `numterms` by *MSK_getnumdomain*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `domidxlist` (*MSKint64t* *) – The concatenation of index lists of domains appearing in all disjunctive constraints. (output)
- `afeidxlist` (*MSKint64t* *) – The concatenation of index lists of affine expressions appearing in all disjunctive constraints. (output)
- `b` (*MSKrealt* *) – The concatenation of vectors b appearing in all disjunctive constraints. (output)
- `termsizelist` (*MSKint64t* *) – The concatenation of lists of term sizes appearing in all disjunctive constraints. (output)
- `numterms` (*MSKint64t* *) – The number of terms in each of the disjunctive constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getdjctermsizelist

```
MSKrescodee (MSKAPI MSK_getdjctermsizelist) (
  MSKtask_t task,
  MSKint64t djcidx,
  MSKint64t * termsizelist)
```

Obtains the list of term sizes in a disjunctive constraint.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `djcidx` (*MSKint64t*) – Index of the disjunctive constraint. (input)
- `termsizelist` (*MSKint64t* *) – List of term sizes. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getdomainn

```
MSKrescodee (MSKAPI MSK_getdomainn) (
  MSKtask_t task,
  MSKint64t domidx,
  MSKint64t * n)
```

Obtains the dimension of the domain.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `domidx` (*MSKint64t*) – Index of the domain. (input)
- `n` (*MSKint64t by reference*) – Dimension of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*, *Inspecting the task*

MSK_getdomainname

```
MSKrescodee (MSKAPI MSK_getdomainname) (
  MSKtask_t task,
  MSKint64t domidx,
  MSKint32t sizename,
  char * name)
```

Obtains the name of a domain.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `domidx` (*MSKint64t*) – Index of a domain. (input)
- `sizename` (*MSKint32t*) – The length of the buffer pointed to by the `name` argument. (input)
- `name` (`char*`) – Returns the required name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - domain*, *Inspecting the task*

MSK_getdomainnamelen

```
MSKrescodee (MSKAPI MSK_getdomainnamelen) (
  MSKtask_t task,
  MSKint64t domidx,
  MSKint32t * len)
```

Obtains the length of the name of a domain.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `domidx` (*MSKint64t*) – Index of a domain. (input)
- `len` (*MSKint32t by reference*) – Returns the length of the indicated name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - domain*, *Inspecting the task*

MSK_getdomaintype

```
MSKrescodee (MSKAPI MSK_getdomaintype) (
  MSKtask_t task,
  MSKint64t domidx,
  MSKdomaintypee * domtype)
```

Returns the type of the domain.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `domidx` (*MSKint64t*) – Index of the domain. (input)
- `domtype` (*MSKdomaintypee by reference*) – The type of the domain. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*, *Inspecting the task*

MSK_getdouinf

```
MSKrescodee (MSKAPI MSK_getdouinf) (
  MSKtask_t task,
  MSKdinfiteme whichdinf,
  MSKrealt * dvalue)
```

Obtains a double information item from the task information database.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichdinf (*MSKdinfiteme*) – Specifies a double information item. (input)
- dvalue (*MSKrealt by reference*) – The value of the required double information item. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Information items and statistics*

MSK_getdouparam

```
MSKrescodee (MSKAPI MSK_getdouparam) (
  MSKtask_t task,
  MSKdparame param,
  MSKrealt * parvalue)
```

Obtains the value of a double parameter.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- param (*MSKdparame*) – Which parameter. (input)
- parvalue (*MSKrealt by reference*) – Parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_getdualobj

```
MSKrescodee (MSKAPI MSK_getdualobj) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * dualobj)
```

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_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- dualobj (*MSKrealt by reference*) – Objective value corresponding to the dual solution. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Solution - dual*

MSK_getdualsolutionnorms

```
MSKrescodee (MSKAPI MSK_getdualsolutionnorms) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * nrmy,
  MSKrealt * nrmslc,
  MSKrealt * nrmsuc,
  MSKrealt * nrmslx,
  MSKrealt * nrmsux,
  MSKrealt * nrmsnx,
  MSKrealt * nrmbars)
```

Compute norms of the dual solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- nrmy (*MSKrealt by reference*) – The norm of the y vector. (output)
- nrmslc (*MSKrealt by reference*) – The norm of the s_l^c vector. (output)
- nrmsuc (*MSKrealt by reference*) – The norm of the s_u^c vector. (output)
- nrmslx (*MSKrealt by reference*) – The norm of the s_l^x vector. (output)
- nrmsux (*MSKrealt by reference*) – The norm of the s_u^x vector. (output)
- nrmsnx (*MSKrealt by reference*) – The norm of the s_n^x vector. (output)
- nrmbars (*MSKrealt by reference*) – The norm of the $\overline{S}$ vector. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getdviolacc

```
MSKrescodee (MSKAPI MSK_getdviolacc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint64t numaccidx,
  const MSKint64t * accidxlist,
  MSKrealt * viol)
```

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_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- numaccidx (*MSKint64t*) – Length of sub and viol. (input)
- accidxlist (*MSKint64t* *) – An array of indexes of conic constraints. (input)
- viol (*MSKrealt* *) – viol[k] is the violation of the dual solution associated with the conic constraint sub[k]. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getdviolbarvar

```
MSKrescodee (MSKAPI MSK_getdviolbarvar) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t num,
  const MSKint32t * sub,
  MSKrealt * viol)
```

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_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `num` (*MSKint32t*) – Length of `sub` and `viol`. (input)
- `sub` (*MSKint32t* *) – An array of indexes of $\overline{X}$ variables. (input)
- `viol` (*MSKrealt* *) – `viol[k]` is the violation of the solution for the constraint $\overline{S}_{\texttt{sub}[k]} \in \mathcal{S}_+$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getdviolcon

```
MSKrescodee (MSKAPI MSK_getdviolcon) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t num,
  const MSKint32t * sub,
  MSKrealt * viol)
```

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_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `num` (*MSKint32t*) – Length of `sub` and `viol`. (input)
- `sub` (*MSKint32t* *) – An array of indexes of constraints. (input)
- `viol` (*MSKrealt* *) – `viol[k]` is the violation of dual solution associated with the constraint `sub[k]`. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

~~MSK_getdviolcones~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getdviolcones) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t num,
  const MSKint32t * sub,
  MSKrealt * viol)
```

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

$$\left\{ \begin{array}{ll} \max(0, (\|s_n^x\|_{2:n}^* - (s_n^x)_1^*)/\sqrt{2}, & (s_n^x)^* \geq -\|(s_n^x)_{2:n}^*\|, \\ \|(s_n^x)^*\|, & \text{otherwise.} \end{array} \right.$$

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_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `num` (*MSKint32t*) – Length of `sub` and `viol`. (input)
- `sub` (*MSKint32t* *) – An array of indexes of conic constraints. (input)
- `viol` (*MSKrealt* *) – `viol[k]` is the violation of the dual solution associated with the conic constraint `sub[k]`. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getdviolvar

```
MSKrescodee (MSKAPI MSK_getdviolvar) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t num,
  const MSKint32t * sub,
  MSKrealt * viol)
```

The violation of the dual solution associated with the j-th variable is computed as follows

$$\max \left(\rho((s_l^x)_j^*, (b_l^x)_j),\ \rho((s_u^x)_j^*, -(b_u^x)_j),\ |\sum_{i=0}^{numcon-1} a_{ij} y_i + (s_l^x)_j^* - (s_u^x)_j^* - \tau c_j| \right)$$

where

$$\rho(x, l) = \left\{ \begin{array}{ll} -x, & l > -\infty, \\ |x|, & \text{otherwise} \end{array} \right.$$

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_t*) – An optimization task. (input)

- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- num (*MSKint32t*) – Length of sub and viol. (input)
- sub (*MSKint32t**) – An array of indexes of x variables. (input)
- viol (*MSKrealt**) – viol[k] is the violation of dual solution associated with the variable sub[k]. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getenv

```
MSKrescodee (MSKAPI MSK_getenv) (
  MSKtask_t task,
  MSKenv_t * env)
```

Obtains the environment used to create the task.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- env (*MSKenv_t by reference*) – The MOSEK environment. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*

MSK_getinfeasiblesubproblem

```
MSKrescodee (MSKAPI MSK_getinfeasiblesubproblem) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKtask_t * inftask)
```

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_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Which solution to use when determining the infeasible subproblem. (input)
- inftask (*MSKtask_t by reference*) – A new task containing the infeasible subproblem. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Infeasibility diagnostic*

MSK_getinfindex

```
MSKrescodee (MSKAPI MSK_getinfindex) (
  MSKtask_t task,
  MSKinftypee inftype,
  const char * infname,
  MSKint32t * infindex)
```

Obtains the index of a named information item.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- inftype (*MSKinftypee*) – Type of the information item. (input)
- infname (char*) – Name of the information item. (input)
- infindex (*MSKint32t by reference*) – The item index. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Information items and statistics*

MSK_getinfmax

```
MSKrescodee (MSKAPI MSK_getinfmax) (
  MSKtask_t task,
  MSKinftypee inftype,
  MSKint32t * infmax)
```

Obtains the maximum index of an information item of a given type inftype plus 1.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- inftype (*MSKinftypee*) – Type of the information item. (input)
- infmax (*MSKint32t**) – The maximum index (plus 1) requested. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Information items and statistics*

MSK_getinfname

```
MSKrescodee (MSKAPI MSK_getinfname) (
  MSKtask_t task,
  MSKinftypee inftype,
  MSKint32t whichinf,
  char * infname)
```

Obtains the name of an information item.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- inftype (*MSKinftypee*) – Type of the information item. (input)
- whichinf (*MSKint32t*) – An information item. (input)
- infname (char*) – Name of the information item. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Information items and statistics*, *Names*

MSK_getintinf

```
MSKrescodee (MSKAPI MSK_getintinf) (
  MSKtask_t task,
  MSKiinfiteme whichiinf,
  MSKint32t * ivalue)
```

Obtains an integer information item from the task information database.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- whichiinf (*MSKiinfiteme*) – Specifies an integer information item. (input)

- `ivalue` (*MSKint32t by reference*) – The value of the required integer information item. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Information items and statistics*

`MSK_getintparam`

```
MSKrescodee (MSKAPI MSK_getintparam) (
  MSKtask_t task,
  MSKiparame param,
  MSKint32t * parvalue)
```

Obtains the value of an integer parameter.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `param` (*MSKiparame*) – Which parameter. (input)
- `parvalue` (*MSKint32t by reference*) – Parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

`MSK_getlasterror`

```
MSKrescodee (MSKAPI MSK_getlasterror) (
  MSKtask_t task,
  MSKrescodee * lastrescode,
  MSKint32t sizelastmsg,
  MSKint32t * lastmsglen,
  char * lastmsg)
```

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_t*) – An optimization task. (input)
- `lastrescode` (*MSKrescodee by reference*) – Returns the last error code reported in the task. (output)
- `sizelastmsg` (*MSKint32t*) – The length of the `lastmsg` buffer. (input)
- `lastmsglen` (*MSKint32t by reference*) – Returns the length of the last error message reported in the task. (output)
- `lastmsg` (`char*`) – Returns the last error message reported in the task. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Responses, errors and warnings*

`MSK_getlasterror64`

```
MSKrescodee (MSKAPI MSK_getlasterror64) (
  MSKtask_t task,
  MSKrescodee * lastrescode,
  MSKint64t sizelastmsg,
  MSKint64t * lastmsglen,
  char * lastmsg)
```

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_t*) – An optimization task. (input)
- `lastrescode` (*MSKrescodee by reference*) – Returns the last error code reported in the task. (output)
- `sizelastmsg` (*MSKint64t*) – The length of the `lastmsg` buffer. (input)
- `lastmsglen` (*MSKint64t by reference*) – Returns the length of the last error message reported in the task. (output)
- `lastmsg` (`char*`) – Returns the last error message reported in the task. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Responses, errors and warnings*

`MSK_getlenbarvarj`

```
MSKrescodee (MSKAPI MSK_getlenbarvarj) (
  MSKtask_t task,
  MSKint32t j,
  MSKint64t * lenbarvarj)
```

Obtains the length of the j-th semidefinite variable i.e. the number of elements in the lower triangular part.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `j` (*MSKint32t*) – Index of the semidefinite variable whose length if requested. (input)
- `lenbarvarj` (*MSKint64t by reference*) – Number of scalar elements in the lower triangular part of the semidefinite variable. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - semidefinite*

`MSK_getlintinf`

```
MSKrescodee (MSKAPI MSK_getlintinf) (
  MSKtask_t task,
  MSKliinfiteme whichliinf,
  MSKint64t * ivalue)
```

Obtains a long integer information item from the task information database.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichliinf` (*MSKliinfiteme*) – Specifies a long information item. (input)
- `ivalue` (*MSKint64t by reference*) – The value of the required long integer information item. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Information items and statistics*

`MSK_getmaxnamelen`

```
MSKrescodee (MSKAPI MSK_getmaxnamelen) (
  MSKtask_t task,
  MSKint32t * maxlen)
```

Obtains the maximum length (not including terminating zero character) of any objective, constraint, variable, domain or cone name.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- maxlen (*MSKint32t by reference*) – The maximum length of any name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Names*

MSK_getmaxnumanz

```
MSKrescodee (MSKAPI MSK_getmaxnumanz) (
  MSKtask_t task,
  MSKint32t * maxnumanz)
```

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_t*) – An optimization task. (input)
- maxnumanz (*MSKint32t by reference*) – Number of preallocated non-zero linear matrix elements. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - linear part*

MSK_getmaxnumanz64

```
MSKrescodee (MSKAPI MSK_getmaxnumanz64) (
  MSKtask_t task,
  MSKint64t * maxnumanz)
```

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_t*) – An optimization task. (input)
- maxnumanz (*MSKint64t by reference*) – Number of preallocated non-zero linear matrix elements. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - linear part*

MSK_getmaxnumbarvar

```
MSKrescodee (MSKAPI MSK_getmaxnumbarvar) (
  MSKtask_t task,
  MSKint32t * maxnumbarvar)
```

Obtains maximum number of symmetric matrix variables for which space is currently preallocated.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- maxnumbarvar (*MSKint32t by reference*) – Maximum number of symmetric matrix variables for which space is currently preallocated. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - semidefinite*

MSK_getmaxnumcon

```
MSKrescodee (MSKAPI MSK_getmaxnumcon) (
  MSKtask_t task,
  MSKint32t * maxnumcon)
```

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_t*) – An optimization task. (input)
- maxnumcon (*MSKint32t by reference*) – Number of preallocated constraints in the optimization task. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - linear part*, *Problem data - constraints*

~~MSK_getmaxnumcone~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getmaxnumcone) (
  MSKtask_t task,
  MSKint32t * maxnumcone)
```

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_t*) – An optimization task. (input)
- maxnumcone (*MSKint32t by reference*) – Number of preallocated conic constraints in the optimization task. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - cones (deprecated)*

MSK_getmaxnumqnz

```
MSKrescodee (MSKAPI MSK_getmaxnumqnz) (
  MSKtask_t task,
  MSKint32t * maxnumqnz)
```

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_t*) – An optimization task. (input)
- maxnumqnz (*MSKint32t by reference*) – Number of non-zero elements preallocated in quadratic coefficient matrices. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - quadratic part*

MSK_getmaxnumqnz64

```
MSKrescodee (MSKAPI MSK_getmaxnumqnz64) (
  MSKtask_t task,
  MSKint64t * maxnumqnz)
```

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_t*) – An optimization task. (input)
- `maxnumqnz` (*MSKint64t by reference*) – Number of non-zero elements preallocated in quadratic coefficient matrices. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - quadratic part*

`MSK_getmaxnumvar`

```
MSKrescodee (MSKAPI MSK_getmaxnumvar) (
  MSKtask_t task,
  MSKint32t * maxnumvar)
```

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_t*) – An optimization task. (input)
- `maxnumvar` (*MSKint32t by reference*) – Number of preallocated variables in the optimization task. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - linear part*, *Problem data - variables*

`MSK_getmemusagetask`

```
MSKrescodee (MSKAPI MSK_getmemusagetask) (
  MSKtask_t task,
  MSKint64t * meminuse,
  MSKint64t * maxmemuse)
```

Obtains information about the amount of memory used by a task.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `meminuse` (*MSKint64t by reference*) – Amount of memory currently used by the `task`. (output)
- `maxmemuse` (*MSKint64t by reference*) – Maximum amount of memory used by the `task` until now. (output)

Return (*MSKrescodee*) – The function response code.

Groups *System, memory and debugging*

`MSK_getnadouinf`

```
MSKrescodee (MSKAPI MSK_getnadouinf) (
  MSKtask_t task,
  const char * infitemname,
  MSKrealt * dvalue)
```

Obtains a named double information item from task information database.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- infitemname (char*) – The name of a double information item. (input)
- dvalue (*MSKrealt by reference*) – The value of the required double information item. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Information items and statistics*

MSK_getnadouparam

```
MSKrescodee (MSKAPI MSK_getnadouparam) (
  MSKtask_t task,
  const char * paramname,
  MSKrealt * parvalue)
```

Obtains the value of a named double parameter.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- paramname (char*) – Name of a parameter. (input)
- parvalue (*MSKrealt by reference*) – Parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_getnaintinf

```
MSKrescodee (MSKAPI MSK_getnaintinf) (
  MSKtask_t task,
  const char * infitemname,
  MSKint32t * ivalue)
```

Obtains a named integer information item from the task information database.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- infitemname (char*) – The name of an integer information item. (input)
- ivalue (*MSKint32t by reference*) – The value of the required integer information item. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Information items and statistics*

MSK_getnaintparam

```
MSKrescodee (MSKAPI MSK_getnaintparam) (
  MSKtask_t task,
  const char * paramname,
  MSKint32t * parvalue)
```

Obtains the value of a named integer parameter.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- paramname (char*) – Name of a parameter. (input)
- parvalue (*MSKint32t by reference*) – Parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_getnastrparam

```
MSKrescodee (MSKAPI MSK_getnastrparam) (
  MSKtask_t task,
  const char * paramname,
  MSKint32t sizeparamname,
  MSKint32t * len,
  char * parvalue)
```

Obtains the value of a named string parameter.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `paramname` (`char*`) – Name of a parameter. (input)
- `sizeparamname` (*MSKint32t*) – Size of the name buffer `parvalue`. (input)
- `len` (*MSKint32t by reference*) – Length of the string in `parvalue`. (output)
- `parvalue` (`char*`) – Parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*, *Names*

MSK_getnastrparamal

```
MSKrescodee (MSKAPI MSK_getnastrparamal) (
  MSKtask_t task,
  const char * paramname,
  MSKint32t numaddchr,
  char ** value)
```

Obtains the value of a named string parameter.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `paramname` (`char*`) – Name of a parameter. (input)
- `numaddchr` (*MSKint32t*) – Number of additional characters for which room is left in `value`. (input)
- `value` (`char*` *by reference*) – Parameter value. **MOSEK** will allocate this char buffer of size equal to the actual length of the string parameter plus `numaddchr`. This memory must be freed by *MSK_freetask*. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_getnumacc

```
MSKrescodee (MSKAPI MSK_getnumacc) (
  MSKtask_t task,
  MSKint64t * num)
```

Obtains the number of affine conic constraints.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint64t by reference*) – The number of affine conic constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*, *Inspecting the task*

MSK_getnumafe

```
MSKrescodee (MSKAPI MSK_getnumafe) (
  MSKtask_t task,
  MSKint64t * numafe)
```

Obtains the number of affine expressions.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numafe (*MSKint64t by reference*) – Number of affine expressions. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Inspecting the task*

MSK_getnumanz

```
MSKrescodee (MSKAPI MSK_getnumanz) (
  MSKtask_t task,
  MSKint32t * numanz)
```

Obtains the number of non-zeros in A.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numanz (*MSKint32t by reference*) – Number of non-zero elements in the linear constraint matrix. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - linear part*

MSK_getnumanz64

```
MSKrescodee (MSKAPI MSK_getnumanz64) (
  MSKtask_t task,
  MSKint64t * numanz)
```

Obtains the number of non-zeros in A.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numanz (*MSKint64t by reference*) – Number of non-zero elements in the linear constraint matrix. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - linear part*

MSK_getnumbarablocktriplets

```
MSKrescodee (MSKAPI MSK_getnumbarablocktriplets) (
  MSKtask_t task,
  MSKint64t * num)
```

Obtains an upper bound on the number of elements in the block triplet form of $\overline{A}$.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint64t by reference*) – An upper bound on the number of elements in the block triplet form of $\overline{A}$. (output)

Return (*MSKrescodee*) – The function response code.
Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getnumbaranz

```
MSKrescodee (MSKAPI MSK_getnumbaranz) (
  MSKtask_t task,
  MSKint64t * nz)
```

Get the number of nonzero elements in $\overline{A}$.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- nz (*MSKint64t by reference*) – The number of nonzero block elements in $\overline{A}$ i.e. the number of $\overline{A}_{ij}$ elements that are nonzero. (output)

Return (*MSKrescodee*) – The function response code.
Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getnumbarcblocktriplets

```
MSKrescodee (MSKAPI MSK_getnumbarcblocktriplets) (
  MSKtask_t task,
  MSKint64t * num)
```

Obtains an upper bound on the number of elements in the block triplet form of $\overline{C}$.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint64t by reference*) – An upper bound on the number of elements in the block triplet form of $\overline{C}$. (output)

Return (*MSKrescodee*) – The function response code.
Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getnumbarcnz

```
MSKrescodee (MSKAPI MSK_getnumbarcnz) (
  MSKtask_t task,
  MSKint64t * nz)
```

Obtains the number of nonzero elements in $\overline{C}$.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- nz (*MSKint64t by reference*) – The number of nonzeros in $\overline{C}$ i.e. the number of elements $\overline{C}_j$ that are nonzero. (output)

Return (*MSKrescodee*) – The function response code.
Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getnumbarvar

```
MSKrescodee (MSKAPI MSK_getnumbarvar) (
  MSKtask_t task,
  MSKint32t * numbarvar)
```

Obtains the number of semidefinite variables.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numbarvar (*MSKint32t by reference*) – Number of semidefinite variables in the problem. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - semidefinite*

MSK_getnumcon

```
MSKrescodee (MSKAPI MSK_getnumcon) (
  MSKtask_t task,
  MSKint32t * numcon)
```

Obtains the number of constraints.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numcon (*MSKint32t by reference*) – Number of constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - constraints*, *Inspecting the task*

~~MSK_getnumcone~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getnumcone) (
  MSKtask_t task,
  MSKint32t * numcone)
```

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_t*) – An optimization task. (input)
- numcone (*MSKint32t by reference*) – Number of conic constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - cones (deprecated)*, *Inspecting the task*

~~MSK_getnumconemem~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getnumconemem) (
  MSKtask_t task,
  MSKint32t k,
  MSKint32t * nummem)
```

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_t*) – An optimization task. (input)
- k (*MSKint32t*) – Index of the cone. (input)
- nummem (*MSKint32t by reference*) – Number of member variables in the cone. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - cones (deprecated)*, *Inspecting the task*

MSK_getnumdjc

```
MSKrescodee (MSKAPI MSK_getnumdjc) (
  MSKtask_t task,
  MSKint64t * num)
```

Obtains the number of disjunctive constraints.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint64t by reference*) – The number of disjunctive constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*, *Inspecting the task*

MSK_getnumdomain

```
MSKrescodee (MSKAPI MSK_getnumdomain) (
  MSKtask_t task,
  MSKint64t * numdomain)
```

Obtain the number of domains defined.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- numdomain (*MSKint64t by reference*) – Number of domains in the task. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - domain*

MSK_getnumintvar

```
MSKrescodee (MSKAPI MSK_getnumintvar) (
  MSKtask_t task,
  MSKint32t * numintvar)
```

Obtains the number of integer-constrained variables.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- numintvar (*MSKint32t by reference*) – Number of integer variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - variables*

MSK_getnumparam

```
MSKrescodee (MSKAPI MSK_getnumparam) (
  MSKtask_t task,
  MSKparametertypee partype,
  MSKint32t * numparam)
```

Obtains the number of parameters of a given type.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- partype (*MSKparametertypee*) – Parameter type. (input)
- numparam (*MSKint32t by reference*) – The number of parameters of type partype. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Parameters*

MSK_getnumqconknz

```
MSKrescodee (MSKAPI MSK_getnumqconknz) (
  MSKtask_t task,
  MSKint32t k,
  MSKint32t * numqcnz)
```

Obtains the number of non-zero quadratic terms in a constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- k (*MSKint32t*) – Index of the constraint for which the number of non-zero quadratic terms should be obtained. (input)
- numqcnz (*MSKint32t by reference*) – Number of quadratic terms. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - constraints*, *Problem data - quadratic part*

MSK_getnumqconknz64

```
MSKrescodee (MSKAPI MSK_getnumqconknz64) (
  MSKtask_t task,
  MSKint32t k,
  MSKint64t * numqcnz)
```

Obtains the number of non-zero quadratic terms in a constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- k (*MSKint32t*) – Index of the constraint for which the number quadratic terms should be obtained. (input)
- numqcnz (*MSKint64t by reference*) – Number of quadratic terms. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - constraints*, *Problem data - quadratic part*

MSK_getnumqobjnz

```
MSKrescodee (MSKAPI MSK_getnumqobjnz) (
  MSKtask_t task,
  MSKint32t * numqonz)
```

Obtains the number of non-zero quadratic terms in the objective.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numqonz (*MSKint32t by reference*) – Number of non-zero elements in the quadratic objective terms. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - quadratic part*

MSK_getnumqobjnz64

```
MSKrescodee (MSKAPI MSK_getnumqobjnz64) (
  MSKtask_t task,
  MSKint64t * numqonz)
```

Obtains the number of non-zero quadratic terms in the objective.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `numqonz` (*MSKint64t by reference*) – Number of non-zero elements in the quadratic objective terms. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - quadratic part*

`MSK_getnumsymmat`

```
MSKrescodee (MSKAPI MSK_getnumsymmat) (
  MSKtask_t task,
  MSKint64t * num)
```

Obtains the number of symmetric matrices stored in the vector E.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint64t by reference*) – The number of symmetric sparse matrices. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

`MSK_getnumvar`

```
MSKrescodee (MSKAPI MSK_getnumvar) (
  MSKtask_t task,
  MSKint32t * numvar)
```

Obtains the number of variables.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `numvar` (*MSKint32t by reference*) – Number of variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - variables*

`MSK_getobjname`

```
MSKrescodee (MSKAPI MSK_getobjname) (
  MSKtask_t task,
  MSKint32t sizeobjname,
  char * objname)
```

Obtains the name assigned to the objective function.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `sizeobjname` (*MSKint32t*) – Length of `objname`. (input)
- `objname` (`char*`) – Assigned the objective name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Names*

`MSK_getobjnamelen`

```
MSKrescodee (MSKAPI MSK_getobjnamelen) (
  MSKtask_t task,
  MSKint32t * len)
```

Obtains the length of the name assigned to the objective function.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- len (*MSKint32t by reference*) – Assigned the length of the objective name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Names*

MSK_getobjsense

```
MSKrescodee (MSKAPI MSK_getobjsense) (
  MSKtask_t task,
  MSKobjsensee * sense)
```

Gets the objective sense of the task.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- sense (*MSKobjsensee by reference*) – The returned objective sense. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

MSK_getparammax

```
MSKrescodee (MSKAPI MSK_getparammax) (
  MSKtask_t task,
  MSKparametertypee partype,
  MSKint32t * parammax)
```

Obtains the maximum index of a parameter of type partype plus 1.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- partype (*MSKparametertypee*) – Parameter type. (input)
- parammax (*MSKint32t by reference*) – The maximum index (plus 1) of the given parameter type. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_getparamname

```
MSKrescodee (MSKAPI MSK_getparamname) (
  MSKtask_t task,
  MSKparametertypee partype,
  MSKint32t param,
  char * parname)
```

Obtains the name for a parameter param of type partype.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)

- partype (*MSKparametertypee*) – Parameter type. (input)
- param (*MSKint32t*) – Which parameter. (input)
- parname (char*) – Parameter name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Parameters*

MSK_getpowerdomainalpha

```
MSKrescodee (MSKAPI MSK_getpowerdomainalpha) (
  MSKtask_t task,
  MSKint64t domidx,
  MSKrealt * alpha)
```

Obtains the exponent vector α of a primal or dual power cone domain.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- domidx (*MSKint64t*) – Index of the domain. (input)
- alpha (*MSKrealt**) – The vector α. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*, *Inspecting the task*

MSK_getpowerdomaininfo

```
MSKrescodee (MSKAPI MSK_getpowerdomaininfo) (
  MSKtask_t task,
  MSKint64t domidx,
  MSKint64t * n,
  MSKint64t * nleft)
```

Obtains structural information about a primal or dual power cone domain.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- domidx (*MSKint64t*) – Index of the domain. (input)
- n (*MSKint64t by reference*) – Dimension of the domain. (output)
- nleft (*MSKint64t by reference*) – Number of variables on the left hand side. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - domain*, *Inspecting the task*

MSK_getprimalobj

```
MSKrescodee (MSKAPI MSK_getprimalobj) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * primalobj)
```

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_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- primalobj (*MSKrealt by reference*) – Objective value corresponding to the primal solution. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Solution - primal*

MSK_getprimalsolutionnorms

```
MSKrescodee (MSKAPI MSK_getprimalsolutionnorms) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * nrmxc,
  MSKrealt * nrmxx,
  MSKrealt * nrmbarx)
```

Compute norms of the primal solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- nrmxc (*MSKrealt by reference*) – The norm of the x^c vector. (output)
- nrmxx (*MSKrealt by reference*) – The norm of the x vector. (output)
- nrmbarx (*MSKrealt by reference*) – The norm of the $\overline{X}$ vector. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getprobtype

```
MSKrescodee (MSKAPI MSK_getprobtype) (
  MSKtask_t task,
  MSKproblemtypee * probtype)
```

Obtains the problem type.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- probtype (*MSKproblemtypee by reference*) – The problem type. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*

MSK_getprosta

```
MSKrescodee (MSKAPI MSK_getprosta) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKprostae * problemsta)
```

Obtains the problem status.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- problemsta (*MSKprostae by reference*) – Problem status. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getpviolacc

```
MSKrescodee (MSKAPI MSK_getpviolacc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint64t numaccidx,
  const MSKint64t * accidxlist,
  MSKrealt * viol)
```

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_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- numaccidx (*MSKint64t*) – Length of sub and viol. (input)
- accidxlist (*MSKint64t**) – An array of indexes of conic constraints. (input)
- viol (*MSKrealt**) – viol[k] is the violation of the solution associated with the affine conic constraint number accidxlist[k]. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getpviolbarvar

```
MSKrescodee (MSKAPI MSK_getpviolbarvar) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t num,
  const MSKint32t * sub,
  MSKrealt * viol)
```

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_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- num (*MSKint32t*) – Length of sub and viol. (input)
- sub (*MSKint32t**) – An array of indexes of $\overline{X}$ variables. (input)
- viol (*MSKrealt**) – viol[k] is how much the solution violates the constraint $\overline{X}_{\mathtt{sub}[k]} \in \mathcal{S}_+$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getpviolcon

```
MSKrescodee (MSKAPI MSK_getpviolcon) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t num,
  const MSKint32t * sub,
  MSKrealt * viol)
```

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

$$\max(\tau l_i^c - (x_i^c)^*,\ (x_i^c)^* - \tau u_i^c),\ |\sum_{j=0}^{numvar-1} a_{ij}x_j^* - x_i^c|)$$

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_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- num (*MSKint32t*) – Length of sub and viol. (input)
- sub (*MSKint32t* *) – An array of indexes of constraints. (input)
- viol (*MSKrealt* *) – viol[k] is the violation associated with the solution for the constraint sub[k]. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

~~MSK_getpviolcones~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getpviolcones) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t num,
  const MSKint32t * sub,
  MSKrealt * viol)
```

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_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- num (*MSKint32t*) – Length of sub and viol. (input)
- sub (*MSKint32t* *) – An array of indexes of conic constraints. (input)
- viol (*MSKrealt* *) – viol[k] is the violation of the solution associated with the conic constraint number sub[k]. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getpvioldjc

```
MSKrescodee (MSKAPI MSK_getpvioldjc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint64t numdjcidx,
  const MSKint64t * djcidxlist,
  MSKrealt * viol)
```

Computes the primal solution violation for a set of disjunctive constraints. For a single DJC the violation is defined as

$$\operatorname{viol}\left(\bigvee_{i=1}^{t}\bigwedge_{j=1}^{s_i} T_{i,j}\right) = \min_{i=1,\ldots,t}\left(\max_{j=1,\ldots,s_j}(\operatorname{viol}(T_{i,j}))\right)$$

where the violation of each simple term $T_{i,j}$ is defined as for an ordinary linear constraint.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `numdjcidx` (*MSKint64t*) – Length of `sub` and `viol`. (input)
- `djcidxlist` (*MSKint64t* *) – An array of indexes of disjunctive constraints. (input)
- `viol` (*MSKrealt* *) – `viol[k]` is the violation of the solution associated with the disjunctive constraint number `djcidxlist[k]`. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getpviolvar

```
MSKrescodee (MSKAPI MSK_getpviolvar) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t num,
  const MSKint32t * sub,
  MSKrealt * viol)
```

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_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `num` (*MSKint32t*) – Length of `sub` and `viol`. (input)
- `sub` (*MSKint32t* *) – An array of indexes of x variables. (input)
- `viol` (*MSKrealt* *) – `viol[k]` is the violation associated with the solution for the variable $x_{\texttt{sub[k]}}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getqconk

```
MSKrescodee (MSKAPI MSK_getqconk) (
  MSKtask_t task,
  MSKint32t k,
  MSKint32t maxnumqcnz,
  MSKint32t * numqcnz,
  MSKint32t * qcsubi,
  MSKint32t * qcsubj,
  MSKrealt * qcval)
```

Obtains all the quadratic terms in a constraint. The quadratic terms are stored sequentially in qcsubi, qcsubj, and qcval.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- k (*MSKint32t*) – Which constraint. (input)
- maxnumqcnz (*MSKint32t*) – Length of the arrays qcsubi, qcsubj, and qcval. (input)
- numqcnz (*MSKint32t by reference*) – Number of quadratic terms. (output)
- qcsubi (*MSKint32t* *) – Row subscripts for quadratic constraint matrix. (output)
- qcsubj (*MSKint32t* *) – Column subscripts for quadratic constraint matrix. (output)
- qcval (*MSKrealt* *) – Quadratic constraint coefficient values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - quadratic part*, *Problem data - constraints*

MSK_getqconk64

```
MSKrescodee (MSKAPI MSK_getqconk64) (
  MSKtask_t task,
  MSKint32t k,
  MSKint64t maxnumqcnz,
  MSKint64t * numqcnz,
  MSKint32t * qcsubi,
  MSKint32t * qcsubj,
  MSKrealt * qcval)
```

Obtains all the quadratic terms in a constraint. The quadratic terms are stored sequentially in qcsubi, qcsubj, and qcval.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- k (*MSKint32t*) – Which constraint. (input)
- maxnumqcnz (*MSKint64t*) – Length of the arrays qcsubi, qcsubj, and qcval. (input)
- numqcnz (*MSKint64t by reference*) – Number of quadratic terms. (output)
- qcsubi (*MSKint32t* *) – Row subscripts for quadratic constraint matrix. (output)
- qcsubj (*MSKint32t* *) – Column subscripts for quadratic constraint matrix. (output)
- qcval (*MSKrealt* *) – Quadratic constraint coefficient values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - quadratic part*, *Problem data - constraints*

MSK_getqobj

```
MSKrescodee (MSKAPI MSK_getqobj) (
  MSKtask_t task,
  MSKint32t maxnumqonz,
  MSKint32t * numqonz,
  MSKint32t * qosubi,
  MSKint32t * qosubj,
  MSKrealt * qoval)
```

Obtains the quadratic terms in the objective. The required quadratic terms are stored sequentially in `qosubi`, `qosubj`, and `qoval`.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `maxnumqonz` (*MSKint32t*) – The length of the arrays `qosubi`, `qosubj`, and `qoval`. (input)
- `numqonz` (*MSKint32t by reference*) – Number of non-zero elements in the quadratic objective terms. (output)
- `qosubi` (*MSKint32t* *) – Row subscripts for quadratic objective coefficients. (output)
- `qosubj` (*MSKint32t* *) – Column subscripts for quadratic objective coefficients. (output)
- `qoval` (*MSKrealt* *) – Quadratic objective coefficient values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - quadratic part*

MSK_getqobj64

```
MSKrescodee (MSKAPI MSK_getqobj64) (
  MSKtask_t task,
  MSKint64t maxnumqonz,
  MSKint64t * numqonz,
  MSKint32t * qosubi,
  MSKint32t * qosubj,
  MSKrealt * qoval)
```

Obtains the quadratic terms in the objective. The required quadratic terms are stored sequentially in `qosubi`, `qosubj`, and `qoval`.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `maxnumqonz` (*MSKint64t*) – The length of the arrays `qosubi`, `qosubj`, and `qoval`. (input)
- `numqonz` (*MSKint64t by reference*) – Number of non-zero elements in the quadratic objective terms. (output)
- `qosubi` (*MSKint32t* *) – Row subscripts for quadratic objective coefficients. (output)
- `qosubj` (*MSKint32t* *) – Column subscripts for quadratic objective coefficients. (output)
- `qoval` (*MSKrealt* *) – Quadratic objective coefficient values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - quadratic part*

MSK_getqobjij

```
MSKrescodee (MSKAPI MSK_getqobjij) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t j,
  MSKrealt * qoij)
```

Obtains one coefficient q_{ij}^o in the quadratic term of the objective.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Row index of the coefficient. (input)
- j (*MSKint32t*) – Column index of coefficient. (input)
- qoij (*MSKrealt by reference*) – The required coefficient. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - quadratic part*

MSK_getreducedcosts

```
MSKrescodee (MSKAPI MSK_getreducedcosts) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * redcosts)
```

Computes the reduced costs for a slice of variables and returns them in the array redcosts i.e.

$$\texttt{redcosts}[j-\texttt{first}] = (s_l^x)_j - (s_u^x)_j, \; j = \texttt{first}, \ldots, \texttt{last}-1 \tag{15.2}$$

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – The index of the first variable in the sequence. (input)
- last (*MSKint32t*) – The index of the last variable in the sequence plus 1. (input)
- redcosts (*MSKrealt**) – The reduced costs for the required slice of variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getresponseclass

```
MSKrescodee (MSKAPI MSK_getresponseclass) (
  MSKrescodee r,
  MSKrescodetypee * rc)
```

Obtain the class of a response code.

Parameters

- r (*MSKrescodee*) – A response code indicating the result of function call. (input)
- rc (*MSKrescodetypee by reference*) – The response class. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Responses, errors and warnings*

MSK_getskc

```
MSKrescodee (MSKAPI MSK_getskc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKstakeye * skc)
```

Obtains the status keys for the constraints.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- skc (*MSKstakeye**) – Status keys for the constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getskcslice

```
MSKrescodee (MSKAPI MSK_getskcslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKstakeye * skc)
```

Obtains the status keys for a slice of the constraints.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- skc (*MSKstakeye**) – Status keys for the constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getskn

```
MSKrescodee (MSKAPI MSK_getskn) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKstakeye * skn)
```

Obtains the status keys for the conic constraints.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- skn (*MSKstakeye**) – Status keys for the conic constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getskx

```
MSKrescodee (MSKAPI MSK_getskx) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKstakeye * skx)
```

Obtains the status keys for the scalar variables.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `skx` (*MSKstakeye**) – Status keys for the variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getskxslice

```
MSKrescodee (MSKAPI MSK_getskxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKstakeye * skx)
```

Obtains the status keys for a slice of the scalar variables.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – First index in the sequence. (input)
- `last` (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- `skx` (*MSKstakeye**) – Status keys for the variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getslc

```
MSKrescodee (MSKAPI MSK_getslc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * slc)
```

Obtains the s_l^c vector for a solution.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `slc` (*MSKrealt**) – Dual variables corresponding to the lower bounds on the constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getslcslice

```
MSKrescodee (MSKAPI MSK_getslcslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * slc)
```

Obtains a slice of the s_l^c vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- slc (*MSKrealt**) – Dual variables corresponding to the lower bounds on the constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getslx

```
MSKrescodee (MSKAPI MSK_getslx) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * slx)
```

Obtains the s_l^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- slx (*MSKrealt**) – Dual variables corresponding to the lower bounds on the variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getslxslice

```
MSKrescodee (MSKAPI MSK_getslxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * slx)
```

Obtains a slice of the s_l^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- slx (*MSKrealt**) – Dual variables corresponding to the lower bounds on the variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getsnx

```
MSKrescodee (MSKAPI MSK_getsnx) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * snx)
```

Obtains the s_n^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- snx (*MSKrealt* *) – Dual variables corresponding to the conic constraints on the variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getsnxslice

```
MSKrescodee (MSKAPI MSK_getsnxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * snx)
```

Obtains a slice of the s_n^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- snx (*MSKrealt* *) – Dual variables corresponding to the conic constraints on the variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getsolsta

```
MSKrescodee (MSKAPI MSK_getsolsta) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKsolstae * solutionsta)
```

Obtains the solution status.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- solutionsta (*MSKsolstae by reference*) – Solution status. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getsolution

```
MSKrescodee (MSKAPI MSK_getsolution) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKprostae * problemsta,
  MSKsolstae * solutionsta,
  MSKstakeye * skc,
  MSKstakeye * skx,
  MSKstakeye * skn,
  MSKrealt * xc,
  MSKrealt * xx,
  MSKrealt * y,
  MSKrealt * slc,
  MSKrealt * suc,
  MSKrealt * slx,
  MSKrealt * sux,
  MSKrealt * snx)
```

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

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

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:

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

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 *MSK_getbarxj* and *MSK_getbarsj*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `problemsta` (*MSKprostae by reference*) – Problem status. (output)
- `solutionsta` (*MSKsolstae by reference*) – Solution status. (output)
- `skc` (*MSKstakeye**) – Status keys for the constraints. (output)
- `skx` (*MSKstakeye**) – Status keys for the variables. (output)
- `skn` (*MSKstakeye**) – Status keys for the conic constraints. (output)
- `xc` (*MSKrealt**) – Primal constraint solution. (output)
- `xx` (*MSKrealt**) – Primal variable solution. (output)
- `y` (*MSKrealt**) – Vector of dual variables corresponding to the constraints. (output)
- `slc` (*MSKrealt**) – Dual variables corresponding to the lower bounds on the constraints. (output)
- `suc` (*MSKrealt**) – Dual variables corresponding to the upper bounds on the constraints. (output)
- `slx` (*MSKrealt**) – Dual variables corresponding to the lower bounds on the variables. (output)
- `sux` (*MSKrealt**) – Dual variables corresponding to the upper bounds on the variables. (output)
- `snx` (*MSKrealt**) – Dual variables corresponding to the conic constraints on the variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Solution - primal*, *Solution - dual*

MSK_getsolutioninfo

```
MSKrescodee (MSKAPI MSK_getsolutioninfo) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * pobj,
  MSKrealt * pviolcon,
  MSKrealt * pviolvar,
  MSKrealt * pviolbarvar,
  MSKrealt * pviolcone,
  MSKrealt * pviolitg,
  MSKrealt * dobj,
  MSKrealt * dviolcon,
  MSKrealt * dviolvar,
  MSKrealt * dviolbarvar,
  MSKrealt * dviolcone)
```

Obtains information about a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- pobj (*MSKrealt by reference*) – The primal objective value as computed by *MSK_getprimalobj*. (output)
- pviolcon (*MSKrealt by reference*) – Maximal primal violation of the solution associated with the x^c variables where the violations are computed by *MSK_getpviolcon*. (output)
- pviolvar (*MSKrealt by reference*) – Maximal primal violation of the solution for the x variables where the violations are computed by *MSK_getpviolvar*. (output)
- pviolbarvar (*MSKrealt by reference*) – Maximal primal violation of solution for the $\overline{X}$ variables where the violations are computed by *MSK_getpviolbarvar*. (output)
- pviolcone (*MSKrealt by reference*) – Maximal primal violation of solution for the conic constraints where the violations are computed by ~~*MSK_getpviolcones*~~. (output)
- pviolitg (*MSKrealt by reference*) – 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. (output)
- dobj (*MSKrealt by reference*) – Dual objective value as computed by *MSK_getdualobj*. (output)
- dviolcon (*MSKrealt by reference*) – Maximal violation of the dual solution associated with the x^c variable as computed by *MSK_getdviolcon*. (output)
- dviolvar (*MSKrealt by reference*) – Maximal violation of the dual solution associated with the x variable as computed by *MSK_getdviolvar*. (output)
- dviolbarvar (*MSKrealt by reference*) – Maximal violation of the dual solution associated with the $\overline{S}$ variable as computed by *MSK_getdviolbarvar*. (output)
- dviolcone (*MSKrealt by reference*) – Maximal violation of the dual solution associated with the dual conic constraints as computed by ~~*MSK_getdviolcones*~~. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getsolutioninfonew

```
MSKrescodee (MSKAPI MSK_getsolutioninfonew) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * pobj,
  MSKrealt * pviolcon,
  MSKrealt * pviolvar,
  MSKrealt * pviolbarvar,
  MSKrealt * pviolcone,
  MSKrealt * pviolacc,
  MSKrealt * pvioldjc,
  MSKrealt * pviolitg,
  MSKrealt * dobj,
  MSKrealt * dviolcon,
  MSKrealt * dviolvar,
  MSKrealt * dviolbarvar,
  MSKrealt * dviolcone,
  MSKrealt * dviolacc)
```

Obtains information about a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- pobj (*MSKrealt by reference*) – The primal objective value as computed by *MSK_getprimalobj*. (output)
- pviolcon (*MSKrealt by reference*) – Maximal primal violation of the solution associated with the x^c variables where the violations are computed by *MSK_getpviolcon*. (output)
- pviolvar (*MSKrealt by reference*) – Maximal primal violation of the solution for the x variables where the violations are computed by *MSK_getpviolvar*. (output)
- pviolbarvar (*MSKrealt by reference*) – Maximal primal violation of solution for the $\overline{X}$ variables where the violations are computed by *MSK_getpviolbarvar*. (output)
- pviolcone (*MSKrealt by reference*) – Maximal primal violation of solution for the conic constraints where the violations are computed by ~~*MSK_getpviolcones*~~. (output)
- pviolacc (*MSKrealt by reference*) – Maximal primal violation of solution for the affine conic constraints where the violations are computed by *MSK_getpviolacc*. (output)
- pvioldjc (*MSKrealt by reference*) – Maximal primal violation of solution for the disjunctive constraints where the violations are computed by *MSK_getpvioldjc*. (output)
- pviolitg (*MSKrealt by reference*) – 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. (output)
- dobj (*MSKrealt by reference*) – Dual objective value as computed by *MSK_getdualobj*. (output)
- dviolcon (*MSKrealt by reference*) – Maximal violation of the dual solution associated with the x^c variable as computed by *MSK_getdviolcon*. (output)
- dviolvar (*MSKrealt by reference*) – Maximal violation of the dual solution associated with the x variable as computed by *MSK_getdviolvar*. (output)
- dviolbarvar (*MSKrealt by reference*) – Maximal violation of the dual solution associated with the $\overline{S}$ variable as computed by *MSK_getdviolbarvar*. (output)
- dviolcone (*MSKrealt by reference*) – Maximal violation of the dual solution associated with the dual conic constraints as computed by ~~*MSK_getdviolcones*~~. (output)
- dviolacc (*MSKrealt by reference*) – Maximal violation of the dual solution associated with the affine conic constraints as computed by *MSK_getdviolacc*. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getsolutionnew

```
MSKrescodee (MSKAPI MSK_getsolutionnew) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKprostae * problemsta,
  MSKsolstae * solutionsta,
  MSKstakeye * skc,
  MSKstakeye * skx,
  MSKstakeye * skn,
```

(continues on next page)

(continued from previous page)

```
  MSKrealt * xc,
  MSKrealt * xx,
  MSKrealt * y,
  MSKrealt * slc,
  MSKrealt * suc,
  MSKrealt * slx,
  MSKrealt * sux,
  MSKrealt * snx,
  MSKrealt * doty)
```

Obtains the complete solution. See *MSK_getsolution* for further information.

In order to retrieve the primal and dual values of semidefinite variables see *MSK_getbarxj* and *MSK_getbarsj*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- problemsta (*MSKprostae by reference*) – Problem status. (output)
- solutionsta (*MSKsolstae by reference*) – Solution status. (output)
- skc (*MSKstakeye* *) – Status keys for the constraints. (output)
- skx (*MSKstakeye* *) – Status keys for the variables. (output)
- skn (*MSKstakeye* *) – Status keys for the conic constraints. (output)
- xc (*MSKrealt* *) – Primal constraint solution. (output)
- xx (*MSKrealt* *) – Primal variable solution. (output)
- y (*MSKrealt* *) – Vector of dual variables corresponding to the constraints. (output)
- slc (*MSKrealt* *) – Dual variables corresponding to the lower bounds on the constraints. (output)
- suc (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the constraints. (output)
- slx (*MSKrealt* *) – Dual variables corresponding to the lower bounds on the variables. (output)
- sux (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the variables. (output)
- snx (*MSKrealt* *) – Dual variables corresponding to the conic constraints on the variables. (output)
- doty (*MSKrealt* *) – Dual variables corresponding to affine conic constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Solution - primal*, *Solution - dual*

MSK_getsolutionslice

```
MSKrescodee (MSKAPI MSK_getsolutionslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKsoliteme solitem,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * values)
```

Obtains a slice of one item from the solution. The format of the solution is exactly as in *MSK_getsolution*. The parameter solitem determines which of the solution vectors should be returned.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- solitem (*MSKsoliteme*) – Which part of the solution is required. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- values (*MSKrealt* *) – The values in the required sequence are stored sequentially in values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*, *Solution - dual*, *Solution information*

MSK_getsparsesymmat

```
MSKrescodee (MSKAPI MSK_getsparsesymmat) (
  MSKtask_t task,
  MSKint64t idx,
  MSKint64t maxlen,
  MSKint32t * subi,
  MSKint32t * subj,
  MSKrealt * valij)
```

Get a single symmetric matrix from the matrix store.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- idx (*MSKint64t*) – Index of the matrix to retrieve. (input)
- maxlen (*MSKint64t*) – Length of the output arrays subi, subj and valij. (input)
- subi (*MSKint32t* *) – Row subscripts of the matrix non-zero elements. (output)
- subj (*MSKint32t* *) – Column subscripts of the matrix non-zero elements. (output)
- valij (*MSKrealt* *) – Coefficients of the matrix non-zero elements. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_getstrparam

```
MSKrescodee (MSKAPI MSK_getstrparam) (
  MSKtask_t task,
  MSKsparame param,
  MSKint32t maxlen,
  MSKint32t * len,
  char * parvalue)
```

Obtains the value of a string parameter.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- param (*MSKsparame*) – Which parameter. (input)
- maxlen (*MSKint32t*) – Length of the parvalue buffer. (input)
- len (*MSKint32t by reference*) – The length of the parameter value. (output)
- parvalue (char*) – Parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Parameters*

MSK_getstrparamal

```
MSKrescodee (MSKAPI MSK_getstrparamal) (
  MSKtask_t task,
  MSKsparame param,
  MSKint32t numaddchr,
  char ** value)
```

Obtains the value of a string parameter.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- param (*MSKsparame*) – Which parameter. (input)
- numaddchr (*MSKint32t*) – Number of additional characters for which room is left in value. (input)
- value (char* *by reference*) – Parameter value. **MOSEK** will allocate this char buffer of size equal to the actual length of the string parameter plus numaddchr. This memory must be freed by *MSK_freetask*. (input/output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_getstrparamlen

```
MSKrescodee (MSKAPI MSK_getstrparamlen) (
  MSKtask_t task,
  MSKsparame param,
  MSKint32t * len)
```

Obtains the length of a string parameter.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- param (*MSKsparame*) – Which parameter. (input)
- len (*MSKint32t by reference*) – The length of the parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Parameters*

MSK_getsuc

```
MSKrescodee (MSKAPI MSK_getsuc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * suc)
```

Obtains the s_u^c vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- suc (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getsucslice

```
MSKrescodee (MSKAPI MSK_getsucslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * suc)
```

Obtains a slice of the s_u^c vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- suc (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getsux

```
MSKrescodee (MSKAPI MSK_getsux) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * sux)
```

Obtains the s_u^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- sux (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getsuxslice

```
MSKrescodee (MSKAPI MSK_getsuxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * sux)
```

Obtains a slice of the s_u^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- sux (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the variables. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_getsymbcon

```
MSKrescodee (MSKAPI MSK_getsymbcon) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t sizevalue,
  char * name,
  MSKint32t * value)
```

Obtains the name and corresponding value for the ith symbolic constant.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Index. (input)
- sizevalue (*MSKint32t*) – The length of the buffer pointed to by the value argument. (input)
- name (char*) – Name of the ith symbolic constant. (output)
- value (*MSKint32t by reference*) – The corresponding value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

MSK_getsymbcondim

```
MSKrescodee (MSKAPI MSK_getsymbcondim) (
  MSKenv_t env,
  MSKint32t * num,
  size_t * maxlen)
```

Obtains the number of symbolic constants defined by **MOSEK** and the maximum length of the name of any symbolic constant.

Parameters

- env (*MSKenv_t*) – The MOSEK environment. (input)
- num (*MSKint32t by reference*) – Number of symbolic constants defined by **MOSEK**. (output)
- maxlen (size_t *by reference*) – Maximum length of the name of any symbolic constants. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_getsymmatinfo

```
MSKrescodee (MSKAPI MSK_getsymmatinfo) (
  MSKtask_t task,
  MSKint64t idx,
  MSKint32t * dim,
  MSKint64t * nz,
  MSKsymmattypee * mattype)
```

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_t*) – An optimization task. (input)

- idx (*MSKint64t*) – Index of the matrix for which information is requested. (input)
- dim (*MSKint32t by reference*) – Returns the dimension of the requested matrix. (output)
- nz (*MSKint64t by reference*) – Returns the number of non-zeros in the requested matrix. (output)
- mattype (*MSKsymmattypee by reference*) – Returns the type of the requested matrix. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Inspecting the task*

MSK_gettaskname

```
MSKrescodee (MSKAPI MSK_gettaskname) (
  MSKtask_t task,
  MSKint32t sizetaskname,
  char * taskname)
```

Obtains the name assigned to the task.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- sizetaskname (*MSKint32t*) – Length of the taskname buffer. (input)
- taskname (char*) – Returns the task name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Inspecting the task*

MSK_gettasknamelen

```
MSKrescodee (MSKAPI MSK_gettasknamelen) (
  MSKtask_t task,
  MSKint32t * len)
```

Obtains the length the task name.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- len (*MSKint32t by reference*) – Returns the length of the task name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Inspecting the task*

MSK_getvarbound

```
MSKrescodee (MSKAPI MSK_getvarbound) (
  MSKtask_t task,
  MSKint32t i,
  MSKboundkeye * bk,
  MSKrealt * bl,
  MSKrealt * bu)
```

Obtains bound information for one variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Index of the variable for which the bound information should be obtained. (input)

- bk (*MSKboundkeye by reference*) – Bound keys. (output)
- bl (*MSKrealt by reference*) – Values for lower bounds. (output)
- bu (*MSKrealt by reference*) – Values for upper bounds. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*, *Problem data - bounds*, *Problem data - variables*

MSK_getvarboundslice

```
MSKrescodee (MSKAPI MSK_getvarboundslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKboundkeye * bk,
  MSKrealt * bl,
  MSKrealt * bu)
```

Obtains bounds information for a slice of the variables.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- bk (*MSKboundkeye**) – Bound keys. (output)
- bl (*MSKrealt**) – Values for lower bounds. (output)
- bu (*MSKrealt**) – Values for upper bounds. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Inspecting the task*, *Problem data - bounds*, *Problem data - variables*

MSK_getvarname

```
MSKrescodee (MSKAPI MSK_getvarname) (
  MSKtask_t task,
  MSKint32t j,
  MSKint32t sizename,
  char * name)
```

Obtains the name of a variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- j (*MSKint32t*) – Index of a variable. (input)
- sizename (*MSKint32t*) – The length of the buffer pointed to by the name argument. (input)
- name (char*) – Returns the required name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - linear part*, *Problem data - variables*, *Inspecting the task*

MSK_getvarnameindex

```
MSKrescodee (MSKAPI MSK_getvarnameindex) (
  MSKtask_t task,
  const char * somename,
  MSKint32t * asgn,
  MSKint32t * index)
```

Checks whether the name `somename` has been assigned to any variable. If so, the index of the variable is reported.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `somename` (`char*`) – The name which should be checked. (input)
- `asgn` (*MSKint32t by reference*) – Is non-zero if the name `somename` is assigned to a variable. (output)
- `index` (*MSKint32t by reference*) – If the name `somename` is assigned to a variable, then `index` is the index of the variable. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - linear part*, *Problem data - variables*, *Inspecting the task*

`MSK_getvarnamelen`

```
MSKrescodee (MSKAPI MSK_getvarnamelen) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t * len)
```

Obtains the length of the name of a variable.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `i` (*MSKint32t*) – Index of a variable. (input)
- `len` (*MSKint32t by reference*) – Returns the length of the indicated name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - linear part*, *Problem data - variables*, *Inspecting the task*

`MSK_getvartype`

```
MSKrescodee (MSKAPI MSK_getvartype) (
  MSKtask_t task,
  MSKint32t j,
  MSKvariabletypee * vartype)
```

Gets the variable type of one variable.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `j` (*MSKint32t*) – Index of the variable. (input)
- `vartype` (*MSKvariabletypee by reference*) – Variable type of the j-th variable. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - variables*

MSK_getvartypelist

```
MSKrescodee (MSKAPI MSK_getvartypelist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subj,
  MSKvariabletypee * vartype)
```

Obtains the variable type of one or more variables. Upon return `vartype[k]` is the variable type of variable `subj[k]`.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of variables for which the variable type should be obtained. (input)
- `subj` (*MSKint32t**) – A list of variable indexes. (input)
- `vartype` (*MSKvariabletypee**) – The variables types corresponding to the variables specified by `subj`. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Problem data - variables*

MSK_getversion

```
MSKrescodee (MSKAPI MSK_getversion) (
  MSKint32t * major,
  MSKint32t * minor,
  MSKint32t * revision)
```

Obtains **MOSEK** version information.

Parameters

- `major` (*MSKint32t by reference*) – Major version number. (output)
- `minor` (*MSKint32t by reference*) – Minor version number. (output)
- `revision` (*MSKint32t by reference*) – Revision number. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Versions*

MSK_getxc

```
MSKrescodee (MSKAPI MSK_getxc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * xc)
```

Obtains the x^c vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `xc` (*MSKrealt**) – Primal constraint solution. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*

MSK_getxcslice

```
MSKrescodee (MSKAPI MSK_getxcslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * xc)
```

Obtains a slice of the x^c vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – First index in the sequence. (input)
- `last` (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- `xc` (*MSKrealt* *) – Primal constraint solution. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*

`MSK_getxx`

```
MSKrescodee (MSKAPI MSK_getxx) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * xx)
```

Obtains the x^x vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `xx` (*MSKrealt* *) – Primal variable solution. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*

`MSK_getxxslice`

```
MSKrescodee (MSKAPI MSK_getxxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * xx)
```

Obtains a slice of the x^x vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – First index in the sequence. (input)
- `last` (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- `xx` (*MSKrealt* *) – Primal variable solution. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*

`MSK_gety`

```
MSKrescodee (MSKAPI MSK_gety) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * y)
```

Obtains the y vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `y` (*MSKrealt**) – Vector of dual variables corresponding to the constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

`MSK_getyslice`

```
MSKrescodee (MSKAPI MSK_getyslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  MSKrealt * y)
```

Obtains a slice of the y vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – First index in the sequence. (input)
- `last` (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- `y` (*MSKrealt**) – Vector of dual variables corresponding to the constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

`MSK_infeasibilityreport`

```
MSKrescodee (MSKAPI MSK_infeasibilityreport) (
  MSKtask_t task,
  MSKstreamtypee whichstream,
  MSKsoltypee whichsol)
```

Prints the infeasibility report to an output stream.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichstream` (*MSKstreamtypee*) – Index of the stream. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Infeasibility diagnostic*

`MSK_initbasissolve`

```
MSKrescodee (MSKAPI MSK_initbasissolve) (
  MSKtask_t task,
  MSKint32t * basis)
```

Prepare a task for use with the *MSK_solvewithbasis* function.

This function should be called

- immediately before the first call to *MSK_solvewithbasis*, and
- immediately before any subsequent call to *MSK_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_t*) – An optimization task. (input)
- basis (*MSKint32t* *) – The array of basis indexes to use. The array is interpreted as follows: If $\mathtt{basis}[i] \leq numcon - 1$, then $x^c_{\mathtt{basis}[i]}$ is in the basis at position i, otherwise $x_{\mathtt{basis}[i]-\mathtt{numcon}}$ is in the basis at position i. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solving systems with basis matrix*

MSK_inputdata

```
MSKrescodee (MSKAPI MSK_inputdata) (
  MSKtask_t task,
  MSKint32t maxnumcon,
  MSKint32t maxnumvar,
  MSKint32t numcon,
  MSKint32t numvar,
  const MSKrealt * c,
  MSKrealt cfix,
  const MSKint32t * aptrb,
  const MSKint32t * aptre,
  const MSKint32t * asub,
  const MSKrealt * aval,
  const MSKboundkeye * bkc,
  const MSKrealt * blc,
  const MSKrealt * buc,
  const MSKboundkeye * bkx,
  const MSKrealt * blx,
  const MSKrealt * bux)
```

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_t*) – An optimization task. (input)
- maxnumcon (*MSKint32t*) – Number of preallocated constraints in the optimization task. (input)
- maxnumvar (*MSKint32t*) – Number of preallocated variables in the optimization task. (input)
- numcon (*MSKint32t*) – Number of constraints. (input)
- numvar (*MSKint32t*) – Number of variables. (input)
- c (*MSKrealt* *) – Linear terms of the objective as a dense vector. The length is the number of variables. (input)

- cfix (*MSKrealt*) – Fixed term in the objective. (input)
- aptrb (*MSKint32t* *) – Row or column start pointers. (input)
- aptre (*MSKint32t* *) – Row or column end pointers. (input)
- asub (*MSKint32t* *) – Coefficient subscripts. (input)
- aval (*MSKrealt* *) – Coefficient values. (input)
- bkc (*MSKboundkeye* *) – Bound keys for the constraints. (input)
- blc (*MSKrealt* *) – Lower bounds for the constraints. (input)
- buc (*MSKrealt* *) – Upper bounds for the constraints. (input)
- bkx (*MSKboundkeye* *) – Bound keys for the variables. (input)
- blx (*MSKrealt* *) – Lower bounds for the variables. (input)
- bux (*MSKrealt* *) – Upper bounds for the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - bounds*, *Problem data - constraints*

MSK_inputdata64

```
MSKrescodee (MSKAPI MSK_inputdata64) (
  MSKtask_t task,
  MSKint32t maxnumcon,
  MSKint32t maxnumvar,
  MSKint32t numcon,
  MSKint32t numvar,
  const MSKrealt * c,
  MSKrealt cfix,
  const MSKint64t * aptrb,
  const MSKint64t * aptre,
  const MSKint32t * asub,
  const MSKrealt * aval,
  const MSKboundkeye * bkc,
  const MSKrealt * blc,
  const MSKrealt * buc,
  const MSKboundkeye * bkx,
  const MSKrealt * blx,
  const MSKrealt * bux)
```

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_t*) – An optimization task. (input)
- maxnumcon (*MSKint32t*) – Number of preallocated constraints in the optimization task. (input)
- maxnumvar (*MSKint32t*) – Number of preallocated variables in the optimization task. (input)
- numcon (*MSKint32t*) – Number of constraints. (input)
- numvar (*MSKint32t*) – Number of variables. (input)
- c (*MSKrealt* *) – Linear terms of the objective as a dense vector. The length is the number of variables. (input)
- cfix (*MSKrealt*) – Fixed term in the objective. (input)
- aptrb (*MSKint64t* *) – Row or column start pointers. (input)
- aptre (*MSKint64t* *) – Row or column end pointers. (input)
- asub (*MSKint32t* *) – Coefficient subscripts. (input)

- aval (*MSKrealt**) – Coefficient values. (input)
- bkc (*MSKboundkeye**) – Bound keys for the constraints. (input)
- blc (*MSKrealt**) – Lower bounds for the constraints. (input)
- buc (*MSKrealt**) – Upper bounds for the constraints. (input)
- bkx (*MSKboundkeye**) – Bound keys for the variables. (input)
- blx (*MSKrealt**) – Lower bounds for the variables. (input)
- bux (*MSKrealt**) – Upper bounds for the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - bounds*, *Problem data - constraints*

MSK_iparvaltosymnam

```
MSKrescodee (MSKAPI MSK_iparvaltosymnam) (
  MSKenv_t env,
  MSKiparame whichparam,
  MSKint32t whichvalue,
  char * symbolicname)
```

Obtains the symbolic name corresponding to a value that can be assigned to an integer parameter.

Parameters
- env (*MSKenv_t*) – The MOSEK environment. (input)
- whichparam (*MSKiparame*) – Which parameter. (input)
- whichvalue (*MSKint32t*) – Which value. (input)
- symbolicname (char*) – The symbolic name corresponding to whichvalue. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*, *Names*

MSK_isdouparname

```
MSKrescodee (MSKAPI MSK_isdouparname) (
  MSKtask_t task,
  const char * parname,
  MSKdparame * param)
```

Checks whether parname is a valid double parameter name.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- parname (char*) – Parameter name. (input)
- param (*MSKdparame by reference*) – Returns the parameter corresponding to the name, if one exists. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*, *Names*

MSK_isinfinity

```
MSKbooleant (MSKAPI MSK_isinfinity) (
  MSKrealt value)
```

Return true if value is considered infinity by **MOSEK**.

Parameters value (*MSKrealt*) – The value to be checked (input)

Return (*MSKbooleant*) – True if the value represents infinity.

MSK_isintparname

```
MSKrescodee (MSKAPI MSK_isintparname) (
  MSKtask_t task,
  const char * parname,
  MSKiparame * param)
```

Checks whether `parname` is a valid integer parameter name.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `parname` (`char*`) – Parameter name. (input)
- `param` (*MSKiparame by reference*) – Returns the parameter corresponding to the name, if one exists. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*, *Names*

MSK_isstrparname

```
MSKrescodee (MSKAPI MSK_isstrparname) (
  MSKtask_t task,
  const char * parname,
  MSKsparame * param)
```

Checks whether `parname` is a valid string parameter name.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `parname` (`char*`) – Parameter name. (input)
- `param` (*MSKsparame by reference*) – Returns the parameter corresponding to the name, if one exists. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*, *Names*

MSK_licensecleanup

```
MSKrescodee (MSKAPI MSK_licensecleanup) (
void)
```

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.

Return (*MSKrescodee*) – The function response code.

Groups *License system*

MSK_linkfiletoenvstream

```
MSKrescodee (MSKAPI MSK_linkfiletoenvstream) (
  MSKenv_t env,
  MSKstreamtypee whichstream,
  const char * filename,
  MSKint32t append)
```

Sends all output from the stream defined by `whichstream` to the file given by `filename`.

Parameters

- env (*MSKenv_t*) – The MOSEK environment. (input)
- whichstream (*MSKstreamtypee*) – Index of the stream. (input)
- filename (char*) – A valid file name. (input)
- append (*MSKint32t*) – If this argument is 0 the file will be overwritten, otherwise it will be appended to. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*

MSK_linkfiletotaskstream

```
MSKrescodee (MSKAPI MSK_linkfiletotaskstream) (
  MSKtask_t task,
  MSKstreamtypee whichstream,
  const char * filename,
  MSKint32t append)
```

Directs all output from a task stream whichstream to a file filename.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichstream (*MSKstreamtypee*) – Index of the stream. (input)
- filename (char*) – A valid file name. (input)
- append (*MSKint32t*) – If this argument is 0 the output file will be overwritten, otherwise it will be appended to. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*

MSK_linkfunctoenvstream

```
MSKrescodee (MSKAPI MSK_linkfunctoenvstream) (
  MSKenv_t env,
  MSKstreamtypee whichstream,
  MSKuserhandle_t handle,
  MSKstreamfunc func)
```

Connects a user-defined function to a stream.

Parameters

- env (*MSKenv_t*) – The MOSEK environment. (input)
- whichstream (*MSKstreamtypee*) – Index of the stream. (input)
- handle (*MSKuserhandle_t*) – A user-defined handle which is passed to the user-defined function func. (input)
- func (*MSKstreamfunc*) – All output to the stream whichstream is passed to func. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*, *Callback*

MSK_linkfunctotaskstream

```
MSKrescodee (MSKAPI MSK_linkfunctotaskstream) (
  MSKtask_t task,
  MSKstreamtypee whichstream,
  MSKuserhandle_t handle,
  MSKstreamfunc func)
```

Connects a user-defined function to a task stream.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- whichstream (*MSKstreamtypee*) – Index of the stream. (input)
- handle (*MSKuserhandle_t*) – A user-defined handle which is passed to the user-defined function func. (input)
- func (*MSKstreamfunc*) – All output to the stream whichstream is passed to func. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*, *Callback*

MSK_makeemptytask

```
MSKrescodee (MSKAPI MSK_makeemptytask) (
  MSKenv_t env,
  MSKtask_t * task)
```

Creates a new optimization task.

Parameters
- env (*MSKenv_t*) – The MOSEK environment. (input)
- task (*MSKtask_t by reference*) – An optimization task. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*

MSK_makeenv

```
MSKrescodee (MSKAPI MSK_makeenv) (
  MSKenv_t * env,
  const char * dbgfile)
```

Creates a new **MOSEK** environment. The environment must be shared among all tasks in a program.

Parameters
- env (*MSKenv_t by reference*) – The MOSEK environment. (output)
- dbgfile (char*) – A user-defined memory debug file. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*

MSK_maketask

```
MSKrescodee (MSKAPI MSK_maketask) (
  MSKenv_t env,
  MSKint32t maxnumcon,
  MSKint32t maxnumvar,
  MSKtask_t * task)
```

Creates a new task.

Parameters
- env (*MSKenv_t*) – The MOSEK environment. (input)
- maxnumcon (*MSKint32t*) – An optional estimate on the maximum number of constraints in the task. Can be 0 if no such estimate is known. (input)
- maxnumvar (*MSKint32t*) – An optional estimate on the maximum number of variables in the task. Can be 0 if no such estimate is known. (input)
- task (*MSKtask_t by reference*) – An optimization task. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*

`MSK_onesolutionsummary`

```
MSKrescodee (MSKAPI MSK_onesolutionsummary) (
  MSKtask_t task,
  MSKstreamtypee whichstream,
  MSKsoltypee whichsol)
```

Prints a short summary of a specified solution.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichstream` (*MSKstreamtypee*) – Index of the stream. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*, *Solution information*

`MSK_optimize`

```
MSKrescodee (MSKAPI MSK_optimize) (
  MSKtask_t task)
```

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*.

Response codes come in three categories:

- Errors: Indicate that an error has occurred during the optimization, e.g the optimizer has run out of memory (*MSK_RES_ERR_SPACE*).
- Warnings: Less fatal than errors. E.g *MSK_RES_WRN_LARGE_CJ* indicating possibly problematic problem data.
- Termination codes: Relaying information about the conditions under which the optimizer terminated. E.g *MSK_RES_TRM_MAX_ITERATIONS* indicates that the optimizer finished because it reached the maximum number of iterations specified by the user.

Parameters `task` (*MSKtask_t*) – An optimization task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Optimization*

`MSK_optimizebatch`

```
MSKrescodee (MSKAPI MSK_optimizebatch) (
  MSKenv_t env,
  MSKbooleant israce,
  MSKrealt maxtime,
  MSKint32t numthreads,
  MSKint64t numtask,
  const MSKtask_t * task,
  MSKrescodee * trmcode,
  MSKrescodee * rcode)
```

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_t*) – The MOSEK environment. (input)
- `israce` (*MSKbooleant*) – If nonzero, then the function is terminated after the first task has been completed. (input)
- `maxtime` (*MSKrealt*) – Time limit for the function: if nonnegative, then the function is terminated after maxtime (seconds) has expired. (input)
- `numthreads` (*MSKint32t*) – Number of threads to be employed. (input)
- `numtask` (*MSKint64t*) – Number of tasks to optimize. Length of the `task` array. (input)
- `task` (*MSKtask_t**) – An array of tasks to optimize in parallel. (input)
- `trmcode` (*MSKrescodee**) – The termination code for each task. (output)
- `rcode` (*MSKrescodee**) – The response code for each task. (output)

Return (*MSKrescodee*)

Groups *Optimization*

MSK_optimizermt

```
MSKrescodee (MSKAPI MSK_optimizermt) (
  MSKtask_t task,
  const char * address,
  const char * accesstoken,
  MSKrescodee * trmcode)
```

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_t*) – An optimization task. (input)
- `address` (`char*`) – Address of the OptServer. (input)
- `accesstoken` (`char*`) – Access token. (input)
- `trmcode` (*MSKrescodee by reference*) – Is either *MSK_RES_OK* or a termination response code. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Remote optimization*

MSK_optimizersummary

```
MSKrescodee (MSKAPI MSK_optimizersummary) (
  MSKtask_t task,
  MSKstreamtypee whichstream)
```

Prints a short summary with optimizer statistics from last optimization.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichstream` (*MSKstreamtypee*) – Index of the stream. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*

`MSK_optimizetrm`

```
MSKrescodee (MSKAPI MSK_optimizetrm) (
  MSKtask_t task,
  MSKrescodee * trmcode)
```

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*.

This function is equivalent to *MSK_optimize* except for the handling of return values. This function returns errors on the left hand side. Warnings are not returned and termination codes are returned through the separate argument `trmcode`.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `trmcode` (*MSKrescodee by reference*) – Is either *MSK_RES_OK* or a termination response code. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Optimization*

`MSK_potrf`

```
MSKrescodee (MSKAPI MSK_potrf) (
  MSKenv_t env,
  MSKuploe uplo,
  MSKint32t n,
  MSKrealt * a)
```

Computes a Cholesky factorization of a real symmetric positive definite dense matrix.

Parameters

- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `uplo` (*MSKuploe*) – Indicates whether the upper or lower triangular part of the matrix is stored. (input)
- `n` (*MSKint32t*) – Dimension of the symmetric matrix. (input)
- `a` (*MSKrealt**) – A symmetric matrix stored in column-major order. Only the lower or the upper triangular part is used, accordingly with the `uplo` parameter. It will contain the result on exit. (input/output)

Return (*MSKrescodee*) – The function response code.

Groups *Linear algebra*

`MSK_primalrepair`

```
MSKrescodee (MSKAPI MSK_primalrepair) (
  MSKtask_t task,
  const MSKrealt * wlc,
  const MSKrealt * wuc,
  const MSKrealt * wlx,
  const MSKrealt * wux)
```

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_t*) – An optimization task. (input)
- wlc (*MSKrealt* *) – $(w_l^c)_i$ 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 NULL, then all the weights are assumed to be 1. (input)
- wuc (*MSKrealt* *) – $(w_u^c)_i$ 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 NULL, then all the weights are assumed to be 1. (input)
- wlx (*MSKrealt* *) – $(w_l^x)_j$ 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 NULL, then all the weights are assumed to be 1. (input)
- wux (*MSKrealt* *) – $(w_l^x)_i$ 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 NULL, then all the weights are assumed to be 1. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Infeasibility diagnostic*

MSK_primalsensitivity

```
MSKrescodee (MSKAPI MSK_primalsensitivity) (
  MSKtask_t task,
  MSKint32t numi,
  const MSKint32t * subi,
  const MSKmarke * marki,
  MSKint32t numj,
  const MSKint32t * subj,
  const MSKmarke * markj,
  MSKrealt * leftpricei,
  MSKrealt * rightpricei,
  MSKrealt * leftrangei,
  MSKrealt * rightrangei,
  MSKrealt * leftpricej,
  MSKrealt * rightpricej,
  MSKrealt * leftrangej,
  MSKrealt * rightrangej)
```

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_t*) – An optimization task. (input)
- numi (*MSKint32t*) – Number of bounds on constraints to be analyzed. Length of subi and marki. (input)
- subi (*MSKint32t* *) – Indexes of constraints to analyze. (input)
- marki (*MSKmarke* *) – 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)
- numj (*MSKint32t*) – Number of bounds on variables to be analyzed. Length of subj and markj. (input)
- subj (*MSKint32t* *) – Indexes of variables to analyze. (input)
- markj (*MSKmarke* *) – 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)
- leftpricei (*MSKrealt* *) – leftpricei[i] is the left shadow price for the bound marki[i] of constraint subi[i]. (output)
- rightpricei (*MSKrealt* *) – rightpricei[i] is the right shadow price for the bound marki[i] of constraint subi[i]. (output)
- leftrangei (*MSKrealt* *) – leftrangei[i] is the left range β_1 for the bound marki[i] of constraint subi[i]. (output)
- rightrangei (*MSKrealt* *) – rightrangei[i] is the right range β_2 for the bound marki[i] of constraint subi[i]. (output)
- leftpricej (*MSKrealt* *) – leftpricej[j] is the left shadow price for the bound markj[j] of variable subj[j]. (output)
- rightpricej (*MSKrealt* *) – rightpricej[j] is the right shadow price for the bound markj[j] of variable subj[j]. (output)
- leftrangej (*MSKrealt* *) – leftrangej[j] is the left range β_1 for the bound markj[j] of variable subj[j]. (output)
- rightrangej (*MSKrealt* *) – rightrangej[j] is the right range β_2 for the bound markj[j] of variable subj[j]. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Sensitivity analysis*

MSK_printparam

```
MSKrescodee (MSKAPI MSK_printparam) (
  MSKtask_t task)
```

Prints the current parameter settings to the message stream.

Parameters task (*MSKtask_t*) – An optimization task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Logging*

MSK_probtypetostr

```
MSKrescodee (MSKAPI MSK_probtypetostr) (
  MSKtask_t task,
  MSKproblemtypee probtype,
  char * str)
```

Obtains a string containing the name of a given problem type.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `probtype` (*MSKproblemtypee*) – Problem type. (input)
- `str` (`char*`) – String corresponding to the problem type key `probtype`. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Inspecting the task*, *Names*

`MSK_prostatostr`

```
MSKrescodee (MSKAPI MSK_prostatostr) (
  MSKtask_t task,
  MSKprostae problemsta,
  char * str)
```

Obtains a string containing the name of a given problem status.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `problemsta` (*MSKprostae*) – Problem status. (input)
- `str` (`char*`) – String corresponding to the status key `prosta`. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Names*

`MSK_putacc`

```
MSKrescodee (MSKAPI MSK_putacc) (
  MSKtask_t task,
  MSKint64t accidx,
  MSKint64t domidx,
  MSKint64t numafeidx,
  const MSKint64t * afeidxlist,
  const MSKrealt * b)
```

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 *MSK_appendacc*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `accidx` (*MSKint64t*) – Affine conic constraint index. (input)
- `domidx` (*MSKint64t*) – Domain index. (input)
- `numafeidx` (*MSKint64t*) – Number of affine expressions in the affine expression list (must equal the dimension of the domain). (input)
- `afeidxlist` (*MSKint64t* *) – List of affine expression indexes. (input)
- `b` (*MSKrealt* *) – The vector of constant terms added to affine expressions. Optional, can be NULL. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*

MSK_putaccb

```
MSKrescodee (MSKAPI MSK_putaccb) (
  MSKtask_t task,
  MSKint64t accidx,
  MSKint64t lengthb,
  const MSKrealt * b)
```

Updates an existing affine conic constraint number `accidx` by putting a new vector b.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `accidx` (*MSKint64t*) – Affine conic constraint index. (input)
- `lengthb` (*MSKint64t*) – Length of the vector (must equal the dimension of this constraint). (input)
- `b` (*MSKrealt**) – The vector of constant terms added to affine expressions. Optional, can be NULL. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*

MSK_putaccbj

```
MSKrescodee (MSKAPI MSK_putaccbj) (
  MSKtask_t task,
  MSKint64t accidx,
  MSKint64t j,
  MSKrealt bj)
```

Sets one value $b[j]$ in the b vector for the affine conic constraint number `accidx`.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `accidx` (*MSKint64t*) – Affine conic constraint index. (input)
- `j` (*MSKint64t*) – The index of an element in b to change. (input)
- `bj` (*MSKrealt*) – The new value of $b[j]$. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*

MSK_putaccdoty

```
MSKrescodee (MSKAPI MSK_putaccdoty) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint64t accidx,
  MSKrealt * doty)
```

Puts the $\dot{y}$ vector for a solution (the dual values of an affine conic constraint).

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `accidx` (*MSKint64t*) – The index of the affine conic constraint. (input)
- `doty` (*MSKrealt**) – The dual values for this affine conic constraint. The array should have length equal to the dimension of the constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*, *Problem data - affine conic constraints*

MSK_putacclist

```
MSKrescodee (MSKAPI MSK_putacclist) (
  MSKtask_t task,
  MSKint64t numaccs,
  const MSKint64t * accidxs,
  const MSKint64t * domidxs,
  MSKint64t numafeidx,
  const MSKint64t * afeidxlist,
  const MSKrealt * b)
```

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 *MSK_appendacc* (see also *MSK_appendaccs*).

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `numaccs` (*MSKint64t*) – The number of affine conic constraints to append. (input)
- `accidxs` (*MSKint64t**) – Affine conic constraint indices. (input)
- `domidxs` (*MSKint64t**) – Domain indices. (input)
- `numafeidx` (*MSKint64t*) – Number of affine expressions in the affine expression list (must equal the sum of dimensions of the domains). (input)
- `afeidxlist` (*MSKint64t**) – List of affine expression indexes. (input)
- `b` (*MSKrealt**) – The vector of constant terms added to affine expressions. Optional, can be NULL. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine conic constraints*

MSK_putaccname

```
MSKrescodee (MSKAPI MSK_putaccname) (
  MSKtask_t task,
  MSKint64t accidx,
  const char * name)
```

Sets the name of an affine conic constraint.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `accidx` (*MSKint64t*) – Index of the affine conic constraint. (input)
- `name` (`char*`) – The name of the affine conic constraint. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - affine conic constraints*

MSK_putacol

```
MSKrescodee (MSKAPI MSK_putacol) (
  MSKtask_t task,
  MSKint32t j,
  MSKint32t nzj,
  const MSKint32t * subj,
  const MSKrealt * valj)
```

Change one column of the linear constraint matrix A. Resets all the elements in column j to zero and then sets

$$a_{\mathtt{subj}[k],\mathtt{j}} = \mathtt{valj}[k], \quad k = 0, \ldots, \mathtt{nzj} - 1.$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `j` (*MSKint32t*) – Index of a column in A. (input)
- `nzj` (*MSKint32t*) – Number of non-zeros in column j of A. (input)
- `subj` (*MSKint32t* *) – Row indexes of non-zero values in column j of A. (input)
- `valj` (*MSKrealt* *) – New non-zero values of column j in A. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

`MSK_putacollist`

```
MSKrescodee (MSKAPI MSK_putacollist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * sub,
  const MSKint32t * ptrb,
  const MSKint32t * ptre,
  const MSKint32t * asub,
  const MSKrealt * aval)
```

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:

$$\begin{array}{ll} \mathtt{for} & i = 0, \ldots, num - 1 \\ & a_{\mathtt{asub}[k],\mathtt{sub}[i]} = \mathtt{aval}[k], \quad k = \mathtt{ptrb}[i], \ldots, \mathtt{ptre}[i] - 1. \end{array}$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of columns of A to replace. (input)
- `sub` (*MSKint32t* *) – Indexes of columns that should be replaced, no duplicates. (input)
- `ptrb` (*MSKint32t* *) – Array of pointers to the first element in each column. (input)
- `ptre` (*MSKint32t* *) – Array of pointers to the last element plus one in each column. (input)
- `asub` (*MSKint32t* *) – Row indexes of new elements. (input)
- `aval` (*MSKrealt* *) – Coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

`MSK_putacollist64`

```
MSKrescodee (MSKAPI MSK_putacollist64) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * sub,
  const MSKint64t * ptrb,
  const MSKint64t * ptre,
  const MSKint32t * asub,
  const MSKrealt * aval)
```

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:

$$\begin{array}{ll}\texttt{for} & i = 0, \ldots, num - 1 \\ & a_{\texttt{asub}[k],\texttt{sub}[i]} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i], \ldots, \texttt{ptre}[i] - 1.\end{array}$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of columns of A to replace. (input)
- `sub` (*MSKint32t* *) – Indexes of columns that should be replaced, no duplicates. (input)
- `ptrb` (*MSKint64t* *) – Array of pointers to the first element in each column. (input)
- `ptre` (*MSKint64t* *) – Array of pointers to the last element plus one in each column. (input)
- `asub` (*MSKint32t* *) – Row indexes of new elements. (input)
- `aval` (*MSKrealt* *) – Coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

`MSK_putacolslice`

```
MSKrescodee (MSKAPI MSK_putacolslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  const MSKint32t * ptrb,
  const MSKint32t * ptre,
  const MSKint32t * asub,
  const MSKrealt * aval)
```

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:

$$\begin{array}{ll}\texttt{for} & i = \texttt{first}, \ldots, \texttt{last} - 1 \\ & a_{\texttt{asub}[k],i} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i - \texttt{first}], \ldots, \texttt{ptre}[i - \texttt{first}] - 1.\end{array}$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `first` (*MSKint32t*) – First column in the slice. (input)
- `last` (*MSKint32t*) – Last column plus one in the slice. (input)
- `ptrb` (*MSKint32t* *) – Array of pointers to the first element in each column. (input)
- `ptre` (*MSKint32t* *) – Array of pointers to the last element plus one in each column. (input)
- `asub` (*MSKint32t* *) – Row indexes of new elements. (input)
- `aval` (*MSKrealt* *) – Coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

`MSK_putacolslice64`

```
MSKrescodee (MSKAPI MSK_putacolslice64) (
  MSKtask_t task,
  MSKint32t first,
```

(continues on next page)

(continued from previous page)

```
  MSKint32t last,
  const MSKint64t * ptrb,
  const MSKint64t * ptre,
  const MSKint32t * asub,
  const MSKrealt * aval)
```

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:

```
for  $i = \mathtt{first}, \ldots, \mathtt{last} - 1$
     $a_{\mathtt{asub}[k],i} = \mathtt{aval}[k], \quad k = \mathtt{ptrb}[i - \mathtt{first}], \ldots, \mathtt{ptre}[i - \mathtt{first}] - 1.$
```

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First column in the slice. (input)
- last (*MSKint32t*) – Last column plus one in the slice. (input)
- ptrb (*MSKint64t* *) – Array of pointers to the first element in each column. (input)
- ptre (*MSKint64t* *) – Array of pointers to the last element plus one in each column. (input)
- asub (*MSKint32t* *) – Row indexes of new elements. (input)
- aval (*MSKrealt* *) – Coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

MSK_putafebarfblocktriplet

```
MSKrescodee (MSKAPI MSK_putafebarfblocktriplet) (
  MSKtask_t task,
  MSKint64t numtrip,
  const MSKint64t * afeidx,
  const MSKint32t * barvaridx,
  const MSKint32t * subk,
  const MSKint32t * subl,
  const MSKrealt * valkl)
```

Inputs the $\overline{F}$ matrix data in block triplet form.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numtrip (*MSKint64t*) – Number of elements in the block triplet form. (input)
- afeidx (*MSKint64t* *) – Constraint index. (input)
- barvaridx (*MSKint32t* *) – Symmetric matrix variable index. (input)
- subk (*MSKint32t* *) – Block row index. (input)
- subl (*MSKint32t* *) – Block column index. (input)
- valkl (*MSKrealt* *) – The numerical value associated with each block triplet. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*

MSK_putafebarfentry

```
MSKrescodee (MSKAPI MSK_putafebarfentry) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKint32t barvaridx,
  MSKint64t numterm,
  const MSKint64t * termidx,
  const MSKrealt * termweight)
```

This function sets one entry $\overline{F}_{ij}$ where $i = \text{afeidx}$ is the row index in the store of affine expressions and $j = \text{barvaridx}$ 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:

$$\overline{F}_{\text{afeidx},\text{barvaridx}} = \sum_k \text{termweight}[k] \cdot E_{\text{termidx}[k]}.$$

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 *MSK_appendsparsesymmat* .

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `afeidx` (*MSKint64t*) – Row index of $\overline{F}$. (input)
- `barvaridx` (*MSKint32t*) – Semidefinite variable index. (input)
- `numterm` (*MSKint64t*) – The number of terms in the weighted sum that forms the $\overline{F}$ entry being specified. (input)
- `termidx` (*MSKint64t* *) – Indices in E of the matrices appearing in the weighted sum for the $\overline{F}$ entry being specified. (input)
- `termweight` (*MSKrealt* *) – `termweight[k]` is the coefficient of the `termidx[k]`-th element of E in the weighted sum the $\overline{F}$ entry being specified. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*

`MSK_putafebarfentrylist`

```
MSKrescodee (MSKAPI MSK_putafebarfentrylist) (
  MSKtask_t task,
  MSKint64t numafeidx,
  const MSKint64t * afeidx,
  const MSKint32t * barvaridx,
  const MSKint64t * numterm,
  const MSKint64t * ptrterm,
  MSKint64t lenterm,
  const MSKint64t * termidx,
  const MSKrealt * termweight)
```

This function sets a list of entries in $\overline{F}$. Each entry should be described as in *MSK_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 *MSK_putafebarfentry* for details.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numafeidx (*MSKint64t*) – Number of elements in the list. (input)
- afeidx (*MSKint64t* *) – Row indexes of $\overline{F}$. (input)
- barvaridx (*MSKint32t* *) – Semidefinite variable indexes. (input)
- numterm (*MSKint64t* *) – The number of terms in the weighted sums that form each entry. (input)
- ptrterm (*MSKint64t* *) – The pointer to the beginning of the description of each entry. (input)
- lenterm (*MSKint64t*) – Length of the index and weight lists. (input)
- termidx (*MSKint64t* *) – Concatenated lists of indices in E of the matrices appearing in the weighted sums for the $\overline{F}$ being specified. (input)
- termweight (*MSKrealt* *) – Concatenated lists of weights appearing in the weighted sums forming the $\overline{F}$ elements being specified. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*

MSK_putafebarfrow

```
MSKrescodee (MSKAPI MSK_putafebarfrow) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKint32t numentr,
  const MSKint32t * barvaridx,
  const MSKint64t * numterm,
  const MSKint64t * ptrterm,
  MSKint64t lenterm,
  const MSKint64t * termidx,
  const MSKrealt * termweight)
```

This function inputs one row in $\overline{F}$. It first clears the row, i.e. sets $\overline{F}_{\text{afeidx},*} = 0$ and then sets the new entries. Each entry should be described as in *MSK_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

$$\text{ptrterm}[k], \ldots, \text{ptrterm}[k] + \text{numterm}[k] - 1$$

in the corresponding arrays termidx and termweight. See *MSK_putafebarfentry* for details.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- afeidx (*MSKint64t*) – Row index of $\overline{F}$. (input)
- numentr (*MSKint32t*) – Number of entries in the row of $\overline{F}$; length of barvaridx, numterm, ptrterm. (input)
- barvaridx (*MSKint32t* *) – Semidefinite variable indexes. (input)
- numterm (*MSKint64t* *) – The number of terms in the weighted sums that form each entry. (input)
- ptrterm (*MSKint64t* *) – The pointer to the beginning of the description of each entry. (input)
- lenterm (*MSKint64t*) – Length of the index and weight lists. (input)
- termidx (*MSKint64t* *) – Concatenated lists of indices in E of the matrices appearing in the weighted sums for the $\overline{F}$ entries in the row. (input)

- `termweight` (*MSKrealt* *) – Concatenated lists of weights appearing in the weighted sums forming the $\overline{F}$ entries in the row. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*, *Problem data - semidefinite*

`MSK_putafefcol`

```
MSKrescodee (MSKAPI MSK_putafefcol) (
  MSKtask_t task,
  MSKint32t varidx,
  MSKint64t numnz,
  const MSKint64t * afeidx,
  const MSKrealt * val)
```

Change one column of the matrix F of affine expressions. Resets all the elements in column `varidx` to zero and then sets

$$F_{\mathtt{afeidx}[k],\mathtt{varidx}} = \mathtt{val}[k], \quad k = 0,\ldots,\mathtt{numnz}-1.$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `varidx` (*MSKint32t*) – Index of a column in F. (input)
- `numnz` (*MSKint64t*) – Number of non-zeros in the column of F. (input)
- `afeidx` (*MSKint64t* *) – Row indexes of non-zero values in the column of F. (input)
- `val` (*MSKrealt* *) – New non-zero values in the column of F. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

`MSK_putafefentry`

```
MSKrescodee (MSKAPI MSK_putafefentry) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKint32t varidx,
  MSKrealt value)
```

Replaces one entry in the affine expression store F, that is it sets:

$$F_{\text{afeidx},\text{varidx}} = \text{value}.$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `afeidx` (*MSKint64t*) – Row index in F. (input)
- `varidx` (*MSKint32t*) – Column index in F. (input)
- `value` (*MSKrealt*) – Value of $F_{\text{afeidx},\text{varidx}}$. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

`MSK_putafefentrylist`

```
MSKrescodee (MSKAPI MSK_putafefentrylist) (
  MSKtask_t task,
  MSKint64t numentr,
  const MSKint64t * afeidx,
  const MSKint32t * varidx,
  const MSKrealt * val)
```

Replaces a number of entries in the affine expression store F, that is it sets:

$$F_{\mathrm{afeidxs}[k],\mathrm{varidx}[k]} = \mathrm{val}[k]$$

for all k.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `numentr` (*MSKint64t*) – Number of entries. (input)
- `afeidx` (*MSKint64t* *) – Row indices in F. (input)
- `varidx` (*MSKint32t* *) – Column indices in F. (input)
- `val` (*MSKrealt* *) – Values of the entries in F. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

`MSK_putafefrow`

```
MSKrescodee (MSKAPI MSK_putafefrow) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKint32t numnz,
  const MSKint32t * varidx,
  const MSKrealt * val)
```

Change one row of the matrix F of affine expressions. Resets all the elements in row `afeidx` to zero and then sets

$$F_{\mathtt{afeidx},\mathtt{varidx}[k]} = \mathtt{val}[k], \quad k = 0, \ldots, \mathtt{numnz} - 1.$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `afeidx` (*MSKint64t*) – Index of a row in F. (input)
- `numnz` (*MSKint32t*) – Number of non-zeros in the row of F. (input)
- `varidx` (*MSKint32t* *) – Column indexes of non-zero values in the row of F. (input)
- `val` (*MSKrealt* *) – New non-zero values in the row of F. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

`MSK_putafefrowlist`

```
MSKrescodee (MSKAPI MSK_putafefrowlist) (
  MSKtask_t task,
  MSKint64t numafeidx,
  const MSKint64t * afeidx,
  const MSKint32t * numnzrow,
  const MSKint64t * ptrrow,
  MSKint64t lenidxval,
  const MSKint32t * varidx,
  const MSKrealt * val)
```

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 $i = \mathrm{afeidx}[k]$, contains $\mathrm{numnzrow}[k]$ nonzeros and its description as in *MSK_putafefrow* starts in position $\mathrm{ptrrow}[k]$ of the arrays `varidx` and `val`. Formally, the row with index i is cleared and then set as:

$$F_{i,\mathrm{varidx}[\mathrm{ptrrow}[k]+j]} = \mathrm{val}[\mathrm{ptrrow}[k] + j], \quad j = 0, \ldots, \mathrm{numnzrow}[k] - 1.$$

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numafeidx (*MSKint64t*) – The number of rows of F to modify; length of afeidx. (input)
- afeidx (*MSKint64t* *) – Indices of rows in F. (input)
- numnzrow (*MSKint32t* *) – Number of non-zeros in each of the modified rows of F. (input)
- ptrrow (*MSKint64t* *) – Pointer to the first nonzero in each row of F. (input)
- lenidxval (*MSKint64t*) – Length of arrays varidx and val with indexes and values of nonzeros in the modified rows. (input)
- varidx (*MSKint32t* *) – Column indexes of non-zero values. (input)
- val (*MSKrealt* *) – New non-zero values in the rows of F. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

MSK_putafeg

```
MSKrescodee (MSKAPI MSK_putafeg) (
  MSKtask_t task,
  MSKint64t afeidx,
  MSKrealt g)
```

Change one element of the vector g in affine expressions i.e.

$$g_{\texttt{afeidx}} = \texttt{gi}.$$

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- afeidx (*MSKint64t*) – Index of an entry in g. (input)
- g (*MSKrealt*) – New value for g_{afeidx}. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

MSK_putafeglist

```
MSKrescodee (MSKAPI MSK_putafeglist) (
  MSKtask_t task,
  MSKint64t numafeidx,
  const MSKint64t * afeidx,
  const MSKrealt * g)
```

Changes a list of elements of the vector g in affine expressions i.e. for all k it sets

$$g_{\text{afeidx}[k]} = \text{glist}[k].$$

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numafeidx (*MSKint64t*) – Number of entries to change; length of afeidxlist. (input)
- afeidx (*MSKint64t* *) – Indices of entries in g. (input)
- g (*MSKrealt* *) – New values for g. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

MSK_putafegslice

```
MSKrescodee (MSKAPI MSK_putafegslice) (
  MSKtask_t task,
  MSKint64t first,
  MSKint64t last,
  const MSKrealt * slice)
```

Modifies a slice in the vector g of constant terms in affine expressions using the principle

$$g_{\mathtt{j}} = \mathtt{slice}[\mathtt{j} - \mathtt{first}], \quad j = \text{first}, .., \text{last} - 1$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `first` (*MSKint64t*) – First index in the sequence. (input)
- `last` (*MSKint64t*) – Last index plus 1 in the sequence. (input)
- `slice` (*MSKrealt**) – The slice of g as a dense vector. The length is `last-first`. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - affine expressions*

`MSK_putaij`

```
MSKrescodee (MSKAPI MSK_putaij) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t j,
  MSKrealt aij)
```

Changes a coefficient in the linear coefficient matrix A using the method

$$a_{i,j} = \mathtt{aij}.$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `i` (*MSKint32t*) – Constraint (row) index. (input)
- `j` (*MSKint32t*) – Variable (column) index. (input)
- `aij` (*MSKrealt*) – New coefficient for $a_{i,j}$. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

`MSK_putaijlist`

```
MSKrescodee (MSKAPI MSK_putaijlist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subi,
  const MSKint32t * subj,
  const MSKrealt * valij)
```

Changes one or more coefficients in A using the method

$$a_{\mathtt{subi}[\mathrm{k}],\mathtt{subj}[\mathrm{k}]} = \mathtt{valij}[\mathrm{k}], \quad k = 0, \ldots, num - 1.$$

Duplicates are not allowed.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)

- `num` (*MSKint32t*) – Number of coefficients that should be changed. (input)
- `subi` (*MSKint32t* *) – Constraint (row) indices. (input)
- `subj` (*MSKint32t* *) – Variable (column) indices. (input)
- `valij` (*MSKrealt* *) – New coefficient values for $a_{i,j}$. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

`MSK_putaijlist64`

```
MSKrescodee (MSKAPI MSK_putaijlist64) (
  MSKtask_t task,
  MSKint64t num,
  const MSKint32t * subi,
  const MSKint32t * subj,
  const MSKrealt * valij)
```

Changes one or more coefficients in A using the method

$$a_{\texttt{subi}[\texttt{k}],\texttt{subj}[\texttt{k}]} = \texttt{valij}[\texttt{k}], \quad k = 0, \ldots, num - 1.$$

Duplicates are not allowed.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint64t*) – Number of coefficients that should be changed. (input)
- `subi` (*MSKint32t* *) – Constraint (row) indices. (input)
- `subj` (*MSKint32t* *) – Variable (column) indices. (input)
- `valij` (*MSKrealt* *) – New coefficient values for $a_{i,j}$. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

`MSK_putarow`

```
MSKrescodee (MSKAPI MSK_putarow) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t nzi,
  const MSKint32t * subi,
  const MSKrealt * vali)
```

Change one row of the linear constraint matrix A. Resets all the elements in row i to zero and then sets

$$a_{\texttt{i},\texttt{subi}[k]} = \texttt{vali}[k], \quad k = 0, \ldots, \texttt{nzi} - 1.$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `i` (*MSKint32t*) – Index of a row in A. (input)
- `nzi` (*MSKint32t*) – Number of non-zeros in row i of A. (input)
- `subi` (*MSKint32t* *) – Column indexes of non-zero values in row i of A. (input)
- `vali` (*MSKrealt* *) – New non-zero values of row i in A. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

`MSK_putarowlist`

```
MSKrescodee (MSKAPI MSK_putarowlist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * sub,
  const MSKint32t * ptrb,
  const MSKint32t * ptre,
  const MSKint32t * asub,
  const MSKrealt * aval)
```

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:

$$\begin{array}{ll}\texttt{for} & i = 0, \ldots, num - 1 \\ & a_{\texttt{sub}[i],\texttt{asub}[k]} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i], \ldots, \texttt{ptre}[i] - 1.\end{array}$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of rows of A to replace. (input)
- `sub` (*MSKint32t* *) – Indexes of rows that should be replaced, no duplicates. (input)
- `ptrb` (*MSKint32t* *) – Array of pointers to the first element in each row. (input)
- `ptre` (*MSKint32t* *) – Array of pointers to the last element plus one in each row. (input)
- `asub` (*MSKint32t* *) – Column indexes of new elements. (input)
- `aval` (*MSKrealt* *) – Coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

MSK_putarowlist64

```
MSKrescodee (MSKAPI MSK_putarowlist64) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * sub,
  const MSKint64t * ptrb,
  const MSKint64t * ptre,
  const MSKint32t * asub,
  const MSKrealt * aval)
```

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:

$$\begin{array}{ll}\texttt{for} & i = 0, \ldots, num - 1 \\ & a_{\texttt{sub}[i],\texttt{asub}[k]} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i], \ldots, \texttt{ptre}[i] - 1.\end{array}$$

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of rows of A to replace. (input)
- `sub` (*MSKint32t* *) – Indexes of rows that should be replaced, no duplicates. (input)
- `ptrb` (*MSKint64t* *) – Array of pointers to the first element in each row. (input)
- `ptre` (*MSKint64t* *) – Array of pointers to the last element plus one in each row. (input)
- `asub` (*MSKint32t* *) – Column indexes of new elements. (input)
- `aval` (*MSKrealt* *) – Coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

MSK_putarowslice

```
MSKrescodee (MSKAPI MSK_putarowslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  const MSKint32t * ptrb,
  const MSKint32t * ptre,
  const MSKint32t * asub,
  const MSKrealt * aval)
```

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:

```
for  $i = \texttt{first}, \ldots, \texttt{last} - 1$
     $a_{i,\texttt{asub}[k]} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i - \texttt{first}], \ldots, \texttt{ptre}[i - \texttt{first}] - 1.$
```

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First row in the slice. (input)
- last (*MSKint32t*) – Last row plus one in the slice. (input)
- ptrb (*MSKint32t* *) – Array of pointers to the first element in each row. (input)
- ptre (*MSKint32t* *) – Array of pointers to the last element plus one in each row. (input)
- asub (*MSKint32t* *) – Column indexes of new elements. (input)
- aval (*MSKrealt* *) – Coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

MSK_putarowslice64

```
MSKrescodee (MSKAPI MSK_putarowslice64) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  const MSKint64t * ptrb,
  const MSKint64t * ptre,
  const MSKint32t * asub,
  const MSKrealt * aval)
```

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:

```
for  $i = \texttt{first}, \ldots, \texttt{last} - 1$
     $a_{i,\texttt{asub}[k]} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i - \texttt{first}], \ldots, \texttt{ptre}[i - \texttt{first}] - 1.$
```

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First row in the slice. (input)
- last (*MSKint32t*) – Last row plus one in the slice. (input)
- ptrb (*MSKint64t* *) – Array of pointers to the first element in each row. (input)
- ptre (*MSKint64t* *) – Array of pointers to the last element plus one in each row. (input)
- asub (*MSKint32t* *) – Column indexes of new elements. (input)

- aval (*MSKrealt* *) – Coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

MSK_putatruncatetol

```
MSKrescodee (MSKAPI MSK_putatruncatetol) (
  MSKtask_t task,
  MSKrealt tolzero)
```

Truncates (sets to zero) all elements in A that satisfy

$$|a_{i,j}| \leq \texttt{tolzero}.$$

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- tolzero (*MSKrealt*) – Truncation tolerance. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*

MSK_putbarablocktriplet

```
MSKrescodee (MSKAPI MSK_putbarablocktriplet) (
  MSKtask_t task,
  MSKint64t num,
  const MSKint32t * subi,
  const MSKint32t * subj,
  const MSKint32t * subk,
  const MSKint32t * subl,
  const MSKrealt * valijkl)
```

Inputs the $\overline{A}$ matrix in block triplet form.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint64t*) – Number of elements in the block triplet form. (input)
- subi (*MSKint32t* *) – Constraint index. (input)
- subj (*MSKint32t* *) – Symmetric matrix variable index. (input)
- subk (*MSKint32t* *) – Block row index. (input)
- subl (*MSKint32t* *) – Block column index. (input)
- valijkl (*MSKrealt* *) – The numerical value associated with each block triplet. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*

MSK_putbaraij

```
MSKrescodee (MSKAPI MSK_putbaraij) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t j,
  MSKint64t num,
  const MSKint64t * sub,
  const MSKrealt * weights)
```

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 *MSK_appendsparsesymmat*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Row index of $\overline{A}$. (input)
- j (*MSKint32t*) – Column index of $\overline{A}$. (input)
- num (*MSKint64t*) – The number of terms in the weighted sum that forms $\overline{A}_{ij}$. (input)
- sub (*MSKint64t* *) – Indices in E of the matrices appearing in the weighted sum for $\overline{A}_{ij}$. (input)
- weights (*MSKrealt* *) – weights[k] is the coefficient of the sub[k]-th element of E in the weighted sum forming $\overline{A}_{ij}$. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*

MSK_putbaraijlist

```
MSKrescodee (MSKAPI MSK_putbaraijlist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subi,
  const MSKint32t * subj,
  const MSKint64t * alphaptrb,
  const MSKint64t * alphaptre,
  const MSKint64t * matidx,
  const MSKrealt * weights)
```

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 *MSK_appendsparsesymmat*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of $\overline{A}$ entries to add. (input)
- subi (*MSKint32t* *) – Row index of $\overline{A}$. (input)
- subj (*MSKint32t* *) – Column index of $\overline{A}$. (input)
- alphaptrb (*MSKint64t* *) – Start entries for terms in the weighted sum that forms $\overline{A}_{ij}$. (input)
- alphaptre (*MSKint64t* *) – End entries for terms in the weighted sum that forms $\overline{A}_{ij}$. (input)
- matidx (*MSKint64t* *) – Indices in E of the matrices appearing in the weighted sum for $\overline{A}_{ij}$. (input)
- weights (*MSKrealt* *) – weights[k] is the coefficient of the sub[k]-th element of E in the weighted sum forming $\overline{A}_{ij}$. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*

MSK_putbararowlist

```
MSKrescodee (MSKAPI MSK_putbararowlist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subi,
  const MSKint64t * ptrb,
  const MSKint64t * ptre,
  const MSKint32t * subj,
  const MSKint64t * nummat,
  const MSKint64t * matidx,
  const MSKrealt * weights)
```

This function replaces a list of rows in the $\overline{A}$ matrix.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of $\overline{A}$ entries to add. (input)
- subi (*MSKint32t* *) – Row indexes of $\overline{A}$. (input)
- ptrb (*MSKint64t* *) – Start of rows in $\overline{A}$. (input)
- ptre (*MSKint64t* *) – End of rows in $\overline{A}$. (input)
- subj (*MSKint32t* *) – Column index of $\overline{A}$. (input)
- nummat (*MSKint64t* *) – Number of entries in weighted sum of matrixes. (input)
- matidx (*MSKint64t* *) – Matrix indexes for weighted sum of matrixes. (input)
- weights (*MSKrealt* *) – Weights for weighted sum of matrixes. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*

MSK_putbarcblocktriplet

```
MSKrescodee (MSKAPI MSK_putbarcblocktriplet) (
  MSKtask_t task,
  MSKint64t num,
  const MSKint32t * subj,
  const MSKint32t * subk,
  const MSKint32t * subl,
  const MSKrealt * valjkl)
```

Inputs the $\overline{C}$ matrix in block triplet form.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint64t*) – Number of elements in the block triplet form. (input)
- subj (*MSKint32t* *) – Symmetric matrix variable index. (input)
- subk (*MSKint32t* *) – Block row index. (input)
- subl (*MSKint32t* *) – Block column index. (input)
- valjkl (*MSKrealt* *) – The numerical value associated with each block triplet. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*

MSK_putbarcj

```
MSKrescodee (MSKAPI MSK_putbarcj) (
  MSKtask_t task,
  MSKint32t j,
  MSKint64t num,
  const MSKint64t * sub,
  const MSKrealt * weights)
```

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 *MSK_appendsparsesymmat*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- j (*MSKint32t*) – Index of the element in $\overline{C}$ that should be changed. (input)
- num (*MSKint64t*) – The number of elements in the weighted sum that forms $\overline{C}_j$. (input)
- sub (*MSKint64t* *) – Indices in E of matrices appearing in the weighted sum for $\overline{C}_j$ (input)
- weights (*MSKrealt* *) – weights[k] is the coefficient of the sub[k]-th element of E in the weighted sum forming $\overline{C}_j$. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*, *Problem data - objective*

MSK_putbarsj

```
MSKrescodee (MSKAPI MSK_putbarsj) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t j,
  const MSKrealt * barsj)
```

Sets the dual solution for a semidefinite variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- j (*MSKint32t*) – Index of the semidefinite variable. (input)
- barsj (*MSKrealt* *) – Value of $\overline{S}_j$. Format as in *MSK_getbarsj*. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - semidefinite*

MSK_putbarvarname

```
MSKrescodee (MSKAPI MSK_putbarvarname) (
  MSKtask_t task,
  MSKint32t j,
  const char * name)
```

Sets the name of a semidefinite variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)

- j (*MSKint32t*) – Index of the variable. (input)
- name (char*) – The variable name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - semidefinite*

MSK_putbarxj

```
MSKrescodee (MSKAPI MSK_putbarxj) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t j,
  const MSKrealt * barxj)
```

Sets the primal solution for a semidefinite variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- j (*MSKint32t*) – Index of the semidefinite variable. (input)
- barxj (*MSKrealt**) – Value of $\overline{X}_j$. Format as in *MSK_getbarxj*. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - semidefinite*

MSK_putcallbackfunc

```
MSKrescodee (MSKAPI MSK_putcallbackfunc) (
  MSKtask_t task,
  MSKcallbackfunc func,
  MSKuserhandle_t handle)
```

Sets a user-defined progress callback function of type *MSKcallbackfunc*. The callback function is called frequently during the optimization process. See Section *Progress and data callback* for an example.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- func (*MSKcallbackfunc*) – A user-defined function which will be called occasionally from within the **MOSEK** optimizers. If the argument is a NULL pointer, then a previously defined callback function is removed. The progress function has the type *MSKcallbackfunc*. (input)
- handle (*MSKuserhandle_t*) – A pointer to a user-defined data structure. Whenever the function func is called, then handle is passed to the function. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Callback*

MSK_putcfix

```
MSKrescodee (MSKAPI MSK_putcfix) (
  MSKtask_t task,
  MSKrealt cfix)
```

Replaces the fixed term in the objective by a new one.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- cfix (*MSKrealt*) – Fixed term in the objective. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Problem data - linear part*, *Problem data - objective*

MSK_putcj

```
MSKrescodee (MSKAPI MSK_putcj) (
  MSKtask_t task,
  MSKint32t j,
  MSKrealt cj)
```

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_t*) – An optimization task. (input)
- j (*MSKint32t*) – Index of the variable for which c should be changed. (input)
- cj (*MSKrealt*) – New value of c_j. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Problem data - linear part*, *Problem data - objective*

MSK_putclist

```
MSKrescodee (MSKAPI MSK_putclist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subj,
  const MSKrealt * val)
```

Modifies the coefficients in the linear term c in the objective using the principle

$$c_{\mathtt{subj[t]}} = \mathtt{val[t]}, \quad t = 0, \ldots, num - 1.$$

If a variable index is specified multiple times in subj only the last entry is used. Data checks are performed as in *MSK_putcj*.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of coefficients that should be changed. (input)
- subj (*MSKint32t**) – Indices of variables for which the coefficient in c should be changed. (input)
- val (*MSKrealt**) – New numerical values for coefficients in c that should be modified. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Problem data - linear part*, *Problem data - variables*, *Problem data - objective*

MSK_putconbound

```
MSKrescodee (MSKAPI MSK_putconbound) (
  MSKtask_t task,
  MSKint32t i,
  MSKboundkeye bkc,
  MSKrealt blc,
  MSKrealt buc)
```

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_t*) – An optimization task. (input)
- i (*MSKint32t*) – Index of the constraint. (input)
- bkc (*MSKboundkeye*) – New bound key. (input)
- blc (*MSKrealt*) – New lower bound. (input)
- buc (*MSKrealt*) – New upper bound. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - constraints*, *Problem data - bounds*

MSK_putconboundlist

```
MSKrescodee (MSKAPI MSK_putconboundlist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * sub,
  const MSKboundkeye * bkc,
  const MSKrealt * blc,
  const MSKrealt * buc)
```

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 *MSK_putconbound*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of bounds that should be changed. (input)
- sub (*MSKint32t* *) – List of constraint indexes. (input)
- bkc (*MSKboundkeye* *) – Bound keys for the constraints. (input)
- blc (*MSKrealt* *) – Lower bounds for the constraints. (input)
- buc (*MSKrealt* *) – Upper bounds for the constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - constraints*, *Problem data - bounds*

MSK_putconboundlistconst

```
MSKrescodee (MSKAPI MSK_putconboundlistconst) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * sub,
  MSKboundkeye bkc,
  MSKrealt blc,
  MSKrealt buc)
```

Changes the bounds for one or more constraints. Data checks are performed as in *MSK_putconbound*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of bounds that should be changed. (input)
- sub (*MSKint32t* *) – List of constraint indexes. (input)

- bkc (*MSKboundkeye*) – New bound key for all constraints in the list. (input)
- blc (*MSKrealt*) – New lower bound for all constraints in the list. (input)
- buc (*MSKrealt*) – New upper bound for all constraints in the list. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - constraints*, *Problem data - bounds*

MSK_putconboundslice

```
MSKrescodee (MSKAPI MSK_putconboundslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  const MSKboundkeye * bkc,
  const MSKrealt * blc,
  const MSKrealt * buc)
```

Changes the bounds for a slice of the constraints. Data checks are performed as in *MSK_putconbound*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- bkc (*MSKboundkeye**) – Bound keys for the constraints. (input)
- blc (*MSKrealt**) – Lower bounds for the constraints. (input)
- buc (*MSKrealt**) – Upper bounds for the constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - constraints*, *Problem data - bounds*

MSK_putconboundsliceconst

```
MSKrescodee (MSKAPI MSK_putconboundsliceconst) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKboundkeye bkc,
  MSKrealt blc,
  MSKrealt buc)
```

Changes the bounds for a slice of the constraints. Data checks are performed as in *MSK_putconbound*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- bkc (*MSKboundkeye*) – New bound key for all constraints in the slice. (input)
- blc (*MSKrealt*) – New lower bound for all constraints in the slice. (input)
- buc (*MSKrealt*) – New upper bound for all constraints in the slice. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - constraints*, *Problem data - bounds*

~~MSK_putcone~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_putcone) (
  MSKtask_t task,
  MSKint32t k,
  MSKconetypee ct,
  MSKrealt conepar,
  MSKint32t nummem,
  const MSKint32t * submem)
```

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_t*) – An optimization task. (input)
- `k` (*MSKint32t*) – Index of the cone. (input)
- `ct` (*MSKconetypee*) – Specifies the type of the cone. (input)
- `conepar` (*MSKrealt*) – For the power cone it denotes the exponent alpha. For other cone types it is unused and can be set to 0. (input)
- `nummem` (*MSKint32t*) – Number of member variables in the cone. (input)
- `submem` (*MSKint32t* *) – Variable subscripts of the members in the cone. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - cones (deprecated)*

~~`MSK_putconename`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_putconename) (
  MSKtask_t task,
  MSKint32t j,
  const char * name)
```

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_t*) – An optimization task. (input)
- `j` (*MSKint32t*) – Index of the cone. (input)
- `name` (`char*`) – The name of the cone. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - cones (deprecated)*

`MSK_putconname`

```
MSKrescodee (MSKAPI MSK_putconname) (
  MSKtask_t task,
  MSKint32t i,
  const char * name)
```

Sets the name of a constraint.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `i` (*MSKint32t*) – Index of the constraint. (input)
- `name` (`char*`) – The name of the constraint. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - constraints*, *Problem data - linear part*

MSK_putconsolutioni

```
MSKrescodee (MSKAPI MSK_putconsolutioni) (
  MSKtask_t task,
  MSKint32t i,
  MSKsoltypee whichsol,
  MSKstakeye sk,
  MSKrealt x,
  MSKrealt sl,
  MSKrealt su)
```

Sets the primal and dual solution information for a single constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Index of the constraint. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- sk (*MSKstakeye*) – Status key of the constraint. (input)
- x (*MSKrealt*) – Primal solution value of the constraint. (input)
- sl (*MSKrealt*) – Solution value of the dual variable associated with the lower bound. (input)
- su (*MSKrealt*) – Solution value of the dual variable associated with the upper bound. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Solution - primal*, *Solution - dual*

MSK_putcslice

```
MSKrescodee (MSKAPI MSK_putcslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  const MSKrealt * slice)
```

Modifies a slice in the linear term c in the objective using the principle

$$c_{\mathtt{j}} = \mathtt{slice}[\mathtt{j} - \mathtt{first}], \quad j = first, .., last - 1$$

Data checks are performed as in *MSK_putcj*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First element in the slice of c. (input)
- last (*MSKint32t*) – Last element plus 1 of the slice in c to be changed. (input)
- slice (*MSKrealt* *) – New numerical values for coefficients in c that should be modified. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - objective*

MSK_putdjc

```
MSKrescodee (MSKAPI MSK_putdjc) (
  MSKtask_t task,
  MSKint64t djcidx,
```

(continues on next page)

(continued from previous page)

```
  MSKint64t numdomidx,
  const MSKint64t * domidxlist,
  MSKint64t numafeidx,
  const MSKint64t * afeidxlist,
  const MSKrealt * b,
  MSKint64t numterms,
  const MSKint64t * termsizelist)
```

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_t*) – An optimization task. (input)
- `djcidx` (*MSKint64t*) – Index of the disjunctive constraint. (input)
- `numdomidx` (*MSKint64t*) – Number of domains. (input)
- `domidxlist` (*MSKint64t* *) – List of domain indexes. (input)
- `numafeidx` (*MSKint64t*) – Number of affine expressions. (input)
- `afeidxlist` (*MSKint64t* *) – List of affine expression indexes. (input)
- `b` (*MSKrealt* *) – The vector of constant terms added to affine expressions. (input)
- `numterms` (*MSKint64t*) – Number of terms in disjunctive constraint. (input)
- `termsizelist` (*MSKint64t* *) – List of term sizes. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*

`MSK_putdjcname`

```
MSKrescodee (MSKAPI MSK_putdjcname) (
  MSKtask_t task,
  MSKint64t djcidx,
  const char * name)
```

Sets the name of a disjunctive constraint.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- djcidx (*MSKint64t*) – Index of the disjunctive constraint. (input)
- name (char*) – The name of the disjunctive constraint. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - disjunctive constraints*

MSK_putdjcslice

```
MSKrescodee (MSKAPI MSK_putdjcslice) (
  MSKtask_t task,
  MSKint64t idxfirst,
  MSKint64t idxlast,
  MSKint64t numdomidx,
  const MSKint64t * domidxlist,
  MSKint64t numafeidx,
  const MSKint64t * afeidxlist,
  const MSKrealt * b,
  MSKint64t numterms,
  const MSKint64t * termsizelist,
  const MSKint64t * termsindjc)
```

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 *MSK_putdjc*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- idxfirst (*MSKint64t*) – Index of the first disjunctive constraint in the slice. (input)
- idxlast (*MSKint64t*) – Index of the last disjunctive constraint in the slice plus 1. (input)
- numdomidx (*MSKint64t*) – Number of domains. (input)
- domidxlist (*MSKint64t**) – List of domain indexes. (input)
- numafeidx (*MSKint64t*) – Number of affine expressions. (input)
- afeidxlist (*MSKint64t**) – List of affine expression indexes. (input)
- b (*MSKrealt**) – The vector of constant terms added to affine expressions. Optional, may be NULL. (input)
- numterms (*MSKint64t*) – Number of terms in the disjunctive constraints. (input)
- termsizelist (*MSKint64t**) – List of term sizes. (input)
- termsindjc (*MSKint64t**) – Number of terms in each of the disjunctive constraints in the slice. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - disjunctive constraints*

MSK_putdomainname

```
MSKrescodee (MSKAPI MSK_putdomainname) (
  MSKtask_t task,
  MSKint64t domidx,
  const char * name)
```

Sets the name of a domain.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `domidx` (*MSKint64t*) – Index of the domain. (input)
- `name` (`char*`) – The name of the domain. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - domain*

`MSK_putdouparam`

```
MSKrescodee (MSKAPI MSK_putdouparam) (
  MSKtask_t task,
  MSKdparame param,
  MSKrealt parvalue)
```

Sets the value of a double parameter.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `param` (*MSKdparame*) – Which parameter. (input)
- `parvalue` (*MSKrealt*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

`MSK_putexitfunc`

```
MSKrescodee (MSKAPI MSK_putexitfunc) (
  MSKenv_t env,
  MSKexitfunc exitfunc,
  MSKuserhandle_t handle)
```

In case **MOSEK** experiences a fatal error, then a user-defined exit function can be called. The exit function should terminate **MOSEK**. In general it is not necessary to define an exit function.

Parameters

- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `exitfunc` (*MSKexitfunc*) – A user-defined exit function. (input)
- `handle` (*MSKuserhandle_t*) – A pointer to a user-defined data structure which is passed to `exitfunc` when called. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*, *Callback*

`MSK_putintparam`

```
MSKrescodee (MSKAPI MSK_putintparam) (
  MSKtask_t task,
  MSKiparame param,
  MSKint32t parvalue)
```

Sets the value of an integer parameter.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `param` (*MSKiparame*) – Which parameter. (input)
- `parvalue` (*MSKint32t*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

`MSK_putlicensecode`

```
MSKrescodee (MSKAPI MSK_putlicensecode) (
  MSKenv_t env,
  const MSKint32t * code)
```

Input a runtime license code.

Parameters

- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `code` (*MSKint32t* *) – A runtime license code. (input)

Return (*MSKrescodee*) – The function response code.

Groups *License system*

`MSK_putlicensedebug`

```
MSKrescodee (MSKAPI MSK_putlicensedebug) (
  MSKenv_t env,
  MSKint32t licdebug)
```

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_t*) – The MOSEK environment. (input)
- `licdebug` (*MSKint32t*) – Whether license checkout debug info should be printed. (input)

Return (*MSKrescodee*) – The function response code.

Groups *License system*

`MSK_putlicensepath`

```
MSKrescodee (MSKAPI MSK_putlicensepath) (
  MSKenv_t env,
  const char * licensepath)
```

Set the path to the license file.

Parameters

- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `licensepath` (`char*`) – A path specifying where to search for the license. (input)

Return (*MSKrescodee*) – The function response code.

Groups *License system*

`MSK_putlicensewait`

```
MSKrescodee (MSKAPI MSK_putlicensewait) (
  MSKenv_t env,
  MSKint32t licwait)
```

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_t*) – The MOSEK environment. (input)
- `licwait` (*MSKint32t*) – Whether **MOSEK** should wait for a license if no license is available. (input)

Return (*MSKrescodee*) – The function response code.

Groups *License system*

`MSK_putmaxnumacc`

```
MSKrescodee (MSKAPI MSK_putmaxnumacc) (
  MSKtask_t task,
  MSKint64t maxnumacc)
```

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_t*) – An optimization task. (input)
- `maxnumacc` (*MSKint64t*) – Number of preallocated affine conic constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*, *Problem data - affine conic constraints*

`MSK_putmaxnumafe`

```
MSKrescodee (MSKAPI MSK_putmaxnumafe) (
  MSKtask_t task,
  MSKint64t maxnumafe)
```

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_t*) – An optimization task. (input)
- `maxnumafe` (*MSKint64t*) – Number of preallocated affine expressions. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*, *Problem data - affine expressions*

`MSK_putmaxnumanz`

```
MSKrescodee (MSKAPI MSK_putmaxnumanz) (
  MSKtask_t task,
  MSKint64t maxnumanz)
```

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_t*) – An optimization task. (input)
- maxnumanz (*MSKint64t*) – Number of preallocated non-zeros in A. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*, *Problem data - linear part*

MSK_putmaxnumbarvar

```
MSKrescodee (MSKAPI MSK_putmaxnumbarvar) (
  MSKtask_t task,
  MSKint32t maxnumbarvar)
```

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_t*) – An optimization task. (input)
- maxnumbarvar (*MSKint32t*) – Number of preallocated symmetric matrix variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*, *Problem data - semidefinite*

MSK_putmaxnumcon

```
MSKrescodee (MSKAPI MSK_putmaxnumcon) (
  MSKtask_t task,
  MSKint32t maxnumcon)
```

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_t*) – An optimization task. (input)
- maxnumcon (*MSKint32t*) – Number of preallocated constraints in the optimization task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*, *Problem data - constraints*

~~MSK_putmaxnumcone~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_putmaxnumcone) (
  MSKtask_t task,
  MSKint32t maxnumcone)
```

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_t*) – An optimization task. (input)
- `maxnumcone` (*MSKint32t*) – Number of preallocated conic constraints in the optimization task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*, *Problem data - cones (deprecated)*

`MSK_putmaxnumdjc`

```
MSKrescodee (MSKAPI MSK_putmaxnumdjc) (
  MSKtask_t task,
  MSKint64t maxnumdjc)
```

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_t*) – An optimization task. (input)
- `maxnumdjc` (*MSKint64t*) – Number of preallocated disjunctive constraints in the task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*, *Problem data - disjunctive constraints*

`MSK_putmaxnumdomain`

```
MSKrescodee (MSKAPI MSK_putmaxnumdomain) (
  MSKtask_t task,
  MSKint64t maxnumdomain)
```

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_t*) – An optimization task. (input)
- `maxnumdomain` (*MSKint64t*) – Number of preallocated domains. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*, *Problem data - domain*

`MSK_putmaxnumqnz`

```
MSKrescodee (MSKAPI MSK_putmaxnumqnz) (
  MSKtask_t task,
  MSKint64t maxnumqnz)
```

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_t*) – An optimization task. (input)
- `maxnumqnz` (*MSKint64t*) – Number of non-zero elements preallocated in quadratic coefficient matrices. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*, *Problem data - quadratic part*

`MSK_putmaxnumvar`

```
MSKrescodee (MSKAPI MSK_putmaxnumvar) (
  MSKtask_t task,
  MSKint32t maxnumvar)
```

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_t*) – An optimization task. (input)
- `maxnumvar` (*MSKint32t*) – Number of preallocated variables in the optimization task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*, *Problem data - variables*

`MSK_putnadouparam`

```
MSKrescodee (MSKAPI MSK_putnadouparam) (
  MSKtask_t task,
  const char * paramname,
  MSKrealt parvalue)
```

Sets the value of a named double parameter.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `paramname` (`char*`) – Name of a parameter. (input)
- `parvalue` (*MSKrealt*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

`MSK_putnaintparam`

```
MSKrescodee (MSKAPI MSK_putnaintparam) (
  MSKtask_t task,
  const char * paramname,
  MSKint32t parvalue)
```

Sets the value of a named integer parameter.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- paramname (char*) – Name of a parameter. (input)
- parvalue (*MSKint32t*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_putnastrparam

```
MSKrescodee (MSKAPI MSK_putnastrparam) (
  MSKtask_t task,
  const char * paramname,
  const char * parvalue)
```

Sets the value of a named string parameter.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- paramname (char*) – Name of a parameter. (input)
- parvalue (char*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_putobjname

```
MSKrescodee (MSKAPI MSK_putobjname) (
  MSKtask_t task,
  const char * objname)
```

Assigns a new name to the objective.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- objname (char*) – Name of the objective. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Names*, *Problem data - objective*

MSK_putobjsense

```
MSKrescodee (MSKAPI MSK_putobjsense) (
  MSKtask_t task,
  MSKobjsensee sense)
```

Sets the objective sense of the task.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- sense (*MSKobjsensee*) – 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)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - objective*

MSK_putoptserverhost

```
MSKrescodee (MSKAPI MSK_putoptserverhost) (
  MSKtask_t task,
  const char * host)
```

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 NULL deactivates this redirection.

Has the same effect as setting the parameter *MSK_SPAR_REMOTE_OPTSERVER_HOST*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `host` (`char*`) – A URL specifying the optimization server to be used. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Remote optimization*

MSK_putparam

```
MSKrescodee (MSKAPI MSK_putparam) (
  MSKtask_t task,
  const char * parname,
  const char * parvalue)
```

Checks if `parname` is valid parameter name. If it is, the parameter is assigned the value specified by `parvalue`.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `parname` (`char*`) – Parameter name. (input)
- `parvalue` (`char*`) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_putqcon

```
MSKrescodee (MSKAPI MSK_putqcon) (
  MSKtask_t task,
  MSKint32t numqcnz,
  const MSKint32t * qcsubk,
  const MSKint32t * qcsubi,
  const MSKint32t * qcsubj,
  const MSKrealt * qcval)
```

Replace all quadratic entries in the constraints. The list of constraints has the form

$$l_k^c \leq \frac{1}{2} \sum_{i=0}^{numvar-1} \sum_{j=0}^{numvar-1} q_{ij}^k x_i x_j + \sum_{j=0}^{numvar-1} a_{kj} x_j \leq u_k^c, \quad k = 0, \ldots, m-1.$$

This function sets all the quadratic terms to zero and then performs the update:

$$q_{\mathtt{qcsubi[t]},\mathtt{qcsubj[t]}}^{\mathtt{qcsubk[t]}} = q_{\mathtt{qcsubj[t]},\mathtt{qcsubi[t]}}^{\mathtt{qcsubk[t]}} = q_{\mathtt{qcsubj[t]},\mathtt{qcsubi[t]}}^{\mathtt{qcsubk[t]}} + \mathtt{qcval[t]},$$

for $t = 0, \ldots, numqcnz - 1$.

Please note that:

- For large problems it is essential for the efficiency that the function *MSK_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_t*) – An optimization task. (input)
- `numqcnz` (*MSKint32t*) – Number of quadratic terms. (input)
- `qcsubk` (*MSKint32t* *) – Constraint subscripts for quadratic coefficients. (input)
- `qcsubi` (*MSKint32t* *) – Row subscripts for quadratic constraint matrix. (input)
- `qcsubj` (*MSKint32t* *) – Column subscripts for quadratic constraint matrix. (input)
- `qcval` (*MSKrealt* *) – Quadratic constraint coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - quadratic part*

`MSK_putqconk`

```
MSKrescodee (MSKAPI MSK_putqconk) (
  MSKtask_t task,
  MSKint32t k,
  MSKint32t numqcnz,
  const MSKint32t * qcsubi,
  const MSKint32t * qcsubj,
  const MSKrealt * qcval)
```

Replaces all the quadratic entries in one constraint. This function performs the same operations as *MSK_putqcon* but only with respect to constraint number `k` and it does not modify the other constraints. See the description of *MSK_putqcon* for definitions and important remarks.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `k` (*MSKint32t*) – The constraint in which the new Q elements are inserted. (input)
- `numqcnz` (*MSKint32t*) – Number of quadratic terms. (input)
- `qcsubi` (*MSKint32t* *) – Row subscripts for quadratic constraint matrix. (input)
- `qcsubj` (*MSKint32t* *) – Column subscripts for quadratic constraint matrix. (input)
- `qcval` (*MSKrealt* *) – Quadratic constraint coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - quadratic part*

`MSK_putqobj`

```
MSKrescodee (MSKAPI MSK_putqobj) (
  MSKtask_t task,
  MSKint32t numqonz,
  const MSKint32t * qosubi,
  const MSKint32t * qosubj,
  const MSKrealt * qoval)
```

Replace all quadratic terms in the objective. If the objective has the form

$$\frac{1}{2}\sum_{i=0}^{numvar-1}\sum_{j=0}^{numvar-1} q_{ij}^o x_i x_j + \sum_{j=0}^{numvar-1} c_j x_j + c^f$$

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 $t = 0, \ldots, numqonz - 1$.

See the description of *MSK_putqcon* for important remarks and example.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numqonz (*MSKint32t*) – Number of non-zero elements in the quadratic objective terms. (input)
- qosubi (*MSKint32t* *) – Row subscripts for quadratic objective coefficients. (input)
- qosubj (*MSKint32t* *) – Column subscripts for quadratic objective coefficients. (input)
- qoval (*MSKrealt* *) – Quadratic objective coefficient values. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - quadratic part*, *Problem data - objective*

MSK_putqobjij

```
MSKrescodee (MSKAPI MSK_putqobjij) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t j,
  MSKrealt qoij)
```

Replaces one coefficient in the quadratic term in the objective. The function performs the assignment

$$q_{ij}^o = q_{ji}^o = \texttt{qoij}.$$

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 *MSK_putqobj* instead whenever possible.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Row index for the coefficient to be replaced. (input)
- j (*MSKint32t*) – Column index for the coefficient to be replaced. (input)
- qoij (*MSKrealt*) – The new value for q_{ij}^o. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - quadratic part*, *Problem data - objective*

MSK_putresponsefunc

```
MSKrescodee (MSKAPI MSK_putresponsefunc) (
  MSKtask_t task,
  MSKresponsefunc responsefunc,
  MSKuserhandle_t handle)
```

Inputs a user-defined error callback which is called when an error or warning occurs.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `responsefunc` (*MSKresponsefunc*) – A user-defined response handling function. (input)
- `handle` (*MSKuserhandle_t*) – A user-defined data structure that is passed to the function `responsefunc` whenever it is called. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Callback*, *Logging*

MSK_putskc

```
MSKrescodee (MSKAPI MSK_putskc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKstakeye * skc)
```

Sets the status keys for the constraints.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `skc` (*MSKstakeye**) – Status keys for the constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_putskcslice

```
MSKrescodee (MSKAPI MSK_putskcslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const MSKstakeye * skc)
```

Sets the status keys for a slice of the constraints.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – First index in the sequence. (input)
- `last` (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- `skc` (*MSKstakeye**) – Status keys for the constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_putskx

```
MSKrescodee (MSKAPI MSK_putskx) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKstakeye * skx)
```

Sets the status keys for the scalar variables.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `skx` (*MSKstakeye**) – Status keys for the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

`MSK_putskxslice`

```
MSKrescodee (MSKAPI MSK_putskxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const MSKstakeye * skx)
```

Sets the status keys for a slice of the variables.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – First index in the sequence. (input)
- `last` (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- `skx` (*MSKstakeye**) – Status keys for the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

`MSK_putslc`

```
MSKrescodee (MSKAPI MSK_putslc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKrealt * slc)
```

Sets the s_l^c vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `slc` (*MSKrealt**) – Dual variables corresponding to the lower bounds on the constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

`MSK_putslcslice`

```
MSKrescodee (MSKAPI MSK_putslcslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const MSKrealt * slc)
```

Sets a slice of the s_l^c vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- slc (*MSKrealt* *) – Dual variables corresponding to the lower bounds on the constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_putslx

```
MSKrescodee (MSKAPI MSK_putslx) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKrealt * slx)
```

Sets the s_l^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- slx (*MSKrealt* *) – Dual variables corresponding to the lower bounds on the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_putslxslice

```
MSKrescodee (MSKAPI MSK_putslxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const MSKrealt * slx)
```

Sets a slice of the s_l^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- slx (*MSKrealt* *) – Dual variables corresponding to the lower bounds on the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_putsnx

```
MSKrescodee (MSKAPI MSK_putsnx) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKrealt * sux)
```

Sets the s_n^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- sux (*MSKrealt**) – Dual variables corresponding to the upper bounds on the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_putsnxslice

```
MSKrescodee (MSKAPI MSK_putsnxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const MSKrealt * snx)
```

Sets a slice of the s_n^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- snx (*MSKrealt**) – Dual variables corresponding to the conic constraints on the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_putsolution

```
MSKrescodee (MSKAPI MSK_putsolution) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKstakeye * skc,
  const MSKstakeye * skx,
  const MSKstakeye * skn,
  const MSKrealt * xc,
  const MSKrealt * xx,
  const MSKrealt * y,
  const MSKrealt * slc,
  const MSKrealt * suc,
  const MSKrealt * slx,
  const MSKrealt * sux,
  const MSKrealt * snx)
```

Inserts a solution into the task.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- skc (*MSKstakeye* *) – Status keys for the constraints. (input)
- skx (*MSKstakeye* *) – Status keys for the variables. (input)
- skn (*MSKstakeye* *) – Status keys for the conic constraints. (input)
- xc (*MSKrealt* *) – Primal constraint solution. (input)
- xx (*MSKrealt* *) – Primal variable solution. (input)
- y (*MSKrealt* *) – Vector of dual variables corresponding to the constraints. (input)
- slc (*MSKrealt* *) – Dual variables corresponding to the lower bounds on the constraints. (input)
- suc (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the constraints. (input)
- slx (*MSKrealt* *) – Dual variables corresponding to the lower bounds on the variables. (input)
- sux (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the variables. (input)
- snx (*MSKrealt* *) – Dual variables corresponding to the conic constraints on the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Solution - primal*, *Solution - dual*

MSK_putsolutionnew

```
MSKrescodee (MSKAPI MSK_putsolutionnew) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKstakeye * skc,
  const MSKstakeye * skx,
  const MSKstakeye * skn,
  const MSKrealt * xc,
  const MSKrealt * xx,
  const MSKrealt * y,
  const MSKrealt * slc,
  const MSKrealt * suc,
  const MSKrealt * slx,
  const MSKrealt * sux,
  const MSKrealt * snx,
  const MSKrealt * doty)
```

Inserts a solution into the task.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- skc (*MSKstakeye* *) – Status keys for the constraints. (input)
- skx (*MSKstakeye* *) – Status keys for the variables. (input)
- skn (*MSKstakeye* *) – Status keys for the conic constraints. (input)
- xc (*MSKrealt* *) – Primal constraint solution. (input)
- xx (*MSKrealt* *) – Primal variable solution. (input)
- y (*MSKrealt* *) – Vector of dual variables corresponding to the constraints. (input)

- slc (*MSKrealt**) – Dual variables corresponding to the lower bounds on the constraints. (input)
- suc (*MSKrealt**) – Dual variables corresponding to the upper bounds on the constraints. (input)
- slx (*MSKrealt**) – Dual variables corresponding to the lower bounds on the variables. (input)
- sux (*MSKrealt**) – Dual variables corresponding to the upper bounds on the variables. (input)
- snx (*MSKrealt**) – Dual variables corresponding to the conic constraints on the variables. (input)
- doty (*MSKrealt**) – Dual variables corresponding to affine conic constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Solution - primal*, *Solution - dual*

MSK_putsolutionyi

```
MSKrescodee (MSKAPI MSK_putsolutionyi) (
  MSKtask_t task,
  MSKint32t i,
  MSKsoltypee whichsol,
  MSKrealt y)
```

Inputs the dual variable of a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- i (*MSKint32t*) – Index of the dual variable. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- y (*MSKrealt*) – Solution value of the dual variable. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Solution - dual*

MSK_putstrparam

```
MSKrescodee (MSKAPI MSK_putstrparam) (
  MSKtask_t task,
  MSKsparame param,
  const char * parvalue)
```

Sets the value of a string parameter.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- param (*MSKsparame*) – Which parameter. (input)
- parvalue (char*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_putsuc

```
MSKrescodee (MSKAPI MSK_putsuc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKrealt * suc)
```

Sets the s_u^c vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- suc (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_putsucslice

```
MSKrescodee (MSKAPI MSK_putsucslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const MSKrealt * suc)
```

Sets a slice of the s_u^c vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- suc (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_putsux

```
MSKrescodee (MSKAPI MSK_putsux) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKrealt * sux)
```

Sets the s_u^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- sux (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

MSK_putsuxslice

```
MSKrescodee (MSKAPI MSK_putsuxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const MSKrealt * sux)
```

Sets a slice of the s_u^x vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – First index in the sequence. (input)
- `last` (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- `sux` (*MSKrealt**) – Dual variables corresponding to the upper bounds on the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

`MSK_puttaskname`

```
MSKrescodee (MSKAPI MSK_puttaskname) (
  MSKtask_t task,
  const char * taskname)
```

Assigns a new name to the task.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `taskname` (`char*`) – Name assigned to the task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Environment and task management*

`MSK_putvarbound`

```
MSKrescodee (MSKAPI MSK_putvarbound) (
  MSKtask_t task,
  MSKint32t j,
  MSKboundkeye bkx,
  MSKrealt blx,
  MSKrealt bux)
```

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_t*) – An optimization task. (input)
- `j` (*MSKint32t*) – Index of the variable. (input)
- `bkx` (*MSKboundkeye*) – New bound key. (input)
- `blx` (*MSKrealt*) – New lower bound. (input)
- `bux` (*MSKrealt*) – New upper bound. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - variables*, *Problem data - bounds*

`MSK_putvarboundlist`

```
MSKrescodee (MSKAPI MSK_putvarboundlist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * sub,
  const MSKboundkeye * bkx,
  const MSKrealt * blx,
  const MSKrealt * bux)
```

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 *MSK_putvarbound*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of bounds that should be changed. (input)
- sub (*MSKint32t* *) – List of variable indexes. (input)
- bkx (*MSKboundkeye* *) – Bound keys for the variables. (input)
- blx (*MSKrealt* *) – Lower bounds for the variables. (input)
- bux (*MSKrealt* *) – Upper bounds for the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - variables*, *Problem data - bounds*

MSK_putvarboundlistconst

```
MSKrescodee (MSKAPI MSK_putvarboundlistconst) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * sub,
  MSKboundkeye bkx,
  MSKrealt blx,
  MSKrealt bux)
```

Changes the bounds for one or more variables. Data checks are performed as in *MSK_putvarbound*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of bounds that should be changed. (input)
- sub (*MSKint32t* *) – List of variable indexes. (input)
- bkx (*MSKboundkeye*) – New bound key for all variables in the list. (input)
- blx (*MSKrealt*) – New lower bound for all variables in the list. (input)
- bux (*MSKrealt*) – New upper bound for all variables in the list. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - variables*, *Problem data - bounds*

MSK_putvarboundslice

```
MSKrescodee (MSKAPI MSK_putvarboundslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  const MSKboundkeye * bkx,
  const MSKrealt * blx,
  const MSKrealt * bux)
```

Changes the bounds for a slice of the variables. Data checks are performed as in *MSK_putvarbound*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- bkx (*MSKboundkeye* *) – Bound keys for the variables. (input)
- blx (*MSKrealt* *) – Lower bounds for the variables. (input)
- bux (*MSKrealt* *) – Upper bounds for the variables. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - variables*, *Problem data - bounds*

MSK_putvarboundsliceconst

```
MSKrescodee (MSKAPI MSK_putvarboundsliceconst) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKboundkeye bkx,
  MSKrealt blx,
  MSKrealt bux)
```

Changes the bounds for a slice of the variables. Data checks are performed as in *MSK_putvarbound*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- bkx (*MSKboundkeye*) – New bound key for all variables in the slice. (input)
- blx (*MSKrealt*) – New lower bound for all variables in the slice. (input)
- bux (*MSKrealt*) – New upper bound for all variables in the slice. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - linear part*, *Problem data - variables*, *Problem data - bounds*

MSK_putvarname

```
MSKrescodee (MSKAPI MSK_putvarname) (
  MSKtask_t task,
  MSKint32t j,
  const char * name)
```

Sets the name of a variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- j (*MSKint32t*) – Index of the variable. (input)
- name (char*) – The variable name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Names*, *Problem data - variables*, *Problem data - linear part*

MSK_putvarsolutionj

```
MSKrescodee (MSKAPI MSK_putvarsolutionj) (
  MSKtask_t task,
  MSKint32t j,
  MSKsoltypee whichsol,
  MSKstakeye sk,
```

(continues on next page)

(continued from previous page)

```
  MSKrealt x,
  MSKrealt sl,
  MSKrealt su,
  MSKrealt sn)
```

Sets the primal and dual solution information for a single variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- j (*MSKint32t*) – Index of the variable. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- sk (*MSKstakeye*) – Status key of the variable. (input)
- x (*MSKrealt*) – Primal solution value of the variable. (input)
- sl (*MSKrealt*) – Solution value of the dual variable associated with the lower bound. (input)
- su (*MSKrealt*) – Solution value of the dual variable associated with the upper bound. (input)
- sn (*MSKrealt*) – Solution value of the dual variable associated with the conic constraint. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*, *Solution - primal*, *Solution - dual*

MSK_putvartype

```
MSKrescodee (MSKAPI MSK_putvartype) (
  MSKtask_t task,
  MSKint32t j,
  MSKvariabletypee vartype)
```

Sets the variable type of one variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- j (*MSKint32t*) – Index of the variable. (input)
- vartype (*MSKvariabletypee*) – The new variable type. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - variables*

MSK_putvartypelist

```
MSKrescodee (MSKAPI MSK_putvartypelist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subj,
  const MSKvariabletypee * vartype)
```

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_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of variables for which the variable type should be set. (input)
- subj (*MSKint32t* *) – A list of variable indexes for which the variable type should be changed. (input)

- vartype (*MSKvariabletypee* *) – A list of variable types that should be assigned to the variables specified by subj. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - variables*

MSK_putxc

```
MSKrescodee (MSKAPI MSK_putxc) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKrealt * xc)
```

Sets the x^c vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- xc (*MSKrealt* *) – Primal constraint solution. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*

MSK_putxcslice

```
MSKrescodee (MSKAPI MSK_putxcslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const MSKrealt * xc)
```

Sets a slice of the x^c vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- xc (*MSKrealt* *) – Primal constraint solution. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*

MSK_putxx

```
MSKrescodee (MSKAPI MSK_putxx) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKrealt * xx)
```

Sets the x^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- xx (*MSKrealt* *) – Primal variable solution. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*

MSK_putxxslice

```
MSKrescodee (MSKAPI MSK_putxxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const MSKrealt * xx)
```

Sets a slice of the x^x vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – First index in the sequence. (input)
- `last` (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- `xx` (*MSKrealt* *) – Primal variable solution. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*

MSK_puty

```
MSKrescodee (MSKAPI MSK_puty) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const MSKrealt * y)
```

Sets the y vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `y` (*MSKrealt* *) – Vector of dual variables corresponding to the constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - primal*

MSK_putyslice

```
MSKrescodee (MSKAPI MSK_putyslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const MSKrealt * y)
```

Sets a slice of the y vector for a solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `first` (*MSKint32t*) – First index in the sequence. (input)
- `last` (*MSKint32t*) – Last index plus 1 in the sequence. (input)

- y (*MSKrealt* *) – Vector of dual variables corresponding to the constraints. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution - dual*

`MSK_readbsolution`

```
MSKrescodee (MSKAPI MSK_readbsolution) (
  MSKtask_t task,
  const char * filename,
  MSKcompresstypee compress)
```

Read a binary dump of the task solution.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)
- `compress` (*MSKcompresstype*) – Data compression type. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

`MSK_readdata`

```
MSKrescodee (MSKAPI MSK_readdata) (
  MSKtask_t task,
  const char * filename)
```

Reads an optimization problem and associated data from a file. The extension of the file name is used to deduce the file format.

For a list of supported file types and their extensions see *Supported File Formats*.

Data is read from the file `filename` if it is a nonempty string. Otherwise data is read from the file specified by *MSK_SPAR_DATA_FILE_NAME*.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

`MSK_readdataautoformat`

```
MSKrescodee (MSKAPI MSK_readdataautoformat) (
  MSKtask_t task,
  const char * filename)
```

Reads an optimization problem and associated data from a file.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

`MSK_readdatacb`

```
MSKrescodee (MSKAPI MSK_readdatacb) (
  MSKtask_t task,
  MSKhreadfunc hread,
  MSKuserhandle_t h,
  MSKdataformate format,
  MSKcompresstypee compress,
  const char * path)
```

Reads an optimization problem and associated data from a handle.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- hread (*MSKhreadfunc*) – Handle read function. (input)
- h (*MSKuserhandle_t*) – Handle for reading. (input)
- format (*MSKdataformate*) – File data dormat. (input)
- compress (*MSKcompresstypee*) – Data compression type. (input)
- path (char*)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

MSK_readdataformat

```
MSKrescodee (MSKAPI MSK_readdataformat) (
  MSKtask_t task,
  const char * filename,
  MSKdataformate format,
  MSKcompresstypee compress)
```

Reads an optimization problem and associated data from a file.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- filename (char*) – A valid file name. (input)
- format (*MSKdataformate*) – File data format. (input)
- compress (*MSKcompresstypee*) – File compression type. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

MSK_readjsonsol

```
MSKrescodee (MSKAPI MSK_readjsonsol) (
  MSKtask_t task,
  const char * filename)
```

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_t*) – An optimization task. (input)
- filename (char*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

MSK_readjsonstring

```
MSKrescodee (MSKAPI MSK_readjsonstring) (
  MSKtask_t task,
  const char * data)
```

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_t*) – An optimization task. (input)
- `data` (`char*`) – Problem data in text format. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

`MSK_readlpstring`

```
MSKrescodee (MSKAPI MSK_readlpstring) (
  MSKtask_t task,
  const char * data)
```

Load task data from a string in LP format, replacing any data that already exists in the task object.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `data` (`char*`) – Problem data in text format. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

`MSK_readopfstring`

```
MSKrescodee (MSKAPI MSK_readopfstring) (
  MSKtask_t task,
  const char * data)
```

Load task data from a string in OPF format, replacing any data that already exists in the task object.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `data` (`char*`) – Problem data in text format. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

`MSK_readparamfile`

```
MSKrescodee (MSKAPI MSK_readparamfile) (
  MSKtask_t task,
  const char * filename)
```

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*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*, *Parameters*

MSK_readptfstring

```
MSKrescodee (MSKAPI MSK_readptfstring) (
  MSKtask_t task,
  const char * data)
```

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_t*) – An optimization task. (input)
- `data` (`char*`) – Problem data in text format. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

MSK_readsolution

```
MSKrescodee (MSKAPI MSK_readsolution) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const char * filename)
```

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_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

MSK_readsolutionfile

```
MSKrescodee (MSKAPI MSK_readsolutionfile) (
  MSKtask_t task,
  const char * filename)
```

Read solution file in format determined by the filename

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

MSK_readsummary

```
MSKrescodee (MSKAPI MSK_readsummary) (
  MSKtask_t task,
  MSKstreamtypee whichstream)
```

Prints a short summary of last file that was read.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichstream` (*MSKstreamtypee*) – Index of the stream. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*, *Inspecting the task*

`MSK_readtask`

```
MSKrescodee (MSKAPI MSK_readtask) (
  MSKtask_t task,
  const char * filename)
```

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_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

`MSK_removebarvars`

```
MSKrescodee (MSKAPI MSK_removebarvars) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subset)
```

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_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of symmetric matrices which should be removed. (input)
- `subset` (*MSKint32t* *) – Indexes of symmetric matrices which should be removed. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - semidefinite*

~~`MSK_removecones`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_removecones) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subset)
```

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_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of cones which should be removed. (input)
- `subset` (*MSKint32t* *) – Indexes of cones which should be removed. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - cones (deprecated)*

`MSK_removecons`

```
MSKrescodee (MSKAPI MSK_removecons) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subset)
```

The function removes a subset of the constraints from the optimization task. This implies that the remaining constraints are renumbered.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of constraints which should be removed. (input)
- `subset` (*MSKint32t* *) – Indexes of constraints which should be removed. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - constraints*, *Problem data - linear part*

`MSK_removevars`

```
MSKrescodee (MSKAPI MSK_removevars) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subset)
```

The function removes a subset of the variables from the optimization task. This implies that the remaining variables are renumbered.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `num` (*MSKint32t*) – Number of variables which should be removed. (input)
- `subset` (*MSKint32t* *) – Indexes of variables which should be removed. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - variables*, *Problem data - linear part*

`MSK_resetexpirylicenses`

```
MSKrescodee (MSKAPI MSK_resetexpirylicenses) (
  MSKenv_t env)
```

Reset the license expiry reporting startpoint.

Parameters `env` (*MSKenv_t*) – The MOSEK environment. (input)

Return (*MSKrescodee*) – The function response code.

Groups *License system*

`MSK_resizetask`

```
MSKrescodee (MSKAPI MSK_resizetask) (
  MSKtask_t task,
  MSKint32t maxnumcon,
  MSKint32t maxnumvar,
  MSKint32t maxnumcone,
  MSKint64t maxnumanz,
  MSKint64t maxnumqnz)
```

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_t*) – An optimization task. (input)
- `maxnumcon` (*MSKint32t*) – New maximum number of constraints. (input)
- `maxnumvar` (*MSKint32t*) – New maximum number of variables. (input)
- `maxnumcone` (*MSKint32t*) – New maximum number of cones. (input)
- `maxnumanz` (*MSKint64t*) – New maximum number of non-zeros in A. (input)
- `maxnumqnz` (*MSKint64t*) – New maximum number of non-zeros in all Q matrices. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment and task management*

`MSK_sensitivityreport`

```
MSKrescodee (MSKAPI MSK_sensitivityreport) (
  MSKtask_t task,
  MSKstreamtypee whichstream)
```

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_t*) – An optimization task. (input)
- `whichstream` (*MSKstreamtypee*) – Index of the stream. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Sensitivity analysis*

`MSK_setdefaults`

```
MSKrescodee (MSKAPI MSK_setdefaults) (
  MSKtask_t task)
```

Resets all the parameters to their default values.

Parameters `task` (*MSKtask_t*) – An optimization task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*

MSK_sktostr

```
MSKrescodee (MSKAPI MSK_sktostr) (
  MSKtask_t task,
  MSKstakeye sk,
  char * str)
```

Obtains a status key abbreviation string, one of UN, BS, SB, LL, UL, EQ, **.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- sk (*MSKstakeye*) – A valid status key. (input)
- str (char*) – Abbreviation string corresponding to the status key sk. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

MSK_solstatostr

```
MSKrescodee (MSKAPI MSK_solstatostr) (
  MSKtask_t task,
  MSKsolstae solutionsta,
  char * str)
```

Obtains an explanatory string corresponding to a solution status.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- solutionsta (*MSKsolstae*) – Solution status. (input)
- str (char*) – String corresponding to the solution status solsta. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

MSK_solutiondef

```
MSKrescodee (MSKAPI MSK_solutiondef) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKbooleant * isdef)
```

Checks whether a solution is defined.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- isdef (*MSKbooleant by reference*) – Is non-zero if the requested solution is defined. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_solutionsummary

```
MSKrescodee (MSKAPI MSK_solutionsummary) (
  MSKtask_t task,
  MSKstreamtypee whichstream)
```

Prints a short summary of the current solutions.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichstream` (*MSKstreamtypee*) – Index of the stream. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*, *Solution information*

`MSK_solvewithbasis`

```
MSKrescodee (MSKAPI MSK_solvewithbasis) (
  MSKtask_t task,
  MSKbooleant transp,
  MSKint32t numnz,
  MSKint32t * sub,
  MSKrealt * val,
  MSKint32t * numnzout)
```

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 *MSK_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

$$B_{i,k} = A_{i,j}, \ i = 0, \ldots, numcon - 1.$$

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 *MSK_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_t*) – An optimization task. (input)
- `transp` (*MSKbooleant*) – If this argument is zero, then (15.3) is solved, if non-zero then (15.4) is solved. (input)
- `numnz` (*MSKint32t*) – The number of non-zeros in b. (input)
- `sub` (*MSKint32t* *) – 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` (*MSKrealt* *) – 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)
- `numnzout` (*MSKint32t by reference*) – The number of non-zeros in $\overline{X}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solving systems with basis matrix*

`MSK_sparsetriangularsolvedense`

```
MSKrescodee (MSKAPI MSK_sparsetriangularsolvedense) (
  MSKenv_t env,
  MSKtransposee transposed,
  MSKint32t n,
  const MSKint32t * lnzc,
  const MSKint64t * lptrc,
  MSKint64t lensubnval,
  const MSKint32t * lsubc,
  const MSKrealt * lvalc,
  MSKrealt * b)
```

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_t*) – The MOSEK environment. (input)
- `transposed` (*MSKtransposee*) – Controls whether to use with L or L^T. (input)
- `n` (*MSKint32t*) – Dimension of L. (input)
- `lnzc` (*MSKint32t* *) – `lnzc[j]` is the number of nonzeros in column `j`. (input)
- `lptrc` (*MSKint64t* *) – `lptrc[j]` is a pointer to the first row index and value in column `j`. (input)
- `lensubnval` (*MSKint64t*) – Number of elements in `lsubc` and `lvalc`. (input)
- `lsubc` (*MSKint32t* *) – Row indexes for each column stored sequentially. Must be stored in increasing order for each column. (input)
- `lvalc` (*MSKrealt* *) – The value corresponding to the row index stored in `lsubc`. (input)
- `b` (*MSKrealt* *) – The right-hand side of linear equation system to be solved as a dense vector. (input/output)

Return (*MSKrescodee*) – The function response code.

Groups *Linear algebra*

~~`MSK_strtoconetype`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_strtoconetype) (
  MSKtask_t task,
  const char * str,
  MSKconetypee * 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_t*) – An optimization task. (input)

- `str` (`char*`) – String corresponding to the cone type code `conetype`. (input)
- `conetype` (*MSKconetypee by reference*) – The cone type corresponding to the string `str`. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

MSK_strtosk

```
MSKrescodee (MSKAPI MSK_strtosk) (
  MSKtask_t task,
  const char * str,
  MSKstakeye * sk)
```

Obtains the status key corresponding to an abbreviation string.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `str` (`char*`) – A status key abbreviation string. (input)
- `sk` (*MSKstakeye by reference*) – Status key corresponding to the string. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Names*

MSK_syeig

```
MSKrescodee (MSKAPI MSK_syeig) (
  MSKenv_t env,
  MSKuploe uplo,
  MSKint32t n,
  const MSKrealt * a,
  MSKrealt * w)
```

Computes all eigenvalues of a real symmetric matrix A. Given a matrix $A \in \mathbb{R}^{n \times n}$ it returns a vector $w \in \mathbb{R}^n$ containing the eigenvalues of A.

Parameters

- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `uplo` (*MSKuploe*) – Indicates whether the upper or lower triangular part is used. (input)
- `n` (*MSKint32t*) – Dimension of the symmetric input matrix. (input)
- `a` (*MSKrealt**) – A symmetric matrix A stored in column-major order. Only the part indicated by `uplo` is used. (input)
- `w` (*MSKrealt**) – Array of length at least `n` containing the eigenvalues of A. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Linear algebra*

MSK_syevd

```
MSKrescodee (MSKAPI MSK_syevd) (
  MSKenv_t env,
  MSKuploe uplo,
  MSKint32t n,
  MSKrealt * a,
  MSKrealt * w)
```

Computes all the eigenvalues and eigenvectors a real symmetric matrix. Given the input matrix $A \in \mathbb{R}^{n \times n}$, this function returns a vector $w \in \mathbb{R}^n$ containing the eigenvalues of A and it also computes the eigenvectors of A. Therefore, this function computes the eigenvalue decomposition of A as

$$A = UVU^T,$$

where $V = \mathbf{diag}(w)$ and U contains the eigenvectors of A.

Note that the matrix U overwrites the input data A.

Parameters

- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `uplo` (*MSKuploe*) – Indicates whether the upper or lower triangular part is used. (input)
- `n` (*MSKint32t*) – Dimension of the symmetric input matrix. (input)
- `a` (*MSKrealt* *) – A symmetric matrix A stored in column-major order. Only the part indicated by `uplo` is used. On exit it will be overwritten by the matrix U. (input/output)
- `w` (*MSKrealt* *) – Array of length at least `n` containing the eigenvalues of A. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Linear algebra*

`MSK_symnamtovalue`

```
MSKbooleant (MSKAPI MSK_symnamtovalue) (
  const char * name,
  char * value)
```

Obtains the value corresponding to a symbolic name defined by **MOSEK**.

Parameters

- `name` (`char*`) – Symbolic name. (input)
- `value` (`char*`) – The corresponding value. (output)

Return (*MSKbooleant*) – Indicates if the symbolic name has been converted.

Groups *Parameters*

`MSK_syrk`

```
MSKrescodee (MSKAPI MSK_syrk) (
  MSKenv_t env,
  MSKuploe uplo,
  MSKtransposee trans,
  MSKint32t n,
  MSKint32t k,
  MSKrealt alpha,
  const MSKrealt * a,
  MSKrealt beta,
  MSKrealt * c)
```

Performs a symmetric rank-k update for a symmetric matrix.

Given a symmetric matrix $C \in \mathbb{R}^{n \times n}$, two scalars α, β and a matrix A of rank $k \leq n$, it computes either

$$C := \alpha AA^T + \beta C,$$

when `trans` is set to *MSK_TRANSPOSE_NO* and $A \in \mathbb{R}^{n \times k}$, or

$$C := \alpha A^T A + \beta C,$$

when `trans` is set to *MSK_TRANSPOSE_YES* and $A \in \mathbb{R}^{k \times n}$.

Only the part of C indicated by `uplo` is used and only that part is updated with the result. It must not overlap with the other input arrays.

Parameters

- `env` (*MSKenv_t*) – The MOSEK environment. (input)
- `uplo` (*MSKuploe*) – Indicates whether the upper or lower triangular part of C is used. (input)
- `trans` (*MSKtransposee*) – Indicates whether the matrix A must be transposed. (input)
- `n` (*MSKint32t*) – Specifies the order of C. (input)
- `k` (*MSKint32t*) – Indicates the number of rows or columns of A, depending on whether or not it is transposed, and its rank. (input)
- `alpha` (*MSKrealt*) – A scalar value multiplying the result of the matrix multiplication. (input)
- `a` (*MSKrealt* *) – The pointer to the array storing matrix A in a column-major format. (input)
- `beta` (*MSKrealt*) – A scalar value that multiplies C. (input)
- `c` (*MSKrealt* *) – The pointer to the array storing matrix C in a column-major format. (input/output)

Return (*MSKrescodee*) – The function response code.

Groups *Linear algebra*

~~`MSK_toconic`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_toconic) (
  MSKtask_t task)
```

This function tries to reformulate a given Quadratically Constrained Quadratic Optimization problem (QCQO) as a Conic Quadratic Optimization problem (CQO). The first step of the reformulation is to convert the quadratic term of the objective function, if any, into a constraint. Then the following steps are repeated for each quadratic constraint:

- a conic constraint is added along with a suitable number of auxiliary variables and constraints;
- the original quadratic constraint is not removed, but all its coefficients are zeroed out.

Note that the reformulation preserves all the original variables.

The conversion is performed in-place, i.e. the task passed as argument is modified on exit. That also means that if the reformulation fails, i.e. the given QCQP is not representable as a CQO, then the task has an undefined state. In some cases, users may want to clone the task to ensure a clean copy is preserved.

Parameters `task` (*MSKtask_t*) – An optimization task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Problem data - quadratic part*

`MSK_unlinkfuncfromenvstream`

```
MSKrescodee (MSKAPI MSK_unlinkfuncfromenvstream) (
  MSKenv_t env,
  MSKstreamtypee whichstream)
```

Disconnects a user-defined function from a stream.

Parameters
- env (*MSKenv_t*) – The MOSEK environment. (input)
- whichstream (*MSKstreamtypee*) – Index of the stream. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Logging*, *Callback*

MSK_unlinkfuncfromtaskstream

```
MSKrescodee (MSKAPI MSK_unlinkfuncfromtaskstream) (
  MSKtask_t task,
  MSKstreamtypee whichstream)
```

Disconnects a user-defined function from a task stream.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- whichstream (*MSKstreamtypee*) – Index of the stream. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Logging*, *Callback*

MSK_updatesolutioninfo

```
MSKrescodee (MSKAPI MSK_updatesolutioninfo) (
  MSKtask_t task,
  MSKsoltypee whichsol)
```

Update the information items related to the solution.

Parameters
- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Information items and statistics*

MSK_utf8towchar

```
MSKrescodee (MSKAPI MSK_utf8towchar) (
  const size_t outputlen,
  size_t * len,
  size_t * conv,
  MSKwchart * output,
  const char * input)
```

Converts an UTF8 string to a *MSKwchart* string.

Parameters
- outputlen (size_t) – The length of the output buffer. (input)
- len (size_t*) – The length of the string contained in the output buffer. (output)
- conv (size_t*) – Returns the number of characters converted, i.e. input[conv] is the first character which was not converted. If the whole string was converted, then input[conv]=0. (output)
- output (*MSKwchart**) – The input string converted to a *MSKwchart* string. (output)
- input (char*) – The UTF8 input string. (input)

Return (*MSKrescodee*) – The function response code.

Groups *System, memory and debugging*

MSK_wchartoutf8

```
MSKrescodee (MSKAPI MSK_wchartoutf8) (
  const size_t outputlen,
  size_t * len,
  size_t * conv,
  char * output,
  const MSKwchart * input)
```

Converts a *MSKwchart* string to an UTF8 string.

Parameters

- `outputlen` (`size_t`) – The length of the output buffer. (input)
- `len` (`size_t*`) – The length of the string contained in the output buffer. (output)
- `conv` (`size_t*`) – Returns the number of characters from converted, i.e. `input[conv]` is the first char which was not converted. If the whole string was converted, then `input[conv]=0`. (output)
- `output` (`char*`) – The input string converted to a UTF8 string. (output)
- `input` (*MSKwchart**) – The *MSKwchart* input string. (input)

Return (*MSKrescodee*) – The function response code.

Groups *System, memory and debugging*

MSK_whichparam

```
MSKrescodee (MSKAPI MSK_whichparam) (
  MSKtask_t task,
  const char * parname,
  MSKparametertypee * partype,
  MSKint32t * param)
```

Checks if `parname` is a valid parameter name. If yes then `partype` and `param` denote the type and the index of the parameter, respectively.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `parname` (`char*`) – Parameter name. (input)
- `partype` (*MSKparametertypee by reference*) – Parameter type. (output)
- `param` (*MSKint32t by reference*) – Which parameter. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters*, *Names*

MSK_writebsolution

```
MSKrescodee (MSKAPI MSK_writebsolution) (
  MSKtask_t task,
  const char * filename,
  MSKcompresstypee compress)
```

Write a binary dump of the task solution.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)

- compress (*MSKcompresstypee*) – Data compression type. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

MSK_writedata

```
MSKrescodee (MSKAPI MSK_writedata) (
  MSKtask_t task,
  const char * filename)
```

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_t*) – An optimization task. (input)
- filename (char*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

MSK_writedatahandle

```
MSKrescodee (MSKAPI MSK_writedatahandle) (
  MSKtask_t task,
  MSKhwritefunc func,
  MSKuserhandle_t handle,
  MSKdataformate format,
  MSKcompresstypee compress)
```

Writes an optimization problem and associated data using a user-defined stream write function `func`.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- func (*MSKhwritefunc*) – A user-defined write function which receives the data. (input)
- handle (*MSKuserhandle_t*) – A user-defined handle which is passed to the user-defined function `func` (can be NULL). (input)
- format (*MSKdataformate*) – Selects data format. (input)
- compress (*MSKcompresstypee*) – Selects compression type. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Input/Output*

MSK_writejsonsol

```
MSKrescodee (MSKAPI MSK_writejsonsol) (
  MSKtask_t task,
  const char * filename)
```

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_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Input/Output*

`MSK_writeparamfile`

```
MSKrescodee (MSKAPI MSK_writeparamfile) (
  MSKtask_t task,
  const char * filename)
```

Writes all the parameters to a parameter file.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Input/Output*, *Parameters*

`MSK_writesolution`

```
MSKrescodee (MSKAPI MSK_writesolution) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  const char * filename)
```

Saves the current basic, interior-point, or integer solution to a file.

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `whichsol` (*MSKsoltypee*) – Selects a solution. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Input/Output*

`MSK_writesolutionfile`

```
MSKrescodee (MSKAPI MSK_writesolutionfile) (
  MSKtask_t task,
  const char * filename)
```

Write solution file in format determined by the filename

Parameters
- `task` (*MSKtask_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.
Groups *Input/Output*

`MSK_writetask`

```
MSKrescodee (MSKAPI MSK_writetask) (
  MSKtask_t task,
  const char * filename)
```

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_t*) – An optimization task. (input)
- `filename` (`char*`) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

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*
- *MSK_IPAR_CHECK_CONVEXITY*
- *MSK_IPAR_LOG_CHECK_CONVEXITY*

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_SOLUTIONS*
- *MSK_IPAR_PTF_WRITE_TRANSFORM*
- *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_BAS_CONSTRAINTS*
- *MSK_IPAR_WRITE_BAS_HEAD*
- *MSK_IPAR_WRITE_BAS_VARIABLES*
- *MSK_IPAR_WRITE_COMPRESSION*
- *MSK_IPAR_WRITE_DATA_PARAM*
- *MSK_IPAR_WRITE_FREE_CON*
- *MSK_IPAR_WRITE_GENERIC_NAMES*
- *MSK_IPAR_WRITE_GENERIC_NAMES_IO*
- *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_IPAR_WRITE_TASK_INC_SOL*
- *MSK_IPAR_WRITE_XML_MODE*
- *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*
- *MSK_SPAR_WRITE_LP_GEN_VAR_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_CHECK_CONVEXITY_REL_TOL*
- *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_MAX_NUM_REFINEMENT_STEPS*
- *MSK_IPAR_INTPNT_OFF_COL_TRH*
- *MSK_IPAR_INTPNT_ORDER_GP_NUM_SEEDS*
- *MSK_IPAR_INTPNT_ORDER_METHOD*
- *MSK_IPAR_INTPNT_PURIFY*
- *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_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_RESPONSE*
- *MSK_IPAR_LOG_SENSITIVITY*
- *MSK_IPAR_LOG_SENSITIVITY_OPT*
- *MSK_IPAR_LOG_SIM*
- *MSK_IPAR_LOG_SIM_FREQ*
- *MSK_IPAR_LOG_STORAGE*

Mixed-integer optimization

- *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_CONIC_OUTER_APPROXIMATION*
- *MSK_IPAR_MIO_CONSTRUCT_SOL*
- *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_FEASPUMP_LEVEL*
- *MSK_IPAR_MIO_HEURISTIC_LEVEL*
- *MSK_IPAR_MIO_MAX_NUM_BRANCHES*
- *MSK_IPAR_MIO_MAX_NUM_RELAXS*
- *MSK_IPAR_MIO_MAX_NUM_ROOT_CUT_ROUNDS*
- *MSK_IPAR_MIO_MAX_NUM_SOLUTIONS*
- *MSK_IPAR_MIO_MEMORY_EMPHASIS_LEVEL*
- *MSK_IPAR_MIO_NODE_OPTIMIZER*
- *MSK_IPAR_MIO_NODE_SELECTION*
- *MSK_IPAR_MIO_NUMERICAL_EMPHASIS_LEVEL*
- *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_RINS_MAX_NODES*
- *MSK_IPAR_MIO_ROOT_OPTIMIZER*
- *MSK_IPAR_MIO_ROOT_REPEAT_PRESOLVE_LEVEL*
- *MSK_IPAR_MIO_SEED*
- *MSK_IPAR_MIO_SYMMETRY_LEVEL*
- *MSK_IPAR_MIO_VB_DETECTION_LEVEL*

Output information

- *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_RESPONSE*
- *MSK_IPAR_LOG_SENSITIVITY*
- *MSK_IPAR_LOG_SENSITIVITY_OPT*
- *MSK_IPAR_LOG_SIM*
- *MSK_IPAR_LOG_SIM_FREQ*
- *MSK_IPAR_LOG_SIM_MINOR*
- *MSK_IPAR_LOG_STORAGE*
- *MSK_IPAR_MAX_NUM_WARNINGS*

Overall solver

- *MSK_IPAR_BI_CLEAN_OPTIMIZER*
- *MSK_IPAR_INFEAS_PREFER_PRIMAL*
- *MSK_IPAR_LICENSE_WAIT*
- *MSK_IPAR_MIO_MODE*
- *MSK_IPAR_OPTIMIZER*
- *MSK_IPAR_PRESOLVE_LEVEL*
- *MSK_IPAR_PRESOLVE_MAX_NUM_REDUCTIONS*
- *MSK_IPAR_PRESOLVE_USE*
- *MSK_IPAR_PRIMAL_REPAIR_OPTIMIZER*
- *MSK_IPAR_SENSITIVITY_ALL*
- *MSK_IPAR_SENSITIVITY_OPTIMIZER*
- *MSK_IPAR_SENSITIVITY_TYPE*
- *MSK_IPAR_SOLUTION_CALLBACK*

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_PRESOLVE_TOL_ABS_LINDEP*
- *MSK_DPAR_PRESOLVE_TOL_AIJ*
- *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_MIO_PRESOLVE_AGGREGATOR_USE*
- *MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_FILL*

- *MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_NUM_TRIES*
- *MSK_IPAR_PRESOLVE_LEVEL*
- *MSK_IPAR_PRESOLVE_LINDEP_ABS_WORK_TRH*
- *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*

Progress callback

- *MSK_IPAR_SOLUTION_CALLBACK*

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_SIMPLEX_ABS_TOL_PIV*
- *MSK_IPAR_BASIS_SOLVE_USE_PLUS_ONE*
- *MSK_IPAR_LOG_SIM*
- *MSK_IPAR_LOG_SIM_FREQ*
- *MSK_IPAR_LOG_SIM_MINOR*
- *MSK_IPAR_SENSITIVITY_OPTIMIZER*
- *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_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_STABILITY_PRIORITY*
- *MSK_IPAR_SIM_SWITCH_OPTIMIZER*

Solution input/output

- *MSK_IPAR_INFEAS_REPORT_AUTO*
- *MSK_IPAR_SOL_FILTER_KEEP_BASIC*
- *MSK_IPAR_SOL_FILTER_KEEP_RANGED*
- *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_TIME*
- *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_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

`MSKdparame`
: 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 `MSK_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 `MSK_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 `MSK_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 `MSK_putdouparam(task, MSK_DPAR_BASIS_TOL_X, 1.0e-6)`
  Groups *Simplex optimizer*, *Termination criteria*

`MSK_DPAR_CHECK_CONVEXITY_REL_TOL`
: This parameter controls when the full convexity check declares a problem to be non-convex. Increasing this tolerance relaxes the criteria for declaring the problem non-convex.

  A problem is declared non-convex if negative (positive) pivot elements are detected in the Cholesky factor of a matrix which is required to be PSD (NSD). This parameter controls how much this non-negativity requirement may be violated.

  If d_i is the pivot element for column i, then the matrix Q is considered to not be PSD if:

$$d_i \leq -|Q_{ii}|\texttt{check\_convexity\_rel\_tol}$$

Default 1e-10
Accepted [0; +inf]
Example `MSK_putdouparam(task, MSK_DPAR_CHECK_CONVEXITY_REL_TOL, 1e-10)`
Groups *Interior-point method*

`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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_putdouparam(task, MSK_DPAR_DATA_TOL_X, 1.0e-8)`

Groups *Data check*

`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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 -1.0e30

Accepted [-inf; +inf]

Example `MSK_putdouparam(task, MSK_DPAR_LOWER_OBJ_CUT, -1.0e30)`

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 `MSK_putdouparam(task, MSK_DPAR_LOWER_OBJ_CUT_FINITE_TRH, -0.5e30)`

Groups *Termination criteria*

`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 `MSK_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. A negative number means infinity.

Default -1.0

Accepted [-inf; +inf]

Example `MSK_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 an integer optimization problem.

Default 1.0e-10

Accepted [1.0e-15; +inf]

Example `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_putdouparam(task, MSK_DPAR_MIO_TOL_REL_GAP, 1.0e-4)`

Groups *Mixed-integer optimization*, *Termination criteria*

`MSK_DPAR_OPTIMIZER_MAX_TIME`

Maximum amount of time the optimizer is allowed to spent on the optimization. A negative number means infinity.

Default -1.0

Accepted [-inf; +inf]

Example `MSK_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 `MSK_putdouparam(task, MSK_DPAR_PRESOLVE_TOL_ABS_LINDEP, 1.0e-6)`

Groups *Presolve*

`MSK_DPAR_PRESOLVE_TOL_AIJ`

Absolute zero tolerance employed for a_{ij} in the presolve.

Default 1.0e-12

Accepted [1.0e-15; +inf]

Example `MSK_putdouparam(task, MSK_DPAR_PRESOLVE_TOL_AIJ, 1.0e-12)`

Groups *Presolve*

`MSK_DPAR_PRESOLVE_TOL_PRIMAL_INFEAS_PERTURBATION`

The presolve is allowed to perturbe a bound on a constraint or variable by this amount if it removes an infeasibility.

Default 1.0e-6

Accepted [0.0; +inf]

Example `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_putdouparam(task, MSK_DPAR_SIM_LU_TOL_REL_PIV, 0.01)`

Groups *Basis identification*, *Simplex optimizer*

`MSK_DPAR_SIMPLEX_ABS_TOL_PIV`

Absolute pivot tolerance employed by the simplex optimizers.

Default 1.0e-7

Accepted [1.0e-12; +inf]

Example `MSK_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 1.0e30

Accepted [-inf; +inf]

Example `MSK_putdouparam(task, MSK_DPAR_UPPER_OBJ_CUT, 1.0e30)`

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 `MSK_putdouparam(task, MSK_DPAR_UPPER_OBJ_CUT_FINITE_TRH, 0.5e30)`

Groups *Termination criteria*

15.5.2 Integer parameters

`MSKiparame`

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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSK_solvewithbasis* function.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_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*, *FREE_SIMPLEX*, *MIXED_INT* (see *MSKoptimizertypee*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 `MSK_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 *MSK_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 *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_CACHE_LICENSE, MSK_ON)`

Groups *License manager*

`MSK_IPAR_CHECK_CONVEXITY`

Specify the level of convexity check on quadratic problems.

Default *FULL*

Accepted *NONE*, *SIMPLE*, *FULL* (see *MSKcheckconvexitytypee*)

Example `MSK_putintparam(task, MSK_IPAR_CHECK_CONVEXITY, MSK_CHECK_CONVEXITY_FULL)`

Groups *Data check*

`MSK_IPAR_COMPRESS_STATFILE`

Control compression of stat files.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_COMPRESS_STATFILE, MSK_ON)`

`MSK_IPAR_INFEAS_GENERIC_NAMES`

Controls whether generic names are used when an infeasible subproblem is created.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_INFEAS_GENERIC_NAMES, MSK_OFF)`

Groups *Infeasibility report*

`MSK_IPAR_INFEAS_PREFER_PRIMAL`

If both certificates of primal and dual infeasibility are supplied then only the primal is used when this option is turned on.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_INFEAS_PREFER_PRIMAL, MSK_ON)`

Groups *Overall solver*

`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 *MSKonoffkeye*)

Example `MSK_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 `MSK_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 *MSKbasindtypee*)

Example `MSK_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 `MSK_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 *MSKintpnthotstarte*)

Example `MSK_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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_INTPNT_MAX_NUM_COR, -1)`

Groups *Interior-point method*

MSK_IPAR_INTPNT_MAX_NUM_REFINEMENT_STEPS

Maximum number of steps to be used by the iterative refinement of the search direction. A negative value implies that the optimizer chooses the maximum number of iterative refinement steps.

Default -1

Accepted [-inf; +inf]

Example `MSK_putintparam(task, MSK_IPAR_INTPNT_MAX_NUM_REFINEMENT_STEPS, -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 `MSK_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 `MSK_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 *MSKorderingtypee*)

Example `MSK_putintparam(task, MSK_IPAR_INTPNT_ORDER_METHOD, MSK_ORDER_METHOD_FREE)`

Groups *Interior-point method*

MSK_IPAR_INTPNT_PURIFY

Currently not in use.

Default *NONE*

Accepted *NONE*, *PRIMAL*, *DUAL*, *PRIMAL_DUAL*, *AUTO* (see *MSKpurifye*)

Example `MSK_putintparam(task, MSK_IPAR_INTPNT_PURIFY, MSK_PURIFY_NONE)`

Groups *Interior-point method*

MSK_IPAR_INTPNT_REGULARIZATION_USE

Controls whether regularization is allowed.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_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 *MSKscalingtypee*)

Example `MSK_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 *MSKsolveforme*)

Example `MSK_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*, *SATISFY_BOUNDS* (see *MSKstartpointtypee*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 `MSK_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 *MSKonoffkeye*)

Example `MSK_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 `MSK_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 *MSKonoffkeye*)

Example `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_LOG_BI_FREQ, 2500)`

Groups *Basis identification*, *Output information*, *Logging*

`MSK_IPAR_LOG_CHECK_CONVEXITY`

Controls logging in convexity check on quadratic problems. Set to a positive value to turn logging on. If a quadratic coefficient matrix is found to violate the requirement of PSD (NSD) then a list of negative (positive) pivot elements is printed. The absolute value of the pivot elements is also shown.

Default 0

Accepted [0; +inf]

Example `MSK_putintparam(task, MSK_IPAR_LOG_CHECK_CONVEXITY, 0)`

Groups *Data check*

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 `MSK_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 0

Accepted [0; +inf]

Example `MSK_putintparam(task, MSK_IPAR_LOG_EXPAND, 0)`

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 `MSK_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 `MSK_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 *MSK_optimize*, so a separate call to *MSK_solutionsummary* is not necessary.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_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 `MSK_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 `MSK_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 *`MSKonoffkeye`*)
Example `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_LOG_PRESOLVE, 1)`
Groups *Logging*

`MSK_IPAR_LOG_RESPONSE`

Controls amount of output printed when response codes are reported. A higher level implies that more information is logged.

Default 0

Accepted [0; +inf]
Example `MSK_putintparam(task, MSK_IPAR_LOG_RESPONSE, 0)`
Groups *Output information*, *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 `MSK_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 `MSK_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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_LOG_SIM_FREQ, 1000)`
Groups *Simplex optimizer*, *Output information*, *Logging*

`MSK_IPAR_LOG_SIM_MINOR`
Currently not in use.

Default 1
Accepted [0; +inf]
Example `MSK_putintparam(task, MSK_IPAR_LOG_SIM_MINOR, 1)`
Groups *Simplex optimizer*, *Output information*

`MSK_IPAR_LOG_STORAGE`
When turned on, **MOSEK** prints messages regarding the storage usage and allocation.

Default 0
Accepted [0; +inf]
Example `MSK_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 `MSK_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 *`MSKbranchdire`*)
Example `MSK_putintparam(task, MSK_IPAR_MIO_BRANCH_DIR, MSK_BRANCH_DIR_FREE)`
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 *`MSKonoffkeye`*)
Example `MSK_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 *`MSKonoffkeye`*)
Example `MSK_putintparam(task, MSK_IPAR_MIO_CONSTRUCT_SOL, MSK_OFF)`
Groups *Mixed-integer optimization*

`MSK_IPAR_MIO_CUT_CLIQUE`
Controls whether clique cuts should be generated.

Default *`ON`*
Accepted *`ON`*, *`OFF`* (see *`MSKonoffkeye`*)
Example `MSK_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 *`MSKonoffkeye`*)
Example `MSK_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 *`MSKonoffkeye`*)

Example `MSK_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 *MSKonoffkeye*)
Example `MSK_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 *OFF*
Accepted *ON*, *OFF* (see *MSKonoffkeye*)
Example `MSK_putintparam(task, MSK_IPAR_MIO_CUT_KNAPSACK_COVER, MSK_OFF)`
Groups *Mixed-integer optimization*

`MSK_IPAR_MIO_CUT_LIPRO`
Controls whether lift-and-project cuts should be generated.

Default *OFF*
Accepted *ON*, *OFF* (see *MSKonoffkeye*)
Example `MSK_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 `MSK_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 *MSKmiodatapermmethode*)
Example `MSK_putintparam(task, MSK_IPAR_MIO_DATA_PERMUTATION_METHOD, MSK_MIO_DATA_PERMUTATION_METHOD_NONE)`
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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_MIO_HEURISTIC_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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_MIO_MAX_NUM_RELAXS, -1)`

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 `MSK_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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_MIO_MEMORY_EMPHASIS_LEVEL, 0)`

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 *MSKmiomodee*)

Example `MSK_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*, *FREE_SIMPLEX*, *MIXED_INT* (see *MSKoptimizertypee*)

Example `MSK_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 *MSKmionodeseltypee*)

Example `MSK_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 `MSK_putintparam(task, MSK_IPAR_MIO_NUMERICAL_EMPHASIS_LEVEL, 0)`

Groups *Mixed-integer optimization*

`MSK_IPAR_MIO_PERSPECTIVE_REFORMULATE`

Enables or disables perspective reformulation in presolve.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 `MSK_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 *`MSKonoffkeye`*)

Example `MSK_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 *`MSKmiqcqoreformmethode`*)

Example `MSK_putintparam(task, MSK_IPAR_MIO_QCQO_REFORMULATION_METHOD, MSK_MIO_QCQO_REFORMULATION_METHOD_FREE)`

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 `MSK_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`*, *`FREE_SIMPLEX`*, *`MIXED_INT`* (see *`MSKoptimizertypee`*)

Example `MSK_putintparam(task, MSK_IPAR_MIO_ROOT_OPTIMIZER, MSK_OPTIMIZER_FREE)`

Groups *Mixed-integer optimization*

`MSK_IPAR_MIO_ROOT_REPEAT_PRESOLVE_LEVEL`

Controls whether presolve can be repeated at root node.

- −1. The optimizer chooses whether presolve is repeated
- 0. Never repeat presolve
- 1. Always repeat presolve

Default -1
Accepted [-1; 1]
Example `MSK_putintparam(task, MSK_IPAR_MIO_ROOT_REPEAT_PRESOLVE_LEVEL, -1)`
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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_MIO_SYMMETRY_LEVEL, -1)`
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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_MT_SPINCOUNT, 0)`
Groups *Overall system*

`MSK_IPAR_NG`

Not in use.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_NG, MSK_OFF)`

`MSK_IPAR_NUM_THREADS`

Controls the number of threads employed by the optimizer. If set to 0 the number of threads used will be equal to the number of cores detected on the machine.

Default 0

Accepted [0; +inf]

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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*, *FREE_SIMPLEX*, *MIXED_INT* (see *MSKoptimizertypee*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_NUM_TRIES, -1)`

Groups *Presolve*

`MSK_IPAR_PRESOLVE_LEVEL`

Currently not used.

Default -1

Accepted [-inf; +inf]

Example `MSK_putintparam(task, MSK_IPAR_PRESOLVE_LEVEL, -1)`

Groups *Overall solver*, *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 `MSK_putintparam(task, MSK_IPAR_PRESOLVE_LINDEP_ABS_WORK_TRH, 100)`

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 `MSK_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 *MSKonoffkeye*)

Example `MSK_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 `MSK_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 `MSK_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 *MSKpresolvemodee*)

Example `MSK_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*, *FREE_SIMPLEX*, *MIXED_INT* (see *MSKoptimizertypee*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_PTF_WRITE_PARAMETERS, 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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_PTF_WRITE_TRANSFORM, MSK_ON)`

Groups *Data input/output*

`MSK_IPAR_READ_DEBUG`

Turns on additional debugging information when reading files.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_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.

Default *OFF*

Accepted

- *ON*: The free constraints are kept.
- *OFF*: The free constraints are discarded.

Example `MSK_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 *MSKmpsformate*)

Example `MSK_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 `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKcompresstypee*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_REMOVE_UNUSED_SOLUTIONS, MSK_OFF)`

Groups *Overall system*

`MSK_IPAR_SENSITIVITY_ALL`

If set to *MSK_ON*, then *MSK_sensitivityreport* analyzes all bounds and variables instead of reading a specification from the file.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_SENSITIVITY_ALL, MSK_OFF)`

Groups *Overall solver*

`MSK_IPAR_SENSITIVITY_OPTIMIZER`

Controls which optimizer is used for optimal partition sensitivity analysis.

Default *FREE_SIMPLEX*

Accepted *FREE*, *INTPNT*, *CONIC*, *PRIMAL_SIMPLEX*, *DUAL_SIMPLEX*, *FREE_SIMPLEX*, *MIXED_INT* (see *MSKoptimizertypee*)

Example `MSK_putintparam(task, MSK_IPAR_SENSITIVITY_OPTIMIZER, MSK_OPTIMIZER_FREE_SIMPLEX)`

Groups *Overall solver*, *Simplex optimizer*

`MSK_IPAR_SENSITIVITY_TYPE`

Controls which type of sensitivity analysis is to be performed.

Default *BASIS*

Accepted *BASIS* (see *MSKsensitivitytypee*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKsimdegene*)

Example `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 *`MSKsimseltypee`*)

Example `MSK_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 *`MSKsimdupvece`*)

Example `MSK_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 *`MSKsimhotstarte`*)

Example `MSK_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 `MSK_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 `MSK_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 `MSK_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 *MSKonoffkeye*)
Example `MSK_putintparam(task, MSK_IPAR_SIM_NON_SINGULAR, MSK_ON)`
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 `MSK_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 `MSK_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 `MSK_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 *MSKsimseltypee*)
Example `MSK_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 `MSK_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 *MSKsimreforme*)
Example `MSK_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 *MSKonoffkeye*)
Example `MSK_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 *MSKscalingtypee*)
Example `MSK_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 *MSKscalingmethode*)
Example `MSK_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 `MSK_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 *`MSKsolveforme`*)
Example `MSK_putintparam(task, MSK_IPAR_SIM_SOLVE_FORM, MSK_SOLVE_FREE)`
Groups *Simplex optimizer*

`MSK_IPAR_SIM_STABILITY_PRIORITY`

Controls how high priority the numerical stability should be given.

Default 50
Accepted [0; 100]
Example `MSK_putintparam(task, MSK_IPAR_SIM_STABILITY_PRIORITY, 50)`
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 *`MSKonoffkeye`*)
Example `MSK_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 *`MSKonoffkeye`*)
Example `MSK_putintparam(task, MSK_IPAR_SOL_FILTER_KEEP_BASIC, MSK_OFF)`
Groups *Solution input/output*

`MSK_IPAR_SOL_FILTER_KEEP_RANGED`

If turned on, then ranged constraints and variables are written to the solution file independent of the filter setting.

Default *`OFF`*
Accepted *`ON`*, *`OFF`* (see *`MSKonoffkeye`*)
Example `MSK_putintparam(task, MSK_IPAR_SOL_FILTER_KEEP_RANGED, 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 `MSK_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 `MSK_putintparam(task, MSK_IPAR_SOL_READ_WIDTH, 1024)`
Groups *Data input/output*, *Solution input/output*

`MSK_IPAR_SOLUTION_CALLBACK`
Indicates whether solution callbacks will be performed during the optimization.

Default *OFF*
Accepted *ON*, *OFF* (see *MSKonoffkeye*)
Example `MSK_putintparam(task, MSK_IPAR_SOLUTION_CALLBACK, MSK_OFF)`
Groups *Progress callback*, *Overall solver*

`MSK_IPAR_TIMING_LEVEL`
Controls the amount of timing performed inside **MOSEK**.

Default 1
Accepted [0; +inf]
Example `MSK_putintparam(task, MSK_IPAR_TIMING_LEVEL, 1)`
Groups *Overall system*

`MSK_IPAR_WRITE_BAS_CONSTRAINTS`
Controls whether the constraint section is written to the basic solution file.

Default *ON*
Accepted *ON*, *OFF* (see *MSKonoffkeye*)
Example `MSK_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 *MSKonoffkeye*)
Example `MSK_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 *MSKonoffkeye*)
Example `MSK_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 `MSK_putintparam(task, MSK_IPAR_WRITE_COMPRESSION, 9)`
Groups *Data input/output*

`MSK_IPAR_WRITE_DATA_PARAM`
If this option is turned on the parameter settings are written to the data file as parameters.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_WRITE_DATA_PARAM, MSK_OFF)`

Groups *Data input/output*

`MSK_IPAR_WRITE_FREE_CON`

Controls whether the free constraints are written to the data file.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_WRITE_GENERIC_NAMES, MSK_OFF)`

Groups *Data input/output*

`MSK_IPAR_WRITE_GENERIC_NAMES_IO`

Index origin used in generic names.

Default 1

Accepted [0; +inf]

Example `MSK_putintparam(task, MSK_IPAR_WRITE_GENERIC_NAMES_IO, 1)`

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 `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 `MSK_putintparam(task, MSK_IPAR_WRITE_LP_LINE_WIDTH, 80)`

Groups *Data input/output*

`MSK_IPAR_WRITE_MPS_FORMAT`

Controls in which format the MPS is written.

Default *FREE*

Accepted *STRICT*, *RELAXED*, *FREE*, *CPLEX* (see *MSKmpsformate*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_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 *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_WRITE_SOL_VARIABLES, MSK_ON)`

Groups *Data input/output*, *Solution input/output*

`MSK_IPAR_WRITE_TASK_INC_SOL`

Controls whether the solutions are stored in the task file too.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Example `MSK_putintparam(task, MSK_IPAR_WRITE_TASK_INC_SOL, MSK_ON)`

Groups *Data input/output*

`MSK_IPAR_WRITE_XML_MODE`

Controls if linear coefficients should be written by row or column when writing in the XML file format.

Default *ROW*

Accepted *ROW*, *COL* (see *MSKxmlwriteroutputtypee*)

Example `MSK_putintparam(task, MSK_IPAR_WRITE_XML_MODE, MSK_WRITE_XML_MODE_ROW)`

Groups *Data input/output*

15.5.3 String parameters

`MSKsparame`

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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_putstrparam(task, MSK_SPAR_REMOTE_TLS_CERT_PATH, "somevalue")`
Groups *Overall system*

`MSK_SPAR_SENSITIVITY_FILE_NAME`
If defined *`MSK_sensitivityreport`* reads this file as a sensitivity analysis data file specifying the type of analysis to be done.

Accepted Any valid string.

Example `MSK_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 *`MSK_sensitivityreport`* writes results to this file.

Accepted Any valid string.

Example `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_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 `MSK_putstrparam(task, MSK_SPAR_STAT_NAME, "somevalue")`

Groups *Data input/output*

`MSK_SPAR_WRITE_LP_GEN_VAR_NAME`

Sometimes when an LP file is written additional variables must be inserted. They will have the prefix denoted by this parameter.

Default xmskgen

Accepted Any valid string.

Example `MSK_putstrparam(task, MSK_SPAR_WRITE_LP_GEN_VAR_NAME, "xmskgen")`

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.

`MSKrescodee`

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.

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 is not valid. For instance when writing an LP file the names must not contain blanks or start with a digit.

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_PERTUBATIONS (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_WRITE_LP_INVALID_VAR_NAMES (854)
: LP file will be written with generic variable names.

MSK_RES_WRN_WRITE_LP_DUPLICATE_VAR_NAMES (855)
: LP file will be written with generic variable names.

MSK_RES_WRN_WRITE_LP_INVALID_CON_NAMES (856)
: LP file will be written with generic constraint names.

MSK_RES_WRN_WRITE_LP_DUPLICATE_CON_NAMES (857)
: LP file will be written with generic constraint names.

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_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.

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 reached.

`MSK_RES_ERR_NULL_ENV (1060)`
: `env` is a NULL pointer.

`MSK_RES_ERR_NULL_TASK (1061)`
: `task` is a NULL 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 NULL 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_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_FORMAT (1090)`
: The specified format cannot be read.

`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 `QSECTION`s 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 `CSECTION`s 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_INCOMPATIBLE (1150)
The problem cannot be written to an LP formatted file.

MSK_RES_ERR_LP_EMPTY (1151)
The problem cannot be written to an LP formatted file.

MSK_RES_ERR_LP_DUP_SLACK_NAME (1152)
The name of the slack variable added to a ranged constraint already exists.

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_LP_FREE_CONSTRAINT (1155)
Free constraints cannot be written in LP file format.

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_WRITE_LP_FORMAT (1158)`
Problem cannot be written as an 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_WRITE_LP_NON_UNIQUE_NAME (1161)`
An auto-generated name is not unique.

`MSK_RES_ERR_READ_LP_NONEXISTING_NAME (1162)`
A variable never occurred in objective or constraints.

`MSK_RES_ERR_LP_WRITE_CONIC_PROBLEM (1163)`
The problem contains cones that cannot be written to an LP formatted file.

`MSK_RES_ERR_LP_WRITE_GECO_PROBLEM (1164)`
The problem contains general convex terms that cannot be written to an LP formatted file.

`MSK_RES_ERR_WRITING_FILE (1166)`
An error occurred while writing file

`MSK_RES_ERR_INVALID_NAME_IN_SOL_FILE (1170)`
An invalid name occurred in a solution file.

`MSK_RES_ERR_LP_INVALID_CON_NAME (1171)`
A constraint name is invalid when used in an LP formatted 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 is 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_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. The parameter *MSK_DPAR_CHECK_CONVEXITY_REL_TOL* can be used to relax the convexity check.

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. The parameter *MSK_DPAR_CHECK_CONVEXITY_REL_TOL* can be used to relax the convexity check.

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. The parameter *MSK_DPAR_CHECK_CONVEXITY_REL_TOL* can be used to relax the convexity check.

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. The parameter *MSK_DPAR_CHECK_CONVEXITY_REL_TOL* can be used to relax the convexity check.

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).

MSK_RES_ERR_UPPER_BOUND_IS_A_NAN (1391)
: The upper bound specified is not a number (nan).

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`.

`MSK_RES_ERR_NAN_IN_BUC (1462)`
: u^c contains an invalid floating point value, i.e. a `NaN`.

`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`.

`MSK_RES_ERR_NAN_IN_BLX (1471)`
: l^x contains an invalid floating point value, i.e. a `NaN`.

`MSK_RES_ERR_NAN_IN_BUX (1472)`
: u^x contains an invalid floating point value, i.e. a `NaN`.

`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 NULL 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_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_XML_INVALID_PROBLEM_TYPE (3600)`
: The problem type is not supported by the XML format.

`MSK_RES_ERR_INVALID_AMPL_STUB (3700)`
: Invalid AMPL stub.

`MSK_RES_ERR_INT64_TO_INT32_CAST (3800)`
: A 64 bit integer could not be cast to a 32 bit integer.

`MSK_RES_ERR_SIZE_LICENSE_NUMCORES (3900)`
: The computer contains more cpu cores than the license allows for.

`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 conic constraints.

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_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_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_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

`MSKbasindtypee`
: Basis identification

`MSK_BI_NEVER (0)`
: Never do basis identification.

`MSK_BI_ALWAYS (1)`
: Basis identification is always performed even if the interior-point optimizer terminates abnormally.

`MSK_BI_NO_ERROR (2)`
: Basis identification is performed if the interior-point optimizer terminates without an error.

`MSK_BI_IF_FEASIBLE (3)`
: 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 (4)`
: Not currently in use.

`MSKboundkeye`
: Bound keys

`MSK_BK_LO (0)`
: The constraint or variable has a finite lower bound and an infinite upper bound.

MSK_BK_UP (1)
: The constraint or variable has an infinite lower bound and an finite upper bound.

MSK_BK_FX (2)
: The constraint or variable is fixed.

MSK_BK_FR (3)
: The constraint or variable is free.

MSK_BK_RA (4)
: The constraint or variable is ranged.

MSKmarke

Mark

MSK_MARK_LO (0)
: The lower bound is selected for sensitivity analysis.

MSK_MARK_UP (1)
: The upper bound is selected for sensitivity analysis.

MSKsimdegene

Degeneracy strategies

MSK_SIM_DEGEN_NONE (0)
: The simplex optimizer should use no degeneration strategy.

MSK_SIM_DEGEN_FREE (1)
: The simplex optimizer chooses the degeneration strategy.

MSK_SIM_DEGEN_AGGRESSIVE (2)
: The simplex optimizer should use an aggressive degeneration strategy.

MSK_SIM_DEGEN_MODERATE (3)
: The simplex optimizer should use a moderate degeneration strategy.

MSK_SIM_DEGEN_MINIMUM (4)
: The simplex optimizer should use a minimum degeneration strategy.

MSKtransposee

Transposed matrix.

MSK_TRANSPOSE_NO (0)
: No transpose is applied.

MSK_TRANSPOSE_YES (1)
: A transpose is applied.

MSKuploe

Triangular part of a symmetric matrix.

MSK_UPLO_LO (0)
: Lower part.

MSK_UPLO_UP (1)
: Upper part.

MSKsimreforme

Problem reformulation.

MSK_SIM_REFORMULATION_ON (1)
: Allow the simplex optimizer to reformulate the problem.

MSK_SIM_REFORMULATION_OFF (0)
: Disallow the simplex optimizer to reformulate the problem.

MSK_SIM_REFORMULATION_FREE (2)
: The simplex optimizer can choose freely.

MSK_SIM_REFORMULATION_AGGRESSIVE (3)
: The simplex optimizer should use an aggressive reformulation strategy.

MSKsimdupvece

Exploit duplicate columns.

MSK_SIM_EXPLOIT_DUPVEC_ON (1)
: Allow the simplex optimizer to exploit duplicated columns.

MSK_SIM_EXPLOIT_DUPVEC_OFF (0)
: Disallow the simplex optimizer to exploit duplicated columns.

MSK_SIM_EXPLOIT_DUPVEC_FREE (2)
: The simplex optimizer can choose freely.

MSKsimhotstarte

Hot-start type employed by the simplex optimizer

MSK_SIM_HOTSTART_NONE (0)
: The simplex optimizer performs a coldstart.

MSK_SIM_HOTSTART_FREE (1)
: The simplex optimize chooses the hot-start type.

MSK_SIM_HOTSTART_STATUS_KEYS (2)
: Only the status keys of the constraints and variables are used to choose the type of hot-start.

MSKintpnthotstarte

Hot-start type employed by the interior-point optimizers.

MSK_INTPNT_HOTSTART_NONE (0)
: The interior-point optimizer performs a coldstart.

MSK_INTPNT_HOTSTART_PRIMAL (1)
: The interior-point optimizer exploits the primal solution only.

MSK_INTPNT_HOTSTART_DUAL (2)
: The interior-point optimizer exploits the dual solution only.

MSK_INTPNT_HOTSTART_PRIMAL_DUAL (3)
: The interior-point optimizer exploits both the primal and dual solution.

MSKpurifye

Solution purification employed optimizer.

MSK_PURIFY_NONE (0)
: The optimizer performs no solution purification.

MSK_PURIFY_PRIMAL (1)
: The optimizer purifies the primal solution.

MSK_PURIFY_DUAL (2)
: The optimizer purifies the dual solution.

MSK_PURIFY_PRIMAL_DUAL (3)
: The optimizer purifies both the primal and dual solution.

MSK_PURIFY_AUTO (4)
: TBD

MSKcallbackcodee

Progress callback codes

MSK_CALLBACK_BEGIN_BI (0)
: The basis identification procedure has been started.

MSK_CALLBACK_BEGIN_CONIC (1)
: The callback function is called when the conic optimizer is started.

MSK_CALLBACK_BEGIN_DUAL_BI (2)
: The callback function is called from within the basis identification procedure when the dual phase is started.

MSK_CALLBACK_BEGIN_DUAL_SENSITIVITY (3)
: Dual sensitivity analysis is started.

MSK_CALLBACK_BEGIN_DUAL_SETUP_BI (4)
: The callback function is called when the dual BI phase is started.

MSK_CALLBACK_BEGIN_DUAL_SIMPLEX (5)
: The callback function is called when the dual simplex optimizer started.

MSK_CALLBACK_BEGIN_DUAL_SIMPLEX_BI (6)
: The callback function is called from within the basis identification procedure when the dual simplex clean-up phase is started.

MSK_CALLBACK_BEGIN_INFEAS_ANA (7)
: The callback function is called when the infeasibility analyzer is started.

MSK_CALLBACK_BEGIN_INTPNT (8)
: The callback function is called when the interior-point optimizer is started.

MSK_CALLBACK_BEGIN_LICENSE_WAIT (9)
: Begin waiting for license.

MSK_CALLBACK_BEGIN_MIO (10)
: The callback function is called when the mixed-integer optimizer is started.

MSK_CALLBACK_BEGIN_OPTIMIZER (11)
: The callback function is called when the optimizer is started.

MSK_CALLBACK_BEGIN_PRESOLVE (12)
: The callback function is called when the presolve is started.

MSK_CALLBACK_BEGIN_PRIMAL_BI (13)
: The callback function is called from within the basis identification procedure when the primal phase is started.

MSK_CALLBACK_BEGIN_PRIMAL_REPAIR (14)
: Begin primal feasibility repair.

MSK_CALLBACK_BEGIN_PRIMAL_SENSITIVITY (15)
: Primal sensitivity analysis is started.

MSK_CALLBACK_BEGIN_PRIMAL_SETUP_BI (16)
: The callback function is called when the primal BI setup is started.

MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX (17)
: The callback function is called when the primal simplex optimizer is started.

MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX_BI (18)
: 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 (19)
: Begin QCQO reformulation.

MSK_CALLBACK_BEGIN_READ (20)
: **MOSEK** has started reading a problem file.

MSK_CALLBACK_BEGIN_ROOT_CUTGEN (21)
: The callback function is called when root cut generation is started.

MSK_CALLBACK_BEGIN_SIMPLEX (22)
: The callback function is called when the simplex optimizer is started.

MSK_CALLBACK_BEGIN_SIMPLEX_BI (23)
: The callback function is called from within the basis identification procedure when the simplex clean-up phase is started.

MSK_CALLBACK_BEGIN_SOLVE_ROOT_RELAX (24)
: The callback function is called when solution of root relaxation is started.

MSK_CALLBACK_BEGIN_TO_CONIC (25)
: Begin conic reformulation.

MSK_CALLBACK_BEGIN_WRITE (26)
: **MOSEK** has started writing a problem file.

MSK_CALLBACK_CONIC (27)
: The callback function is called from within the conic optimizer after the information database has been updated.

MSK_CALLBACK_DUAL_SIMPLEX (28)
: The callback function is called from within the dual simplex optimizer.

MSK_CALLBACK_END_BI (29)
: The callback function is called when the basis identification procedure is terminated.

`MSK_CALLBACK_END_CONIC (30)`
: The callback function is called when the conic optimizer is terminated.

`MSK_CALLBACK_END_DUAL_BI (31)`
: The callback function is called from within the basis identification procedure when the dual phase is terminated.

`MSK_CALLBACK_END_DUAL_SENSITIVITY (32)`
: Dual sensitivity analysis is terminated.

`MSK_CALLBACK_END_DUAL_SETUP_BI (33)`
: The callback function is called when the dual BI phase is terminated.

`MSK_CALLBACK_END_DUAL_SIMPLEX (34)`
: The callback function is called when the dual simplex optimizer is terminated.

`MSK_CALLBACK_END_DUAL_SIMPLEX_BI (35)`
: The callback function is called from within the basis identification procedure when the dual clean-up phase is terminated.

`MSK_CALLBACK_END_INFEAS_ANA (36)`
: The callback function is called when the infeasibility analyzer is terminated.

`MSK_CALLBACK_END_INTPNT (37)`
: The callback function is called when the interior-point optimizer is terminated.

`MSK_CALLBACK_END_LICENSE_WAIT (38)`
: End waiting for license.

`MSK_CALLBACK_END_MIO (39)`
: The callback function is called when the mixed-integer optimizer is terminated.

`MSK_CALLBACK_END_OPTIMIZER (40)`
: The callback function is called when the optimizer is terminated.

`MSK_CALLBACK_END_PRESOLVE (41)`
: The callback function is called when the presolve is completed.

`MSK_CALLBACK_END_PRIMAL_BI (42)`
: The callback function is called from within the basis identification procedure when the primal phase is terminated.

`MSK_CALLBACK_END_PRIMAL_REPAIR (43)`
: End primal feasibility repair.

`MSK_CALLBACK_END_PRIMAL_SENSITIVITY (44)`
: Primal sensitivity analysis is terminated.

`MSK_CALLBACK_END_PRIMAL_SETUP_BI (45)`
: The callback function is called when the primal BI setup is terminated.

`MSK_CALLBACK_END_PRIMAL_SIMPLEX (46)`
: The callback function is called when the primal simplex optimizer is terminated.

`MSK_CALLBACK_END_PRIMAL_SIMPLEX_BI (47)`
: The callback function is called from within the basis identification procedure when the primal clean-up phase is terminated.

`MSK_CALLBACK_END_QCQO_REFORMULATE (48)`
: End QCQO reformulation.

`MSK_CALLBACK_END_READ (49)`
: **MOSEK** has finished reading a problem file.

`MSK_CALLBACK_END_ROOT_CUTGEN (50)`
: The callback function is called when root cut generation is terminated.

`MSK_CALLBACK_END_SIMPLEX (51)`
: The callback function is called when the simplex optimizer is terminated.

`MSK_CALLBACK_END_SIMPLEX_BI (52)`
: 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 (53)
: The callback function is called when solution of root relaxation is terminated.

MSK_CALLBACK_END_TO_CONIC (54)
: End conic reformulation.

MSK_CALLBACK_END_WRITE (55)
: **MOSEK** has finished writing a problem file.

MSK_CALLBACK_IM_BI (56)
: The callback function is called from within the basis identification procedure at an intermediate point.

MSK_CALLBACK_IM_CONIC (57)
: The callback function is called at an intermediate stage within the conic optimizer where the information database has not been updated.

MSK_CALLBACK_IM_DUAL_BI (58)
: The callback function is called from within the basis identification procedure at an intermediate point in the dual phase.

MSK_CALLBACK_IM_DUAL_SENSIVITY (59)
: The callback function is called at an intermediate stage of the dual sensitivity analysis.

MSK_CALLBACK_IM_DUAL_SIMPLEX (60)
: The callback function is called at an intermediate point in the dual simplex optimizer.

MSK_CALLBACK_IM_INTPNT (61)
: The callback function is called at an intermediate stage within the interior-point optimizer where the information database has not been updated.

MSK_CALLBACK_IM_LICENSE_WAIT (62)
: **MOSEK** is waiting for a license.

MSK_CALLBACK_IM_LU (63)
: The callback function is called from within the LU factorization procedure at an intermediate point.

MSK_CALLBACK_IM_MIO (64)
: The callback function is called at an intermediate point in the mixed-integer optimizer.

MSK_CALLBACK_IM_MIO_DUAL_SIMPLEX (65)
: 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 (66)
: 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 (67)
: The callback function is called at an intermediate point in the mixed-integer optimizer while running the primal simplex optimizer.

MSK_CALLBACK_IM_ORDER (68)
: The callback function is called from within the matrix ordering procedure at an intermediate point.

MSK_CALLBACK_IM_PRESOLVE (69)
: The callback function is called from within the presolve procedure at an intermediate stage.

MSK_CALLBACK_IM_PRIMAL_BI (70)
: The callback function is called from within the basis identification procedure at an intermediate point in the primal phase.

MSK_CALLBACK_IM_PRIMAL_SENSIVITY (71)
: The callback function is called at an intermediate stage of the primal sensitivity analysis.

MSK_CALLBACK_IM_PRIMAL_SIMPLEX (72)
: The callback function is called at an intermediate point in the primal simplex optimizer.

MSK_CALLBACK_IM_QO_REFORMULATE (73)
: The callback function is called at an intermediate stage of the conic quadratic reformulation.

MSK_CALLBACK_IM_READ (74)
: Intermediate stage in reading.

MSK_CALLBACK_IM_ROOT_CUTGEN (75)
: The callback is called from within root cut generation at an intermediate stage.

MSK_CALLBACK_IM_SIMPLEX (76)
: The callback function is called from within the simplex optimizer at an intermediate point.

MSK_CALLBACK_IM_SIMPLEX_BI (77)
: The callback function is called from within the basis identification procedure at an intermediate point in the simplex clean-up phase. The frequency of the callbacks is controlled by the *MSK_IPAR_LOG_SIM_FREQ* parameter.

MSK_CALLBACK_INTPNT (78)
: The callback function is called from within the interior-point optimizer after the information database has been updated.

MSK_CALLBACK_NEW_INT_MIO (79)
: The callback function is called after a new integer solution has been located by the mixed-integer optimizer.

MSK_CALLBACK_PRIMAL_SIMPLEX (80)
: The callback function is called from within the primal simplex optimizer.

MSK_CALLBACK_READ_OPF (81)
: The callback function is called from the OPF reader.

MSK_CALLBACK_READ_OPF_SECTION (82)
: A chunk of Q non-zeros has been read from a problem file.

MSK_CALLBACK_SOLVING_REMOTE (83)
: The callback function is called while the task is being solved on a remote server.

MSK_CALLBACK_UPDATE_DUAL_BI (84)
: The callback function is called from within the basis identification procedure at an intermediate point in the dual phase.

MSK_CALLBACK_UPDATE_DUAL_SIMPLEX (85)
: The callback function is called in the dual simplex optimizer.

MSK_CALLBACK_UPDATE_DUAL_SIMPLEX_BI (86)
: 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 (87)
: The callback function is called from within the presolve procedure.

MSK_CALLBACK_UPDATE_PRIMAL_BI (88)
: The callback function is called from within the basis identification procedure at an intermediate point in the primal phase.

MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX (89)
: The callback function is called in the primal simplex optimizer.

MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX_BI (90)
: 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 (91)
: The callback function is called from simplex optimizer.

MSK_CALLBACK_WRITE_OPF (92)
: The callback function is called from the OPF writer.

MSKcheckconvexitytypee
: Types of convexity checks.

MSK_CHECK_CONVEXITY_NONE (0)
: No convexity check.

`MSK_CHECK_CONVEXITY_SIMPLE (1)`
: Perform simple and fast convexity check.

`MSK_CHECK_CONVEXITY_FULL (2)`
: Perform a full convexity check.

`MSKcompresstypee`
Compression types

`MSK_COMPRESS_NONE (0)`
: No compression is used.

`MSK_COMPRESS_FREE (1)`
: The type of compression used is chosen automatically.

`MSK_COMPRESS_GZIP (2)`
: The type of compression used is gzip compatible.

`MSK_COMPRESS_ZSTD (3)`
: The type of compression used is zstd compatible.

`MSKconetypee`
Cone types

`MSK_CT_QUAD (0)`
: The cone is a quadratic cone.

`MSK_CT_RQUAD (1)`
: The cone is a rotated quadratic cone.

`MSK_CT_PEXP (2)`
: A primal exponential cone.

`MSK_CT_DEXP (3)`
: A dual exponential cone.

`MSK_CT_PPOW (4)`
: A primal power cone.

`MSK_CT_DPOW (5)`
: A dual power cone.

`MSK_CT_ZERO (6)`
: The zero cone.

`MSKdomaintypee`
Cone types

`MSK_DOMAIN_R (0)`
: R.

`MSK_DOMAIN_RZERO (1)`
: The zero vector.

`MSK_DOMAIN_RPLUS (2)`
: The positive orthant.

`MSK_DOMAIN_RMINUS (3)`
: The negative orthant.

`MSK_DOMAIN_QUADRATIC_CONE (4)`
: The quadratic cone.

`MSK_DOMAIN_RQUADRATIC_CONE (5)`
: The rotated quadratic cone.

`MSK_DOMAIN_PRIMAL_EXP_CONE (6)`
: The primal exponential cone.

`MSK_DOMAIN_DUAL_EXP_CONE (7)`
: The dual exponential cone.

`MSK_DOMAIN_PRIMAL_POWER_CONE (8)`
: The primal power cone.

`MSK_DOMAIN_DUAL_POWER_CONE (9)`
: The dual power cone.

`MSK_DOMAIN_PRIMAL_GEO_MEAN_CONE (10)`
: The primal geometric mean cone.

`MSK_DOMAIN_DUAL_GEO_MEAN_CONE (11)`
: The dual geometric mean cone.

`MSK_DOMAIN_SVEC_PSD_CONE (12)`
: The vectorized positive semidefinite cone.

`MSKnametypee`

Name types

`MSK_NAME_TYPE_GEN (0)`
: General names. However, no duplicate and blank names are allowed.

`MSK_NAME_TYPE_MPS (1)`
: MPS type names.

`MSK_NAME_TYPE_LP (2)`
: LP type names.

`MSKsymmattypee`

Cone types

`MSK_SYMMAT_TYPE_SPARSE (0)`
: Sparse symmetric matrix.

`MSKdataformate`

Data format types

`MSK_DATA_FORMAT_EXTENSION (0)`
: The file extension is used to determine the data file format.

`MSK_DATA_FORMAT_MPS (1)`
: The data file is MPS formatted.

`MSK_DATA_FORMAT_LP (2)`
: The data file is LP formatted.

`MSK_DATA_FORMAT_OP (3)`
: The data file is an optimization problem formatted file.

`MSK_DATA_FORMAT_FREE_MPS (4)`
: The data a free MPS formatted file.

`MSK_DATA_FORMAT_TASK (5)`
: Generic task dump file.

`MSK_DATA_FORMAT_PTF (6)`
: (P)retty (T)ext (F)format.

`MSK_DATA_FORMAT_CB (7)`
: Conic benchmark format,

`MSK_DATA_FORMAT_JSON_TASK (8)`
: JSON based task format.

`MSKsolformate`

Data format types

`MSK_SOL_FORMAT_EXTENSION (0)`
: The file extension is used to determine the data file format.

`MSK_SOL_FORMAT_B (1)`
: Simple binary format

`MSK_SOL_FORMAT_TASK (2)`
: Tar based format.

`MSK_SOL_FORMAT_JSON_TASK (3)`
: JSON based format.

`MSKdinfiteme`

Double information items

`MSK_DINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_DENSITY (0)`
: Density percentage of the scalarized constraint matrix.

`MSK_DINF_BI_CLEAN_DUAL_TIME (1)`
: Time spent within the dual clean-up optimizer of the basis identification procedure since its invocation.

`MSK_DINF_BI_CLEAN_PRIMAL_TIME (2)`
: Time spent within the primal clean-up optimizer of the basis identification procedure since its invocation.

`MSK_DINF_BI_CLEAN_TIME (3)`
: Time spent within the clean-up phase of the basis identification procedure since its invocation.

`MSK_DINF_BI_DUAL_TIME (4)`
: Time spent within the dual phase basis identification procedure since its invocation.

`MSK_DINF_BI_PRIMAL_TIME (5)`
: Time spent within the primal phase of the basis identification procedure since its invocation.

`MSK_DINF_BI_TIME (6)`
: Time spent within the basis identification procedure since its invocation.

`MSK_DINF_INTPNT_DUAL_FEAS (7)`
: 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 (8)`
: Dual objective value reported by the interior-point optimizer.

`MSK_DINF_INTPNT_FACTOR_NUM_FLOPS (9)`
: An estimate of the number of flops used in the factorization.

`MSK_DINF_INTPNT_OPT_STATUS (10)`
: 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 (11)`
: Order time (in seconds).

`MSK_DINF_INTPNT_PRIMAL_FEAS (12)`
: 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 (13)`
: Primal objective value reported by the interior-point optimizer.

`MSK_DINF_INTPNT_TIME (14)`
: Time spent within the interior-point optimizer since its invocation.

`MSK_DINF_MIO_CLIQUE_SEPARATION_TIME (15)`
: Separation time for clique cuts.

`MSK_DINF_MIO_CMIR_SEPARATION_TIME (16)`
: Separation time for CMIR cuts.

`MSK_DINF_MIO_CONSTRUCT_SOLUTION_OBJ (17)`
: 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 (18)`
: Value of the dual bound after presolve but before cut generation.

`MSK_DINF_MIO_GMI_SEPARATION_TIME (19)`
: Separation time for GMI cuts.

`MSK_DINF_MIO_IMPLIED_BOUND_TIME (20)`
: Separation time for implied bound cuts.

MSK_DINF_MIO_INITIAL_FEASIBLE_SOLUTION_OBJ (21)
: If the user provided solution was found to be feasible this information item contains it's objective value.

MSK_DINF_MIO_KNAPSACK_COVER_SEPARATION_TIME (22)
: Separation time for knapsack cover.

MSK_DINF_MIO_LIPRO_SEPARATION_TIME (23)
: Separation time for lift-and-project cuts.

MSK_DINF_MIO_OBJ_ABS_GAP (24)
: 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 (25)
: 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 (26)
: 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 (27)
: 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 (28)
: Total time for probing.

MSK_DINF_MIO_ROOT_CUTGEN_TIME (29)
: Total time for cut generation.

MSK_DINF_MIO_ROOT_OPTIMIZER_TIME (30)
: Time spent in the contiuous optimizer while processing the root node relaxation.

MSK_DINF_MIO_ROOT_PRESOLVE_TIME (31)
: Time spent presolving the problem at the root node.

MSK_DINF_MIO_ROOT_TIME (32)
: Time spent processing the root node.

MSK_DINF_MIO_TIME (33)
: Time spent in the mixed-integer optimizer.

MSK_DINF_MIO_USER_OBJ_CUT (34)
: If the objective cut is used, then this information item has the value of the cut.

MSK_DINF_OPTIMIZER_TIME (35)
: Total time spent in the optimizer since it was invoked.

MSK_DINF_PRESOLVE_ELI_TIME (36)
: Total time spent in the eliminator since the presolve was invoked.

MSK_DINF_PRESOLVE_LINDEP_TIME (37)
: Total time spent in the linear dependency checker since the presolve was invoked.

MSK_DINF_PRESOLVE_TIME (38)
: Total time (in seconds) spent in the presolve since it was invoked.

MSK_DINF_PRESOLVE_TOTAL_PRIMAL_PERTURBATION (39)
: Total perturbation of the bounds of the primal problem.

MSK_DINF_PRIMAL_REPAIR_PENALTY_OBJ (40)
: The optimal objective value of the penalty function.

MSK_DINF_QCQO_REFORMULATE_MAX_PERTURBATION (41)
: Maximum absolute diagonal perturbation occurring during the QCQO reformulation.

MSK_DINF_QCQO_REFORMULATE_TIME (42)
: Time spent with conic quadratic reformulation.

MSK_DINF_QCQO_REFORMULATE_WORST_CHOLESKY_COLUMN_SCALING (43)
: Worst Cholesky column scaling.

MSK_DINF_QCQO_REFORMULATE_WORST_CHOLESKY_DIAG_SCALING (44)
: Worst Cholesky diagonal scaling.

MSK_DINF_READ_DATA_TIME (45)
: Time spent reading the data file.

MSK_DINF_REMOTE_TIME (46)
: The total real time in seconds spent when optimizing on a server by the process performing the optimization on the server

MSK_DINF_SIM_DUAL_TIME (47)
: Time spent in the dual simplex optimizer since invoking it.

MSK_DINF_SIM_FEAS (48)
: Feasibility measure reported by the simplex optimizer.

MSK_DINF_SIM_OBJ (49)
: Objective value reported by the simplex optimizer.

MSK_DINF_SIM_PRIMAL_TIME (50)
: Time spent in the primal simplex optimizer since invoking it.

MSK_DINF_SIM_TIME (51)
: Time spent in the simplex optimizer since invoking it.

MSK_DINF_SOL_BAS_DUAL_OBJ (52)
: Dual objective value of the basic solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *MSK_updatesolutioninfo* .

MSK_DINF_SOL_BAS_DVIOLCON (53)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_BAS_DVIOLVAR (54)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_BAS_NRM_BARX (55)
: Infinity norm of $\overline{X}$ in the basic solution.

MSK_DINF_SOL_BAS_NRM_SLC (56)
: Infinity norm of s_l^c in the basic solution.

MSK_DINF_SOL_BAS_NRM_SLX (57)
: Infinity norm of s_l^x in the basic solution.

MSK_DINF_SOL_BAS_NRM_SUC (58)
: Infinity norm of s_u^c in the basic solution.

MSK_DINF_SOL_BAS_NRM_SUX (59)
: Infinity norm of s_u^X in the basic solution.

MSK_DINF_SOL_BAS_NRM_XC (60)
: Infinity norm of x^c in the basic solution.

MSK_DINF_SOL_BAS_NRM_XX (61)
: Infinity norm of x^x in the basic solution.

MSK_DINF_SOL_BAS_NRM_Y (62)
: Infinity norm of y in the basic solution.

MSK_DINF_SOL_BAS_PRIMAL_OBJ (63)
: Primal objective value of the basic solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *MSK_updatesolutioninfo* .

MSK_DINF_SOL_BAS_PVIOLCON (64)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_BAS_PVIOLVAR (65)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITG_NRM_BARX (66)
: Infinity norm of $\overline{X}$ in the integer solution.

MSK_DINF_SOL_ITG_NRM_XC (67)
: Infinity norm of x^c in the integer solution.

MSK_DINF_SOL_ITG_NRM_XX (68)
: Infinity norm of x^x in the integer solution.

MSK_DINF_SOL_ITG_PRIMAL_OBJ (69)
: Primal objective value of the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITG_PVIOLACC (70)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITG_PVIOLBARVAR (71)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITG_PVIOLCON (72)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITG_PVIOLCONES (73)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITG_PVIOLDJC (74)
: Maximal primal violation for disjunctive constraints in the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITG_PVIOLITG (75)
: Maximal violation for the integer constraints in the integer solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITG_PVIOLVAR (76)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITR_DUAL_OBJ (77)
: Dual objective value of the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITR_DVIOLACC (78)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITR_DVIOLBARVAR (79)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITR_DVIOLCON (80)
: 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 *MSK_updatesolutioninfo* .

MSK_DINF_SOL_ITR_DVIOLCONES (81)
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 *MSK_updatesolutioninfo*.

MSK_DINF_SOL_ITR_DVIOLVAR (82)
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 *MSK_updatesolutioninfo*.

MSK_DINF_SOL_ITR_NRM_BARS (83)
Infinity norm of $\overline{S}$ in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_BARX (84)
Infinity norm of $\overline{X}$ in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SLC (85)
Infinity norm of s_l^c in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SLX (86)
Infinity norm of s_l^x in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SNX (87)
Infinity norm of s_n^x in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SUC (88)
Infinity norm of s_u^c in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SUX (89)
Infinity norm of s_u^X in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_XC (90)
Infinity norm of x^c in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_XX (91)
Infinity norm of x^x in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_Y (92)
Infinity norm of y in the interior-point solution.

MSK_DINF_SOL_ITR_PRIMAL_OBJ (93)
Primal objective value of the interior-point solution. Updated if *MSK_IPAR_AUTO_UPDATE_SOL_INFO* is set or by the method *MSK_updatesolutioninfo*.

MSK_DINF_SOL_ITR_PVIOLACC (94)
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 *MSK_updatesolutioninfo*.

MSK_DINF_SOL_ITR_PVIOLBARVAR (95)
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 *MSK_updatesolutioninfo*.

MSK_DINF_SOL_ITR_PVIOLCON (96)
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 *MSK_updatesolutioninfo*.

MSK_DINF_SOL_ITR_PVIOLCONES (97)
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 *MSK_updatesolutioninfo*.

MSK_DINF_SOL_ITR_PVIOLVAR (98)
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 *MSK_updatesolutioninfo*.

MSK_DINF_TO_CONIC_TIME (99)
Time spent in the last to conic reformulation.

MSK_DINF_WRITE_DATA_TIME (100)
Time spent writing the data file.

MSKfeaturee
License feature

MSK_FEATURE_PTS (0)
Base system.

`MSK_FEATURE_PTON (1)`
: Conic extension.

`MSKliinfiteme`

Long integer information items.

`MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_COLUMNS (0)`
: Number of columns in the scalarized constraint matrix.

`MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_NZ (1)`
: Number of non-zero entries in the scalarized constraint matrix.

`MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_ROWS (2)`
: Number of rows in the scalarized constraint matrix.

`MSK_LIINF_BI_CLEAN_DUAL_DEG_ITER (3)`
: Number of dual degenerate clean iterations performed in the basis identification.

`MSK_LIINF_BI_CLEAN_DUAL_ITER (4)`
: Number of dual clean iterations performed in the basis identification.

`MSK_LIINF_BI_CLEAN_PRIMAL_DEG_ITER (5)`
: Number of primal degenerate clean iterations performed in the basis identification.

`MSK_LIINF_BI_CLEAN_PRIMAL_ITER (6)`
: Number of primal clean iterations performed in the basis identification.

`MSK_LIINF_BI_DUAL_ITER (7)`
: Number of dual pivots performed in the basis identification.

`MSK_LIINF_BI_PRIMAL_ITER (8)`
: Number of primal pivots performed in the basis identification.

`MSK_LIINF_INTPNT_FACTOR_NUM_NZ (9)`
: Number of non-zeros in factorization.

`MSK_LIINF_MIO_ANZ (10)`
: Number of non-zero entries in the constraint matrix of the problem to be solved by the mixed-integer optimizer.

`MSK_LIINF_MIO_INTPNT_ITER (11)`
: Number of interior-point iterations performed by the mixed-integer optimizer.

`MSK_LIINF_MIO_NUM_DUAL_ILLPOSED_CER (12)`
: Number of dual illposed certificates encountered by the mixed-integer optimizer.

`MSK_LIINF_MIO_NUM_PRIM_ILLPOSED_CER (13)`
: Number of primal illposed certificates encountered by the mixed-integer optimizer.

`MSK_LIINF_MIO_PRESOLVED_ANZ (14)`
: Number of non-zero entries in the constraint matrix of the problem after the mixed-integer optimizer's presolve.

`MSK_LIINF_MIO_SIMPLEX_ITER (15)`
: Number of simplex iterations performed by the mixed-integer optimizer.

`MSK_LIINF_RD_NUMACC (16)`
: Number of affince conic constraints.

`MSK_LIINF_RD_NUMANZ (17)`
: Number of non-zeros in A that is read.

`MSK_LIINF_RD_NUMDJC (18)`
: Number of disjuncive constraints.

`MSK_LIINF_RD_NUMQNZ (19)`
: Number of Q non-zeros.

`MSK_LIINF_SIMPLEX_ITER (20)`
: Number of iterations performed by the simplex optimizer.

`MSKiinfiteme`

Integer information items.

MSK_IINF_ANA_PRO_NUM_CON (0)
: Number of constraints in the problem. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_CON_EQ (1)
: Number of equality constraints. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_CON_FR (2)
: Number of unbounded constraints. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_CON_LO (3)
: Number of constraints with a lower bound and an infinite upper bound. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_CON_RA (4)
: Number of constraints with finite lower and upper bounds. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_CON_UP (5)
: Number of constraints with an upper bound and an infinite lower bound. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR (6)
: Number of variables in the problem. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_BIN (7)
: Number of binary (0-1) variables. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_CONT (8)
: Number of continuous variables. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_EQ (9)
: Number of fixed variables. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_FR (10)
: Number of free variables. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_INT (11)
: Number of general integer variables. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_LO (12)
: Number of variables with a lower bound and an infinite upper bound. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_RA (13)
: Number of variables with finite lower and upper bounds. This value is set by *MSK_analyzeproblem*.

MSK_IINF_ANA_PRO_NUM_VAR_UP (14)
: Number of variables with an upper bound and an infinite lower bound. This value is set by *MSK_analyzeproblem*.

MSK_IINF_INTPNT_FACTOR_DIM_DENSE (15)
: Dimension of the dense sub system in factorization.

MSK_IINF_INTPNT_ITER (16)
: Number of interior-point iterations since invoking the interior-point optimizer.

MSK_IINF_INTPNT_NUM_THREADS (17)
: Number of threads that the interior-point optimizer is using.

MSK_IINF_INTPNT_SOLVE_DUAL (18)
: Non-zero if the interior-point optimizer is solving the dual problem.

MSK_IINF_MIO_ABSGAP_SATISFIED (19)
: Non-zero if absolute gap is within tolerances.

MSK_IINF_MIO_CLIQUE_TABLE_SIZE (20)
: Size of the clique table.

MSK_IINF_MIO_CONSTRUCT_SOLUTION (21)
: 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_INITIAL_FEASIBLE_SOLUTION (22)`
: 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 (23)`
: Depth of the last node solved.

`MSK_IINF_MIO_NUM_ACTIVE_NODES (24)`
: Number of active branch and bound nodes.

`MSK_IINF_MIO_NUM_BRANCH (25)`
: Number of branches performed during the optimization.

`MSK_IINF_MIO_NUM_CLIQUE_CUTS (26)`
: Number of clique cuts.

`MSK_IINF_MIO_NUM_CMIR_CUTS (27)`
: Number of Complemented Mixed Integer Rounding (CMIR) cuts.

`MSK_IINF_MIO_NUM_GOMORY_CUTS (28)`
: Number of Gomory cuts.

`MSK_IINF_MIO_NUM_IMPLIED_BOUND_CUTS (29)`
: Number of implied bound cuts.

`MSK_IINF_MIO_NUM_INT_SOLUTIONS (30)`
: Number of integer feasible solutions that have been found.

`MSK_IINF_MIO_NUM_KNAPSACK_COVER_CUTS (31)`
: Number of clique cuts.

`MSK_IINF_MIO_NUM_LIPRO_CUTS (32)`
: Number of lift-and-project cuts.

`MSK_IINF_MIO_NUM_RELAX (33)`
: Number of relaxations solved during the optimization.

`MSK_IINF_MIO_NUM_REPEATED_PRESOLVE (34)`
: Number of times presolve was repeated at root.

`MSK_IINF_MIO_NUMBIN (35)`
: Number of binary variables in the problem to be solved by the mixed-integer optimizer.

`MSK_IINF_MIO_NUMBINCONEVAR (36)`
: Number of binary cone variables in the problem to be solved by the mixed-integer optimizer.

`MSK_IINF_MIO_NUMCON (37)`
: Number of constraints in the problem to be solved by the mixed-integer optimizer.

`MSK_IINF_MIO_NUMCONE (38)`
: Number of cones in the problem to be solved by the mixed-integer optimizer.

`MSK_IINF_MIO_NUMCONEVAR (39)`
: Number of cone variables in the problem to be solved by the mixed-integer optimizer.

`MSK_IINF_MIO_NUMCONT (40)`
: Number of continuous variables in the problem to be solved by the mixed-integer optimizer.

`MSK_IINF_MIO_NUMCONTCONEVAR (41)`
: Number of continuous cone variables in the problem to be solved by the mixed-integer optimizer.

`MSK_IINF_MIO_NUMDEXPCONES (42)`
: Number of dual exponential cones in the problem to be solved by the mixed-integer optimizer.

`MSK_IINF_MIO_NUMDJC (43)`
: Number of disjunctive constraints in the problem to be solved by the mixed-integer optimizer.

`MSK_IINF_MIO_NUMDPOWCONES (44)`
: Number of dual power cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMINT (45)
: Number of integer variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMINTCONEVAR (46)
: Number of integer cone variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMPEXPCONES (47)
: Number of primal exponential cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMPPOWCONES (48)
: Number of primal power cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMQCONES (49)
: Number of quadratic cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMRQCONES (50)
: Number of rotated quadratic cones in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMVAR (51)
: Number of variables in the problem to be solved by the mixed-integer optimizer.

MSK_IINF_MIO_OBJ_BOUND_DEFINED (52)
: Non-zero if a valid objective bound has been found, otherwise zero.

MSK_IINF_MIO_PRESOLVED_NUMBIN (53)
: Number of binary variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMBINCONEVAR (54)
: Number of binary cone variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMCON (55)
: Number of constraints in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMCONE (56)
: Number of cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMCONEVAR (57)
: Number of cone variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMCONT (58)
: Number of continuous variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMCONTCONEVAR (59)
: Number of continuous cone variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMDEXPCONES (60)
: Number of dual exponential cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMDJC (61)
: Number of disjunctive constraints in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMDPOWCONES (62)
: Number of dual power cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMINT (63)
: Number of integer variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMINTCONEVAR (64)
: Number of integer cone variables in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMPEXPCONES (65)
: Number of primal exponential cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMPPOWCONES (66)
: Number of primal power cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMQCONES (67)
: Number of quadratic cones in the problem after the mixed-integer optimizer's presolve.

MSK_IINF_MIO_PRESOLVED_NUMRQCONES (68)
: Number of rotated quadratic cones in the problem after the mixed-integer optimizer's presolve.

`MSK_IINF_MIO_PRESOLVED_NUMVAR (69)`
: Number of variables in the problem after the mixed-integer optimizer's presolve.

`MSK_IINF_MIO_RELGAP_SATISFIED (70)`
: Non-zero if relative gap is within tolerances.

`MSK_IINF_MIO_TOTAL_NUM_CUTS (71)`
: Total number of cuts generated by the mixed-integer optimizer.

`MSK_IINF_MIO_USER_OBJ_CUT (72)`
: If it is non-zero, then the objective cut is used.

`MSK_IINF_OPT_NUMCON (73)`
: Number of constraints in the problem solved when the optimizer is called.

`MSK_IINF_OPT_NUMVAR (74)`
: Number of variables in the problem solved when the optimizer is called

`MSK_IINF_OPTIMIZE_RESPONSE (75)`
: The response code returned by optimize.

`MSK_IINF_PRESOLVE_NUM_PRIMAL_PERTURBATIONS (76)`
: Number perturbations to thhe bounds of the primal problem.

`MSK_IINF_PURIFY_DUAL_SUCCESS (77)`
: Is nonzero if the dual solution is purified.

`MSK_IINF_PURIFY_PRIMAL_SUCCESS (78)`
: Is nonzero if the primal solution is purified.

`MSK_IINF_RD_NUMBARVAR (79)`
: Number of symmetric variables read.

`MSK_IINF_RD_NUMCON (80)`
: Number of constraints read.

`MSK_IINF_RD_NUMCONE (81)`
: Number of conic constraints read.

`MSK_IINF_RD_NUMINTVAR (82)`
: Number of integer-constrained variables read.

`MSK_IINF_RD_NUMQ (83)`
: Number of nonempty Q matrices read.

`MSK_IINF_RD_NUMVAR (84)`
: Number of variables read.

`MSK_IINF_RD_PROTYPE (85)`
: Problem type.

`MSK_IINF_SIM_DUAL_DEG_ITER (86)`
: The number of dual degenerate iterations.

`MSK_IINF_SIM_DUAL_HOTSTART (87)`
: If 1 then the dual simplex algorithm is solving from an advanced basis.

`MSK_IINF_SIM_DUAL_HOTSTART_LU (88)`
: 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 (89)`
: The number of iterations taken with dual infeasibility.

`MSK_IINF_SIM_DUAL_ITER (90)`
: Number of dual simplex iterations during the last optimization.

`MSK_IINF_SIM_NUMCON (91)`
: Number of constraints in the problem solved by the simplex optimizer.

`MSK_IINF_SIM_NUMVAR (92)`
: Number of variables in the problem solved by the simplex optimizer.

`MSK_IINF_SIM_PRIMAL_DEG_ITER (93)`
: The number of primal degenerate iterations.

`MSK_IINF_SIM_PRIMAL_HOTSTART (94)`
: If 1 then the primal simplex algorithm is solving from an advanced basis.

`MSK_IINF_SIM_PRIMAL_HOTSTART_LU (95)`
: 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 (96)`
: The number of iterations taken with primal infeasibility.

`MSK_IINF_SIM_PRIMAL_ITER (97)`
: Number of primal simplex iterations during the last optimization.

`MSK_IINF_SIM_SOLVE_DUAL (98)`
: Is non-zero if dual problem is solved.

`MSK_IINF_SOL_BAS_PROSTA (99)`
: Problem status of the basic solution. Updated after each optimization.

`MSK_IINF_SOL_BAS_SOLSTA (100)`
: Solution status of the basic solution. Updated after each optimization.

`MSK_IINF_SOL_ITG_PROSTA (101)`
: Problem status of the integer solution. Updated after each optimization.

`MSK_IINF_SOL_ITG_SOLSTA (102)`
: Solution status of the integer solution. Updated after each optimization.

`MSK_IINF_SOL_ITR_PROSTA (103)`
: Problem status of the interior-point solution. Updated after each optimization.

`MSK_IINF_SOL_ITR_SOLSTA (104)`
: Solution status of the interior-point solution. Updated after each optimization.

`MSK_IINF_STO_NUM_A_REALLOC (105)`
: Number of times the storage for storing A has been changed. A large value may indicates that memory fragmentation may occur.

`MSKinftypee`
Information item types

`MSK_INF_DOU_TYPE (0)`
: Is a double information type.

`MSK_INF_INT_TYPE (1)`
: Is an integer.

`MSK_INF_LINT_TYPE (2)`
: Is a long integer.

`MSKiomodee`
Input/output modes

`MSK_IOMODE_READ (0)`
: The file is read-only.

`MSK_IOMODE_WRITE (1)`
: 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 (2)`
: The file is to read and write.

`MSKbranchdire`
Specifies the branching direction.

`MSK_BRANCH_DIR_FREE (0)`
: The mixed-integer optimizer decides which branch to choose.

`MSK_BRANCH_DIR_UP (1)`
: The mixed-integer optimizer always chooses the up branch first.

`MSK_BRANCH_DIR_DOWN (2)`
: The mixed-integer optimizer always chooses the down branch first.

`MSK_BRANCH_DIR_NEAR (3)`
: Branch in direction nearest to selected fractional variable.

`MSK_BRANCH_DIR_FAR (4)`
: Branch in direction farthest from selected fractional variable.

`MSK_BRANCH_DIR_ROOT_LP (5)`
: Chose direction based on root lp value of selected variable.

`MSK_BRANCH_DIR_GUIDED (6)`
: Branch in direction of current incumbent.

`MSK_BRANCH_DIR_PSEUDOCOST (7)`
: Branch based on the pseudocost of the variable.

`MSKmiqcqoreformmethode`

Specifies the reformulation method for mixed-integer quadratic problems.

`MSK_MIO_QCQO_REFORMULATION_METHOD_FREE (0)`
: The mixed-integer optimizer decides which reformulation method to apply.

`MSK_MIO_QCQO_REFORMULATION_METHOD_NONE (1)`
: No reformulation method is applied.

`MSK_MIO_QCQO_REFORMULATION_METHOD_LINEARIZATION (2)`
: A reformulation via linearization is applied.

`MSK_MIO_QCQO_REFORMULATION_METHOD_EIGEN_VAL_METHOD (3)`
: The eigenvalue method is applied.

`MSK_MIO_QCQO_REFORMULATION_METHOD_DIAG_SDP (4)`
: A perturbation of matrix diagonals via the solution of SDPs is applied.

`MSK_MIO_QCQO_REFORMULATION_METHOD_RELAX_SDP (5)`
: A Reformulation based on the solution of an SDP-relaxation of the problem is applied.

`MSKmiodatapermmethode`

Specifies the problem data permutation method for mixed-integer problems.

`MSK_MIO_DATA_PERMUTATION_METHOD_NONE (0)`
: No problem data permutation is applied.

`MSK_MIO_DATA_PERMUTATION_METHOD_CYCLIC_SHIFT (1)`
: A random cyclic shift is applied to permute the problem data.

`MSK_MIO_DATA_PERMUTATION_METHOD_RANDOM (2)`
: A random permutation is applied to the problem data.

`MSKmiocontsoltypee`

Continuous mixed-integer solution type

`MSK_MIO_CONT_SOL_NONE (0)`
: No interior-point or basic solution are reported when the mixed-integer optimizer is used.

`MSK_MIO_CONT_SOL_ROOT (1)`
: 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 (2)`
: 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 (3)`
: 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.

`MSKmiomodee`

Integer restrictions

`MSK_MIO_MODE_IGNORED (0)`
: The integer constraints are ignored and the problem is solved as a continuous problem.

`MSK_MIO_MODE_SATISFIED (1)`
: Integer restrictions should be satisfied.

`MSKmionodeseltypee`
: Mixed-integer node selection types

`MSK_MIO_NODE_SELECTION_FREE (0)`
: The optimizer decides the node selection strategy.

`MSK_MIO_NODE_SELECTION_FIRST (1)`
: The optimizer employs a depth first node selection strategy.

`MSK_MIO_NODE_SELECTION_BEST (2)`
: The optimizer employs a best bound node selection strategy.

`MSK_MIO_NODE_SELECTION_PSEUDO (3)`
: The optimizer employs selects the node based on a pseudo cost estimate.

`MSKmpsformate`
: MPS file format type

`MSK_MPS_FORMAT_STRICT (0)`
: It is assumed that the input file satisfies the MPS format strictly.

`MSK_MPS_FORMAT_RELAXED (1)`
: It is assumed that the input file satisfies a slightly relaxed version of the MPS format.

`MSK_MPS_FORMAT_FREE (2)`
: 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 (3)`
: The CPLEX compatible version of the MPS format is employed.

`MSKobjsensee`
: Objective sense types

`MSK_OBJECTIVE_SENSE_MINIMIZE (0)`
: The problem should be minimized.

`MSK_OBJECTIVE_SENSE_MAXIMIZE (1)`
: The problem should be maximized.

`MSKonoffkeye`
: On/off

`MSK_ON (1)`
: Switch the option on.

`MSK_OFF (0)`
: Switch the option off.

`MSKoptimizertypee`
: Optimizer types

`MSK_OPTIMIZER_CONIC (0)`
: The optimizer for problems having conic constraints.

`MSK_OPTIMIZER_DUAL_SIMPLEX (1)`
: The dual simplex optimizer is used.

`MSK_OPTIMIZER_FREE (2)`
: The optimizer is chosen automatically.

`MSK_OPTIMIZER_FREE_SIMPLEX (3)`
: One of the simplex optimizers is used.

`MSK_OPTIMIZER_INTPNT (4)`
: The interior-point optimizer is used.

`MSK_OPTIMIZER_MIXED_INT (5)`
: The mixed-integer optimizer.

`MSK_OPTIMIZER_PRIMAL_SIMPLEX (6)`
: The primal simplex optimizer is used.

`MSKorderingtypee`
: Ordering strategies

  `MSK_ORDER_METHOD_FREE (0)`
  : The ordering method is chosen automatically.

  `MSK_ORDER_METHOD_APPMINLOC (1)`
  : Approximate minimum local fill-in ordering is employed.

  `MSK_ORDER_METHOD_EXPERIMENTAL (2)`
  : This option should not be used.

  `MSK_ORDER_METHOD_TRY_GRAPHPAR (3)`
  : Always try the graph partitioning based ordering.

  `MSK_ORDER_METHOD_FORCE_GRAPHPAR (4)`
  : Always use the graph partitioning based ordering even if it is worse than the approximate minimum local fill ordering.

  `MSK_ORDER_METHOD_NONE (5)`
  : No ordering is used.

`MSKpresolvemodee`
: Presolve method.

  `MSK_PRESOLVE_MODE_OFF (0)`
  : The problem is not presolved before it is optimized.

  `MSK_PRESOLVE_MODE_ON (1)`
  : The problem is presolved before it is optimized.

  `MSK_PRESOLVE_MODE_FREE (2)`
  : It is decided automatically whether to presolve before the problem is optimized.

`MSKparametertypee`
: Parameter type

  `MSK_PAR_INVALID_TYPE (0)`
  : Not a valid parameter.

  `MSK_PAR_DOU_TYPE (1)`
  : Is a double parameter.

  `MSK_PAR_INT_TYPE (2)`
  : Is an integer parameter.

  `MSK_PAR_STR_TYPE (3)`
  : Is a string parameter.

`MSKproblemiteme`
: Problem data items

  `MSK_PI_VAR (0)`
  : Item is a variable.

  `MSK_PI_CON (1)`
  : Item is a constraint.

  `MSK_PI_CONE (2)`
  : Item is a cone.

`MSKproblemtypee`
: Problem types

  `MSK_PROBTYPE_LO (0)`
  : The problem is a linear optimization problem.

  `MSK_PROBTYPE_QO (1)`
  : The problem is a quadratic optimization problem.

  `MSK_PROBTYPE_QCQO (2)`
  : The problem is a quadratically constrained optimization problem.

  `MSK_PROBTYPE_CONIC (3)`
  : A conic optimization.

`MSK_PROBTYPE_MIXED (4)`
: General nonlinear constraints and conic constraints. This combination can not be solved by **MOSEK**.

`MSKprostae`
: Problem status keys

`MSK_PRO_STA_UNKNOWN (0)`
: Unknown problem status.

`MSK_PRO_STA_PRIM_AND_DUAL_FEAS (1)`
: The problem is primal and dual feasible.

`MSK_PRO_STA_PRIM_FEAS (2)`
: The problem is primal feasible.

`MSK_PRO_STA_DUAL_FEAS (3)`
: The problem is dual feasible.

`MSK_PRO_STA_PRIM_INFEAS (4)`
: The problem is primal infeasible.

`MSK_PRO_STA_DUAL_INFEAS (5)`
: The problem is dual infeasible.

`MSK_PRO_STA_PRIM_AND_DUAL_INFEAS (6)`
: The problem is primal and dual infeasible.

`MSK_PRO_STA_ILL_POSED (7)`
: 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 (8)`
: The problem is either primal infeasible or unbounded. This may occur for mixed-integer problems.

`MSKxmlwriteroutputtypee`
: XML writer output mode

`MSK_WRITE_XML_MODE_ROW (0)`
: Write in row order.

`MSK_WRITE_XML_MODE_COL (1)`
: Write in column order.

`MSKrescodetypee`
: Response code type

`MSK_RESPONSE_OK (0)`
: The response code is OK.

`MSK_RESPONSE_WRN (1)`
: The response code is a warning.

`MSK_RESPONSE_TRM (2)`
: The response code is an optimizer termination status.

`MSK_RESPONSE_ERR (3)`
: The response code is an error.

`MSK_RESPONSE_UNK (4)`
: The response code does not belong to any class.

`MSKscalingtypee`
: Scaling type

`MSK_SCALING_FREE (0)`
: The optimizer chooses the scaling heuristic.

`MSK_SCALING_NONE (1)`
: No scaling is performed.

`MSKscalingmethode`
: Scaling method

`MSK_SCALING_METHOD_POW2 (0)`
: Scales only with power of 2 leaving the mantissa untouched.

`MSK_SCALING_METHOD_FREE (1)`
: The optimizer chooses the scaling heuristic.

`MSKsensitivitytypee`
Sensitivity types

`MSK_SENSITIVITY_TYPE_BASIS (0)`
: Basis sensitivity analysis is performed.

`MSKsimseltypee`
Simplex selection strategy

`MSK_SIM_SELECTION_FREE (0)`
: The optimizer chooses the pricing strategy.

`MSK_SIM_SELECTION_FULL (1)`
: The optimizer uses full pricing.

`MSK_SIM_SELECTION_ASE (2)`
: The optimizer uses approximate steepest-edge pricing.

`MSK_SIM_SELECTION_DEVEX (3)`
: The optimizer uses devex steepest-edge pricing (or if it is not available an approximate steep-edge selection).

`MSK_SIM_SELECTION_SE (4)`
: The optimizer uses steepest-edge selection (or if it is not available an approximate steep-edge selection).

`MSK_SIM_SELECTION_PARTIAL (5)`
: 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.

`MSKsoliteme`
Solution items

`MSK_SOL_ITEM_XC (0)`
: Solution for the constraints.

`MSK_SOL_ITEM_XX (1)`
: Variable solution.

`MSK_SOL_ITEM_Y (2)`
: Lagrange multipliers for equations.

`MSK_SOL_ITEM_SLC (3)`
: Lagrange multipliers for lower bounds on the constraints.

`MSK_SOL_ITEM_SUC (4)`
: Lagrange multipliers for upper bounds on the constraints.

`MSK_SOL_ITEM_SLX (5)`
: Lagrange multipliers for lower bounds on the variables.

`MSK_SOL_ITEM_SUX (6)`
: Lagrange multipliers for upper bounds on the variables.

`MSK_SOL_ITEM_SNX (7)`
: Lagrange multipliers corresponding to the conic constraints on the variables.

`MSKsolstae`
Solution status keys

`MSK_SOL_STA_UNKNOWN (0)`
: Status of the solution is unknown.

`MSK_SOL_STA_OPTIMAL (1)`
: The solution is optimal.

`MSK_SOL_STA_PRIM_FEAS (2)`
: The solution is primal feasible.

`MSK_SOL_STA_DUAL_FEAS (3)`
: The solution is dual feasible.

`MSK_SOL_STA_PRIM_AND_DUAL_FEAS (4)`
: The solution is both primal and dual feasible.

`MSK_SOL_STA_PRIM_INFEAS_CER (5)`
: The solution is a certificate of primal infeasibility.

`MSK_SOL_STA_DUAL_INFEAS_CER (6)`
: The solution is a certificate of dual infeasibility.

`MSK_SOL_STA_PRIM_ILLPOSED_CER (7)`
: The solution is a certificate that the primal problem is illposed.

`MSK_SOL_STA_DUAL_ILLPOSED_CER (8)`
: The solution is a certificate that the dual problem is illposed.

`MSK_SOL_STA_INTEGER_OPTIMAL (9)`
: The primal solution is integer optimal.

`MSKsoltypee`
Solution types

`MSK_SOL_BAS (1)`
: The basic solution.

`MSK_SOL_ITR (0)`
: The interior solution.

`MSK_SOL_ITG (2)`
: The integer solution.

`MSKsolveforme`
Solve primal or dual form

`MSK_SOLVE_FREE (0)`
: The optimizer is free to solve either the primal or the dual problem.

`MSK_SOLVE_PRIMAL (1)`
: The optimizer should solve the primal problem.

`MSK_SOLVE_DUAL (2)`
: The optimizer should solve the dual problem.

`MSKstakeye`
Status keys

`MSK_SK_UNK (0)`
: The status for the constraint or variable is unknown.

`MSK_SK_BAS (1)`
: The constraint or variable is in the basis.

`MSK_SK_SUPBAS (2)`
: The constraint or variable is super basic.

`MSK_SK_LOW (3)`
: The constraint or variable is at its lower bound.

`MSK_SK_UPR (4)`
: The constraint or variable is at its upper bound.

`MSK_SK_FIX (5)`
: The constraint or variable is fixed.

`MSK_SK_INF (6)`
: The constraint or variable is infeasible in the bounds.

`MSKstartpointtypee`
Starting point types

`MSK_STARTING_POINT_FREE (0)`
: The starting point is chosen automatically.

`MSK_STARTING_POINT_GUESS (1)`
: The optimizer guesses a starting point.

`MSK_STARTING_POINT_CONSTANT (2)`
: The optimizer constructs a starting point by assigning a constant value to all primal and dual variables. This starting point is normally robust.

`MSK_STARTING_POINT_SATISFY_BOUNDS (3)`
: The starting point is chosen to satisfy all the simple bounds on nonlinear variables. If this starting point is employed, then more care than usual should employed when choosing the bounds on the nonlinear variables. In particular very tight bounds should be avoided.

`MSKstreamtypee`
: Stream types

`MSK_STREAM_LOG (0)`
: 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 (1)`
: Message stream. Log information relating to performance and progress of the optimization is written to this stream.

`MSK_STREAM_ERR (2)`
: Error stream. Error messages are written to this stream.

`MSK_STREAM_WRN (3)`
: Warning stream. Warning messages are written to this stream.

`MSKvaluee`
: Integer values

`MSK_MAX_STR_LEN (1024)`
: Maximum string length allowed in **MOSEK**.

`MSK_LICENSE_BUFFER_LENGTH (21)`
: The length of a license key buffer.

`MSKvariabletypee`
: Variable types

`MSK_VAR_TYPE_CONT (0)`
: Is a continuous variable.

`MSK_VAR_TYPE_INT (1)`
: Is an integer variable.

15.8 Data Types

`MSKenv_t`
: The **MOSEK** Environment type.

`MSKtask_t`
: The **MOSEK** Task type.

`MSKuserhandle_t`
: A pointer to a user-defined structure.

`MSKbooleant`
: A signed integer interpreted as a boolean value.

`MSKint32t`
: Signed 32bit integer.

`MSKint64t`
: Signed 64bit integer.

`MSKwchart`
: Wide char type. The actual type may differ depending on the platform; it is either a 16 or 32 bits signed or unsigned integer.

`MSKrealt`
: The floating point type used by **MOSEK**.

`MSKstring_t`
: The string type used by **MOSEK**. This is an UTF-8 encoded zero-terminated char string.

15.9 Function Types

MSKcallbackfunc

```
MSKint32t  (MSKAPI * MSKcallbackfunc) (
  MSKtask_t task,
  MSKuserhandle_t usrptr,
  MSKcallbackcodee caller,
  const MSKrealt * douinf,
  const MSKint32t * intinf,
  const MSKint64t * lintinf)
```

The progress callback function is a user-defined function which will be called by **MOSEK** occasionally during the optimization process. In particular, the callback function is called at the beginning of each iteration in the interior-point optimizer. For the simplex optimizers *MSK_IPAR_LOG_SIM_FREQ* controls how frequently the callback is called. The callback provides an code denoting the point in the solver from which the call happened, and a set of arrays containing information items related to the current state of the solver. Typically the user-defined callback function displays information about the solution process. The callback function can also be used to terminate the optimization process by returning a non-zero value.

The user *must not* call any **MOSEK** function directly or indirectly from the callback function. The only exception is the possibility to retrieve a current best integer solution from the mixed-integer optimizer, see Section *Progress and data callback*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `usrptr` (*MSKuserhandle_t*) – A pointer to a user-defined structure. (input/output)
- `caller` (*MSKcallbackcodee*) – The caller key indicating the current progress of the solver. (input)
- `douinf` (*MSKrealt**) – An array of double information items. The elements correspond to the definitions in *MSKdinfiteme*. (input)
- `intinf` (*MSKint32t**) – An array of integer information items. The elements correspond to the definitions in *MSKiinfiteme*. (input)
- `lintinf` (*MSKint64t**) – An array of long information items. The elements correspond to the definitions in *MSKliinfiteme*. (input)

Return (*MSKint32t*) – If the return value is non-zero, **MOSEK** terminates whatever it is doing and returns control to the calling application.

MSKexitfunc

```
void  (MSKAPI * MSKexitfunc) (
  MSKuserhandle_t usrptr,
  const char * file,
  MSKint32t line,
  const char * msg)
```

A user-defined exit function which is called in case of fatal errors to handle an error message and terminate the program. The function should never return.

Parameters

- `usrptr` (*MSKuserhandle_t*) – A pointer to a user-defined structure. (input/output)
- `file` (`char*`) – The name of the file where the fatal error occurred. (input)
- `line` (*MSKint32t*) – The line number in the file where the fatal error occurred. (input)

- `msg` (`char*`) – A message about the error. (input)

Return (`void`)

`MSKhreadfunc`

```
size_t  (MSKAPI * MSKhreadfunc) (
  MSKuserhandle_t handle,
  void * dest,
  const size_t count)
```

Behaves similarly to system read function. Returns the number of bytes read.

Parameters

- `handle` (*MSKuserhandle_t*) – A pointer to a user-defined data structure (or a null pointer). (input/output)
- `dest` (`void*`) – Read into this destination (output)
- `count` (`size_t`) – Number of bytes to read. (input)

Return (`size_t`) – The function response code.

`MSKhwritefunc`

```
size_t  (MSKAPI * MSKhwritefunc) (
  MSKuserhandle_t handle,
  const void * src,
  const size_t count)
```

Behaves similarly to system write function. Returns the number of bytes written.

Parameters

- `handle` (*MSKuserhandle_t*) – A pointer to a user-defined data structure (or a null pointer). (input/output)
- `src` (`void*`) – Write from this location (input)
- `count` (`size_t`) – Number of bytes to write. (input)

Return (`size_t`) – The function response code.

`MSKresponsefunc`

```
MSKrescodee  (MSKAPI * MSKresponsefunc) (
  MSKuserhandle_t handle,
  MSKrescodee r,
  const char * msg)
```

Whenever **MOSEK** generates a warning or an error this function is called. The argument `r` contains the code of the error/warning and the argument `msg` contains the corresponding error/warning message. This function should always return *MSK_RES_OK*.

Parameters

- `handle` (*MSKuserhandle_t*) – A pointer to a user-defined data structure (or a null pointer). (input/output)
- `r` (*MSKrescodee*) – The response code corresponding to the exception. (input)
- `msg` (`char*`) – A string containing the exception message. (input)

Return (*MSKrescodee*) – The function response code.

`MSKstreamfunc`

```
void  (MSKAPI * MSKstreamfunc) (
  MSKuserhandle_t handle,
  const char * str)
```

The message-stream callback function is a user-defined function which can be linked to any of the **MOSEK** streams. Doing so, the function is called whenever **MOSEK** sends a message to the stream.

The user *must not* call any **MOSEK** function directly or indirectly from the callback function.

Parameters

- `handle` (*`MSKuserhandle_t`*) – A pointer to a user-defined data structure (or a null pointer). (input/output)
- `str` (`char*`) – A string containing a message to a stream. (input)

Return (`void`)

15.10 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).

15.10.1 Linear domains

Each linear domain is determined by the dimension n.

- *`MSK_appendrzerodomain`* : the **zero domain**, consisting of the origin $0^n \in \mathbb{R}^n$.
- *`MSK_appendrplusdomain`* : the **nonnegative orthant domain** $\mathbb{R}^n_{\geq 0}$.
- *`MSK_appendrminusdomain`* : the **nonpositive orthant domain** $\mathbb{R}^n_{\leq 0}$.
- *`MSK_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.

15.10.2 Quadratic cone domains

The quadratic domains are determined by the dimension n.

- *`MSK_appendquadraticconedomain`* : the **quadratic cone domain** is the subset of $\mathbb{R}^n$ defined as

$$\mathcal{Q}^n = \left\{ x \in \mathbb{R}^n \ : \ x_1 \geq \sqrt{x_2^2 + \cdots + x_n^2} \right\}.$$

- *`MSK_appendrquadraticconedomain`* : the **rotated quadratic cone domain** is the subset of $\mathbb{R}^n$ defined as

$$\mathcal{Q}^n_r = \left\{ x \in \mathbb{R}^n \ : \ 2x_1x_2 \geq x_3^2 + \cdots + x_n^2, \ x_1, x_2 \geq 0 \right\}.$$

15.10.3 Exponential cone domains

- *`MSK_appendprimalexpconedomain`* : the **primal exponential cone domain** is the subset of $\mathbb{R}^3$ defined as

$$K_{\exp} = \left\{ (x_1, x_2, x_3) \in \mathbb{R}^3 \ : \ x_1 \geq x_2 \exp(x_3/x_2), \ x_1, x_2 \geq 0 \right\}.$$

- *`MSK_appenddualexpconedomain`* : the **dual exponential cone domain** is the subset of $\mathbb{R}^3$ defined as

$$K^*_{\exp} = \left\{ (x_1, x_2, x_3) \in \mathbb{R}^3 \ : \ x_1 \leq -x_3 \exp(x_2/x_3 - 1), \ x_1 \geq 0, x_3 \leq 0 \right\}.$$

15.10.4 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) $\alpha_1, \ldots, \alpha_{n_l}$.

- *MSK_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.

- *MSK_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.

15.10.5 Geometric mean cone domains

A geometric mean cone domain is determined by the dimension n.

- *MSK_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 $\alpha = (1, \ldots, 1)$.

- *MSK_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 $\alpha = (1, \ldots, 1)$.

15.10.6 Vectorized semidefinite domain

- *MSK_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.

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	
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,
- 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.
- $q^o :\in \mathbb{R}^n \rightarrow \mathbb{R}$ is the quadratic term in the objective where

$$q^o(x) = x^T Q^o x$$

and it is assumed that

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

- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $l^c \in \mathbb{R}^m$ is the lower limit on the activity for the constraints.
- $u^c \in \mathbb{R}^m$ is the upper limit on the activity for the constraints.
- $l^x \in \mathbb{R}^n$ is the lower limit on the activity for the variables.
- $u^x \in \mathbb{R}^n$ is the upper limit on the activity for the variables.
- $q : \mathbb{R}^n \rightarrow \mathbb{R}$ is a vector of quadratic functions. Hence,

$$q_i(x) = x^T Q^i x$$

where it is assumed that

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

- $\mathcal{J} \subseteq \{1, 2, \ldots, n\}$ is an index set of the integer constrained variables.

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. If no name is given, then the objective is named `obj`.

The objective function contains linear and quadratic terms. The linear terms are written as

```
4 x1 + x2 - 0.1 x3
```

and so forth. The quadratic terms are written in square brackets (`[ ]/2`) and are either squared or multiplied as in the examples

```
x1^2
```

and

```
x1 * x2
```

There may be zero or more pairs of brackets containing quadratic expressions.

An example of an objective section is

```
minimize
myobj: 4 x1 + x2 - 0.1 x3 + [ x1^2 + 2.1 x1 * x2 ]/2
```

Please note that the quadratic expressions are multiplied with $\frac{1}{2}$, so that the above expression means

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

If the same variable occurs more than once in the linear part, the coefficients are added, so that `4 x1 + 2 x1` is equivalent to `6 x1`. In the quadratic expressions `x1 * x2` is equivalent to `x2 * x1` and, as in the linear part, if the same variables multiplied or squared occur several times their coefficients are added.

Constraints

The second section beginning with one of the keywords

```
subj to
subject to
s.t.
st
```

defines the linear constraint matrix A and the quadratic matrices Q^i.

A constraint contains a name (optional), expressions adhering to the same rules as in the objective and a bound:

```
subject to
con1: x1 + x2 + [ x3^2 ]/2 <= 5.1
```

The bound type (here <=) may be any of <, <=, =, >, >= (< and <= mean the same), and the bound may be any number.

In the standard LP format it is not possible to define more than one bound per line, but **MOSEK** supports defining ranged constraints by using double-colon (::) instead of a single-colon (:) after the constraint name, i.e.

$$-5 \leq x_1 + x_2 \leq 5 \tag{16.1}$$

may be written as

```
con:: -5 < x_1 + x_2 < 5
```

By default **MOSEK** writes ranged constraints this way.

If the files must adhere to the LP standard, ranged constraints must either be split into upper bounded and lower bounded constraints or be written as an equality with a slack variable. For example the expression (16.1) may be written as

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

Bounds

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

```
bound
bounds
```

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

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

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

Variable Types

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

```
bin
binaries
binary
```

and

```
gen
general
```

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

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

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

Terminating Section

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

```
end
```

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

$$
\begin{array}{rccccl}
\text{maximize/minimize} & & c^T x + q_0(x) & & & \\
l^c & \leq & Ax + q(x) & \leq & u^c, & \\
l^x & \leq & x & \leq & u^x, & \\
 & & x \in \mathcal{K}, & & & \\
 & & x_{\mathcal{J}} \quad \text{integer}, & & &
\end{array} \tag{16.2}
$$

where

- $x \in \mathbb{R}^n$ is the vector of decision variables.
- $A \in \mathbb{R}^{m \times n}$ is the constraint matrix.
- $l^c \in \mathbb{R}^m$ is the lower limit on the activity for the constraints.

- $u^c \in \mathbb{R}^m$ is the upper limit on the activity for the constraints.
- $l^x \in \mathbb{R}^n$ is the lower limit on the activity for the variables.
- $u^x \in \mathbb{R}^n$ is the upper limit on the activity for the variables.
- $q : \mathbb{R}^n \rightarrow \mathbb{R}$ is a vector of quadratic functions. Hence,

  $$q_i(x) = \frac{1}{2} x^T Q^i x$$

  where it is assumed that $Q^i = (Q^i)^T$. Please note the explicit $\frac{1}{2}$ in the quadratic term and that Q^i is required to be symmetric. The same applies to q_0.
- $\mathcal{K}$ is a convex cone.
- $\mathcal{J} \subseteq \{1, 2, \ldots, n\}$ is an index set of the integer-constrained variables.
- c is the vector of objective coefficients.

An MPS file with one row and one column can be illustrated like this:

```
*         1         2         3         4         5         6
*23456789012345678901234567890123456789012345678901234567890
NAME          [name]
OBJSENSE
    [objsense]
OBJNAME       [objname]
ROWS
 ?  [cname1]
COLUMNS
    [vname1]  [cname1]  [value1]       [cname2]  [value2]
RHS
    [name]    [cname1]  [value1]       [cname2]  [value2]
RANGES
    [name]    [cname1]  [value1]       [cname2]  [value2]
QSECTION      [cname1]
    [vname1]  [vname2]  [value1]       [vname3]  [value2]
QMATRIX
    [vname1]  [vname2]  [value1]
QUADOBJ
    [vname1]  [vname2]  [value1]
QCMATRIX      [cname1]
    [vname1]  [vname2]  [value1]
BOUNDS
 ?? [name]    [vname1]  [value1]
CSECTION      [kname1]  [value1]       [ktype]
    [vname1]
ENDATA
```

Here the names in capitals are keywords of the MPS format and names in brackets are custom defined names or values. A couple of notes on the structure:

- Fields: All items surrounded by brackets appear in *fields*. The fields named "valueN" are numerical values. Hence, they must have the format

  ```
  [+|-]XXXXXXX.XXXXXX[[e|E][+|-]XXX]
  ```

  where

  ```
  X = [0|1|2|3|4|5|6|7|8|9].
  ```

- Sections: The MPS file consists of several sections where the names in capitals indicate the beginning of a new section. For example, `COLUMNS` denotes the beginning of the columns section.

- Comments: Lines starting with an `*` are comment lines and are ignored by **MOSEK**.
- Keys: The question marks represent keys to be specified later.
- Extensions: The sections `QSECTION` and `CSECTION` are specific **MOSEK** extensions of the MPS format. The sections `QMATRIX`, `QUADOBJ` and `QCMATRIX` are included for sake of compatibility with other vendors extensions to the MPS format.
- The standard MPS format is a fixed format, i.e. everything in the MPS file must be within certain fixed positions. **MOSEK** also supports a *free format*. See Sec. 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

$$\begin{array}{lccc} \text{minimize} & -x_2 + \frac{1}{2}(2x_1^2 - 2x_1x_3 + 0.2x_2^2 + 2x_3^2) & & \\ \text{subject to} & x_1 + x_2 + x_3 & \geq & 1, \\ & x \geq 0 & & \end{array}$$

has the following MPS file representation

```
* File: qo1.mps
NAME          qo1
ROWS
 N  obj
 G  c1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
QSECTION      obj
    x1        x1        2.0
    x1        x3        -1.0
    x2        x2        0.2
    x3        x3        2.0
ENDATA
```

Regarding the `QSECTION`s please note that:

- Only one `QSECTION` is allowed for each constraint.

- The 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

$$\begin{array}{lccc} \text{minimize} & -x_2 + \frac{1}{2}(2x_1^2 - 2x_1x_3 + 0.2x_2^2 + 2x_3^2) & & \\ \text{subject to} & x_1 + x_2 + x_3 & \geq & 1, \\ & x \geq 0 & & \end{array}$$

has the following MPS file representation using QMATRIX

```
* File: qo1_matrix.mps
NAME          qo1_qmatrix
ROWS
 N  obj
 G  c1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
QMATRIX
    x1        x1        2.0
    x1        x3        -1.0
    x3        x1        -1.0
    x2        x2        0.2
    x3        x3        2.0
ENDATA
```

or the following using QUADOBJ

```
* File: qo1_quadobj.mps
NAME          qo1_quadobj
ROWS
 N  obj
 G  c1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
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

$$\begin{array}{lrcl} \text{minimize} & x_2 & & \\ \text{subject to} & x_1+x_2+x_3 & \geq & 1, \\ & \frac{1}{2}(-2x_1x_3+0.2x_2^2+2x_3^2) & \leq & 10, \\ & x\geq 0 & & \end{array}$$

has the following MPS file representation

```
* File: qo1.mps
NAME          qo1
ROWS
 N  obj
 G  c1
 L  q1
COLUMNS
```

(continues on next page)

(continued from previous page)

```
    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^x_j	u^x_j	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 (16.2). It is assumed that $\mathcal{K}$ satisfies the following requirements. Let

$$x^t \in \mathbb{R}^{n^t}, \quad t = 1, \ldots, k$$

be vectors comprised of parts of the decision variables x so that each decision variable is a member of exactly **one** vector x^t, for example

$$x^1 = \begin{bmatrix} x_1 \\ x_4 \\ x_7 \end{bmatrix} \quad \text{and} \quad x^2 = \begin{bmatrix} x_6 \\ x_5 \\ x_3 \\ x_2 \end{bmatrix}.$$

Next define

$$\mathcal{K} := \left\{ x \in \mathbb{R}^n : \quad x^t \in \mathcal{K}_t, \quad t = 1, \ldots, k \right\}$$

where $\mathcal{K}_t$ must have one of the following forms:

- $\mathbb{R}$ set:

$$\mathcal{K}_t = \mathbb{R}^{n^t}.$$

- Zero cone:

$$\mathcal{K}_t = \{0\} \subseteq \mathbb{R}^{n^t}. \tag{16.3}$$

- Quadratic cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : x_1 \geq \sqrt{\sum_{j=2}^{n^t} x_j^2} \right\}. \tag{16.4}$$

- Rotated quadratic cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : 2x_1x_2 \geq \sum_{j=3}^{n^t} x_j^2, \quad x_1, x_2 \geq 0 \right\}. \tag{16.5}$$

- Primal exponential cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^3 : x_1 \geq x_2 \exp(x_3/x_2), \quad x_1, x_2 \geq 0 \right\}. \tag{16.6}$$

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

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : x_1^{\alpha} x_2^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n^t} x_j^2}, \quad x_1, x_2 \geq 0 \right\}. \tag{16.7}$$

- Dual exponential cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^3 : x_1 \geq -x_3 e^{-1} \exp(x_2/x_3), \quad x_3 \leq 0, x_1 \geq 0 \right\}. \tag{16.8}$$

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

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : \left(\frac{x_1}{\alpha}\right)^{\alpha} \left(\frac{x_2}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n^t} x_j^2}, \quad x_1, x_2 \geq 0 \right\}. \tag{16.9}$$

In general, membership in the $\mathbb{R}$ set is not specified. If a variable is not a member of any other cone then it is assumed to be a member of the $\mathbb{R}$ cone.

Next, let us study an example. Assume that the power cone

$$x_4^{1/3} x_5^{2/3} \geq |x_8|$$

and the rotated quadratic cone

$$2x_3x_7 \geq x_1^2 + x_0^2, \quad x_3, x_7 \geq 0,$$

should be specified in the MPS file. One `CSECTION` is required for each cone and they are specified as follows:

```
*        1         2         3         4         5         6
*23456789012345678901234567890123456789012345678901234567890
CSECTION      konea     3e-1           PPOW
x4
x5
x8
CSECTION      koneb     0.0            RQUAD
x7
x3
x1
x0
```

In general, a `CSECTION` header has the format

```
CSECTION      [kname1]    [value1]     [ktype]
```

where the requirements for each field are as follows:

Field	Starting Position	Max Width	Required	Description
`[kname1]`	15	8	Yes	Name of the cone
`[value1]`	25	12	No	Cone parameter
`[ktype]`	40		Yes	Type of the cone.

The possible cone type keys are:

`[ktype]`	Members	`[value1]`	Interpretation.
`ZERO`	≥ 0	unused	Zero cone (16.3).
`QUAD`	≥ 1	unused	Quadratic cone (16.4).
`RQUAD`	≥ 2	unused	Rotated quadratic cone (16.5).
`PEXP`	3	unused	Primal exponential cone (16.6).
`PPOW`	≥ 2	α	Primal power cone (16.7).
`DEXP`	3	unused	Dual exponential cone (16.8).
`DPOW`	≥ 2	α	Dual power cone (16.9).

A record in the `CSECTION` has the format

```
[vname1]
```

where the requirements for each field are

Field	Starting Position	Max Width	required	Description
`[vname1]`	5	8	Yes	A valid variable name

A variable must occur in at most one `CSECTION`.

ENDATA

This keyword denotes the end of the MPS file.

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

$$2x_1x_2 \geq \sum_{i=3}^{n} x_i^2, \quad x_1, x_2 \geq 0.$$

- `pexp`: primal exponential cone of 3 variables x_1, x_2, x_3 defines a constraint of the form

$$x_1 \geq x_2 \exp(x_3/x_2), \quad x_1, x_2 \geq 0.$$

- `ppow` with parameter $0 < \alpha < 1$: primal power cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1^{\alpha} x_2^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n} x_j^2}, \quad x_1, x_2 \geq 0.$$

- `dexp`: dual exponential cone of 3 variables x_1, x_2, x_3 defines a constraint of the form

$$x_1 \geq -x_3 e^{-1} \exp(x_2/x_3), \quad x_3 \leq 0, x_1 \geq 0.$$

- `dpow` with parameter $0 < \alpha < 1$: dual power cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$\left(\frac{x_1}{\alpha}\right)^{\alpha} \left(\frac{x_2}{1-\alpha}\right)^{1-\alpha} \geq \sqrt{\sum_{j=3}^{n^t} x_j^2}, \quad x_1, x_2 \geq 0.$$

- `zero`: zero cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1 = \cdots = x_n = 0$$

A `[bounds]`-section example:

```
[bounds]
[b]  0 <= x,y <= 10  [/b] # ranged bound
[b] 10 >= x,y >=  0  [/b] # ranged bound
[b]  0 <= x,y <= inf [/b] # using inf
[b]       x,y free   [/b] # free variables
# Let (x,y,z,w) belong to the cone K
[cone rquad] x,y,z,w  [/cone] # rotated quadratic cone
[cone ppow '3e-01' 'a'] x1, x2, x3 [/cone] # power cone with alpha=1/3 and name 'a'
[/bounds]
```

By default all variables are free.

[variables]

This defines an ordering of variables as they should appear in the problem. This is simply a space-separated list of variable names.

[integer]

This contains a space-separated list of variables and defines the constraint that the listed variables must be integer-valued.

[hints]

This may contain only non-essential data; for example estimates of the number of variables, constraints and non-zeros. Placed before all other sections containing data this may reduce the time spent reading the file.

In the `hints` section, any subsection which is not recognized by **MOSEK** is simply ignored. In this section a hint is defined as follows:

```
[hint ITEM] value [/hint]
```

The hints recognized by **MOSEK** are:

- `numvar` (number of variables),
- `numcon` (number of linear/quadratic constraints),
- `numanz` (number of linear non-zeros in constraints),
- `numqnz` (number of quadratic non-zeros in constraints).

[solutions]

This section can contain a set of full or partial solutions to a problem. Each solution must be specified using a `[solution]`-section, i.e.

```
[solutions]
[solution]...[/solution] #solution 1
[solution]...[/solution] #solution 2
#other solutions....
[solution]...[/solution] #solution n
[/solutions]
```

The syntax of a `[solution]`-section is the following:

```
[solution SOLTYPE status=STATUS]...[/solution]
```

where `SOLTYPE` is one of the strings

- `interior`, a non-basic solution,
- `basic`, a basic solution,

- `integer`, an integer solution,

and `STATUS` is one of the strings

- `UNKNOWN`,
- `OPTIMAL`,
- `INTEGER_OPTIMAL`,
- `PRIM_FEAS`,
- `DUAL_FEAS`,
- `PRIM_AND_DUAL_FEAS`,
- `NEAR_OPTIMAL`,
- `NEAR_PRIM_FEAS`,
- `NEAR_DUAL_FEAS`,
- `NEAR_PRIM_AND_DUAL_FEAS`,
- `PRIM_INFEAS_CER`,
- `DUAL_INFEAS_CER`,
- `NEAR_PRIM_INFEAS_CER`,
- `NEAR_DUAL_INFEAS_CER`,
- `NEAR_INTEGER_OPTIMAL`.

Most of these values are irrelevant for input solutions; when constructing a solution for simplex hot-start or an initial solution for a mixed integer problem the safe setting is `UNKNOWN`.

A `[solution]`-section contains `[con]` and `[var]` sections. Each `[con]` and `[var]` section defines solution information for a single variable or constraint, specified as list of `KEYWORD`/`value` pairs, in any order, written as

```
KEYWORD=value
```

Allowed keywords are as follows:

- `sk`. The status of the item, where the `value` is one of the following strings:
  - `LOW`, the item is on its lower bound.
  - `UPR`, the item is on its upper bound.
  - `FIX`, it is a fixed item.
  - `BAS`, the item is in the basis.
  - `SUPBAS`, the item is super basic.
  - `UNK`, the status is unknown.
  - `INF`, the item is outside its bounds (infeasible).
- `lvl` Defines the level of the item.
- `sl` Defines the level of the dual variable associated with its lower bound.
- `su` Defines the level of the dual variable associated with its upper bound.
- `sn` Defines the level of the variable associated with its cone.
- `y` Defines the level of the corresponding dual variable (for constraints only).

A `[var]` section should always contain the items `sk`, `lvl`, `sl` and `su`. Items `sl` and `su` are not required for `integer` solutions.

A `[con]` section should always contain `sk`, `lvl`, `sl`, `su` and `y`.

An example of a solution section

```
[solution basic status=UNKNOWN]
[var x0] sk=LOW    lvl=5.0      [/var]
[var x1] sk=UPR    lvl=10.0     [/var]
[var x2] sk=SUPBAS lvl=2.0  sl=1.5 su=0.0 [/var]

[con c0] sk=LOW    lvl=3.0 y=0.0 [/con]
[con c0] sk=UPR    lvl=0.0 y=5.0 [/con]
[/solution]
```

- `[vendor]` This contains solver/vendor specific data. It accepts one argument, which is a vendor ID – for **MOSEK** the ID is simply `mosek` – and the section contains the subsection `parameters` defining solver parameters. When reading a vendor section, any unknown vendor can be safely ignored. This is described later.

Comments using the `#` may appear anywhere in the file. Between the `#` and the following line-break any text may be written, including markup characters.

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_SOLUTIONS	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_PARAMETERS	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:

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

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]

[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

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

This can be formulated in `opf` as shown below.

Listing 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]

```

(continues on next page)

(continued from previous page)

```
[objective minimize 'obj']
  # The quadratic terms are often written with a factor of 1/2 as here,
  # but this is not required.

   - x2 + 0.5 ( 2.0 x1 ^ 2 - 2.0 x3 * x1 + 0.2 x2 ^ 2 + 2.0 x3 ^ 2 )
[/objective]

[constraints]
  [con 'c1'] 1.0 <= x1 + x2 + x3 [/con]
[/constraints]

[bounds]
  [b] 0 <= * [/b]
[/bounds]
```

Conic Quadratic Example `cqo1.opf`

Consider the example:

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

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
```

(continues on next page)

(continued from previous page)

```
  [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:

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

This can be implemented in OPF with the file in Listing 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:

$$\begin{array}{lll} \min / \max & g^{obj} & \\ & g_i \in \mathcal{K}_i, & i \in \mathcal{I}, \\ \text{s.t.} & G_i \in \mathcal{K}_i, & i \in \mathcal{I}^{PSD}, \\ & x_j \in \mathcal{K}_j, & j \in \mathcal{J}, \\ & \overline{X}_j \in \mathcal{K}_j, & j \in \mathcal{J}^{PSD}. \end{array} \tag{16.10}$$

- **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,

$$S_1 = \left\{ \begin{pmatrix} t \\ s \\ r \end{pmatrix} \in \mathbb{R}^3,\ t \geq s e^{\frac{r}{s}},\ s \geq 0 \right\}.$$

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

$$\left\{ \begin{pmatrix} p \\ x \end{pmatrix} \in \mathbb{R}_+^k \times \mathbb{R}^1,\ \left(\prod_{j=1}^{k} k p_j \right)^{\frac{1}{k}} \geq |x| \right\},\ \text{for } n = k+1 \geq 2.$$

and, the following parametric cones:

- **Radial power cone** - A cone in the power cone family defined by

$$\left\{ \left(\begin{array}{c} p \\ x \end{array} \right) \in \mathbb{R}_+^k \times \mathbb{R}^{n-k}, \left(\prod_{j=1}^{k} p_j^{\alpha_j} \right)^{\frac{1}{\sigma}} \geq \|x\|_2 \right\}, \text{ for } n \geq k \geq 1.$$

  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

$$\left\{ \left(\begin{array}{c} p \\ x \end{array} \right) \in \mathbb{R}_+^k \times \mathbb{R}^{n-k}, \left(\prod_{j=1}^{k} \left(\frac{\sigma p_j}{\alpha_j} \right)^{\alpha_j} \right)^{\frac{1}{\sigma}} \geq \|x\|_2 \right\}, \text{ for } n \geq k \geq 1.$$

  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 (16.11) , has three variables in a quadratic cone - first one is integer - and an affine expression in domain 0 (equality constraint).

$$\begin{array}{ll} \text{minimize} & 5.1\,x_0 \\ \text{subject to} & 6.2\,x_1 + 7.3\,x_2 - 8.4 \in \{0\} \\ & x \in \mathcal{Q}^3,\ x_0 \in \mathbb{Z}. \end{array} \tag{16.11}$$

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 (16.12), has a semidefinite cone, a quadratic cone over unordered subindices, and two equality constraints.

$$
\begin{array}{lllll}
\text{minimize} & \left\langle \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, X_1 \right\rangle + x_1 & & \\
\text{subject to} & \left\langle \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, X_1 \right\rangle + x_1 & = & 1.0\,, \\
 & \left\langle \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, X_1 \right\rangle + x_0 + x_2 & = & 0.5\,, \\
 & x_1 \geq \sqrt{x_0^2 + x_2^2}\,, & & \\
 & X_1 \succeq \mathbf{0}\,. & &
\end{array}
\tag{16.12}
$$

The equality constraints are easily rewritten to the conic form, $(g_0, g_1) \in \{0\}^2$, by moving constants such that the right-hand-side becomes zero. The quadratic cone does not fit under the `VAR` keyword in this variable permutation. Instead, it takes a scalar constraint $(g_2, g_3, g_4) = (x_1, x_0, x_2) \in \mathcal{Q}^3$, with scalar variables constructed as $(x_0, x_1, x_2) \in \mathbb{R}^3$. Its formulation in the CBF format is reported in the following list

```
# File written using this version of the Conic Benchmark Format:
#     | Version 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.

$$\begin{array}{lll} \text{minimize} & \left\langle \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, X_1 \right\rangle + x_1 + x_2 + 1 & \\ \text{subject to} & \left\langle \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, X_1 \right\rangle - x_1 - x_2 & \geq \; 0.0\,, \\ & x_1 \begin{bmatrix} 0 & 1 \\ 1 & 3 \end{bmatrix} + x_2 \begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \succeq \; \mathbf{0}\,, \\ & X_1 \succeq \mathbf{0}\,. & \end{array} \tag{16.13}$$

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 (16.14), has one equality constraint, one quadratic cone constraint and an exponential cone constraint.

$$\begin{array}{ll} \text{minimize} & x_0 - x_3 \\ \text{subject to} & x_0 + 2\,x_1 - x_2 \in \{0\} \\ & (5.0, x_0, x_1) \in \mathcal{Q}^3 \\ & (x_2, 1.0, x_3) \in EXP. \end{array} \tag{16.14}$$

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 (16.15), has three variables in a power cone with parameter $\alpha_1 = (1, 1)$ and two power cone constraints each with parameter $\alpha_0 = (8, 1)$.

$$\begin{array}{ll} \text{minimize} & x_3 \\ \text{subject to} & (1.0, x_1, x_1 + x_2) \in POW_{\alpha_0} \\ & (1.0, x_2, x_1 + x_2) \in POW_{\alpha_0} \\ & x \in POW_{\alpha_1}. \end{array} \tag{16.15}$$

The nonlinear conic constraints enforce $x_3 \le x_1 x_2$ and $x_1 + x_2 \le \min(x_1^{\frac{1}{9}}, x_2^{\frac{1}{9}})$.

```
# 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 suuporting all of **MOSEK** optimization problems in conic form, possibly with integer variables and disjunctive constraints.

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 )*
TERM: LINEAR_TERM | MATRIX_TERM
```

A linear term

```
LINEAR_TERM: FLOAT? NAME
```

A matrix term

```
MATRIX_TERM: "<" FLOAT? NAME ( [+-] FLOAT? NAME)* ";" NAME ">"
```

Here the right-hand name is the name of a (semidefinite) matrix variable, and the left-hand side is a sum of symmetric matrixes. The actual matrixes are defined in a separate section.

Expressions can span multiple lines by giving subsequent lines a deeper indentation.

For example following two section are equivalent:

```
# Everything on one line:
x1 + x2 + x3 + x4

# Split into multiple lines:
x1
  + x2
  + x3
  + x4
```

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
  'c1' [0;10] x1 + x2 + x3
  [0] x1 + x2 + x3
```

Constraint blocks put the expression either in a subsection or inline. The cone type (domain) is written in the brackets, and **MOSEK** currently supports following types:

- `SOC(N)` Second order cone of dimension `N`
- `RSOC(N)` Rotated second order cone of dimension `N`
- `PSD(N)` Symmetric positive semidefinite cone of dimension `N`. This contains `N*(N+1)/2` elements.
- `PEXP` Primal exponential cone of dimension 3
- `DEXP` Dual exponential cone of dimension 3
- `PPOW(N,P)` Primal power cone of dimension `N` with parameter `P`
- `DPOW(N,P)` Dual power cone of dimension `N` with parameter `P`
- `ZERO(N)` The zero-cone of dimension `N`.

```
"Constraints"
  NAME? "[" DOMAIN "]" EXPR_LIST
```

For example:

```
Constraints
   'K1' [SOC(3)] x1 + x2 ; x2 + x3 ; x3 + x1
   'K2' [RSOC(3)]
      x1 + x2
      x2 + x3
      x3 + x1
```

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 "[" DOMAIN "]" NAMES

   For example, a linear variable
```

```
Variables
   x1 [0;Inf]
```

As with constraints, members of a conic domain can be listed either inline or in a subsection:

```
Variables
   k1 [SOC(3)] x1 ; x2 ; x3
   k2 [RSOC(3)]
      x1
      x2
      x3
```

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
   "Variables"
      # Linear variable status: level, slx, sux
      NAME "[" STATUS "]" FLOAT (FLOAT FLOAT)?
      # Conic variable status: level, snx
      NAME
         "[" STATUS "]" FLOAT FLOAT?
         ...
   "Constraints"
      # Linear variable status: level, slx, sux
      NAME "[" STATUS "]" FLOAT (FLOAT FLOAT)?
      # Conic variable status: level, snx
      NAME
         "[" STATUS "]" FLOAT FLOAT?
         ...
```

Following values for `WHICHSOL` are supported:

- `interior` Interior solution, the result of an interior-point solver.
- `basic` Basic solution, as produced by a simplex solver.
- `integer` Integer solution, the solution to a mixed-integer problem. This does not define a dual solution.

Following values for `PROSTA` are supported:

- `unknown` The problem status is unknown
- `feasible` The problem has been proven feasible
- `infeasible` The problem has been proven infeasible
- `illposed` The problem has been proved to be ill posed
- `infeasible_or_unbounded` The problem is infeasible or unbounded

Following values for `SOLSTA` are supported:

- `unknown` The solution status is unknown
- `feasible` The solution is feasible
- `optimal` The solution is optimal
- `infeas_cert` The solution is a certificate of infeasibility
- `illposed_cert` The solution is a certificate of illposedness

Following values for `STATUS` are supported:

- `unknown` The value is unknown
- `super_basic` The value is super basic

- `at_lower` The value is basic and at its lower bound
- `at_upper` The value is basic and at its upper bound
- `fixed` The value is basic fixed
- `infinite` The value is at infinity

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 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
```

(continues on next page)

(continued from previous page)

```
    x4
    x1 [0;+inf]
    x2 [0;+inf]
    x5
    x6
    x3 [0;+inf]
```

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]'
```

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 *MSK_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 *MSK_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^c_i 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],
         "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],
```

(continues on next page)

(continued from previous page)

```
            "val":[0.035,1,1,1]
         }
      },
      "domains":{
         "type":[["r",0],
                 ["quad",4],
                 ["ppow",3,[0.6666666666666666,0.33333333333333337]]]
      },
      "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],
         "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>]

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
```

(continues on next page)

(continued from previous page)

```
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.
↪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 C:

Table 17.1: List of distributed examples

File	Description
`acc1.c`	A simple problem with one affine conic constraint (ACC)
`acc2.c`	A simple problem with two affine conic constraints (ACC)
`blas_lapack.c`	Demonstrates the **MOSEK** interface to BLAS/LAPACK linear algebra routines
`callback.c`	An example of data/progress callback
`ceo1.c`	A simple conic exponential problem
`concurrent1.cc`	Implementation of a concurrent optimizer for linear and mixed-integer problems
`cqo1.c`	A simple conic quadratic problem
`djc1.c`	A simple problem with disjunctive constraints (DJC)
`errorreporting.c`	Demonstrates how error reporting can be customized
`feasrepairex1.c`	A simple example of how to repair an infeasible problem
`gp1.c`	A simple geometric program (GP) in conic form
`helloworld.c`	A Hello World example
`lo1.c`	A simple linear problem
`lo2.c`	A simple linear problem
`logistic.c`	Implements logistic regression and simple log-sum-exp (CEO)
`mico1.c`	A simple mixed-integer conic problem
`milo1.c`	A simple mixed-integer linear problem
`mioinitsol.c`	A simple mixed-integer linear problem with an initial guess
`opt_server_async.c`	Uses **MOSEK** OptServer to solve an optimization problem asynchronously
`opt_server_sync.c`	Uses **MOSEK** OptServer to solve an optimization problem synchronously
`parallel.c`	Demonstrates parallel optimization using a batch method in MOSEK
`parameters.c`	Shows how to set optimizer parameters and read information items
`pinfeas.c`	Shows how to obtain and analyze a primal infeasibility certificate
`portfolio_1_basic.c`	Portfolio optimization - basic Markowitz model
`portfolio_2_frontier.c`	Portfolio optimization - efficient frontier
`portfolio_3_impact.c`	Portfolio optimization - market impact costs
`portfolio_4_transcost.c`	Portfolio optimization - transaction costs
`portfolio_5_card.c`	Portfolio optimization - cardinality constraints

continues on next page

Table 17.1 – continued from previous page

File	Description
portfolio_6_factor.c	Portfolio optimization - factor model
pow1.c	A simple power cone problem
qcqo1.c	A simple quadratically constrained quadratic problem
qo1.c	A simple quadratic problem
reoptimization.c	Demonstrate how to modify and re-optimize a linear problem
response.c	Demonstrates proper response handling
sdo1.c	A simple semidefinite problem with one matrix variable and a quadratic cone
sdo2.c	A simple semidefinite problem with two matrix variables
sdo_lmi.c	A simple semidefinite problem with an LMI using the SVEC domain.
sensitivity.c	Sensitivity analysis performed on a small linear problem
simple.c	A simple I/O example: read problem from a file, solve and write solutions
solutionquality.c	Demonstrates how to examine the quality of a solution
solvebasis.c	Demonstrates solving a linear system with the basis matrix
solvelinear.c	Demonstrates solving a general linear system
sparsecholesky.c	Shows how to find a Cholesky factorization of a sparse matrix
unicode.c	Demonstrates string conversion to Unicode

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 C in version 10.0 compared to version 9. See the release notes for general changes and new features of the **MOSEK** Optimization Suite.

18.1 Important changes compared to version 9

- **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.
- **Multithreading.** In the interior-point optimizer it is posible to set the number of threads with *MSK_IPAR_NUM_THREADS* before each optimization, and not just once per process. The parameter `MSK_IPAR_INTPNT_MULTI_THREAD` and the function `MSK_setupthreads` are no longer relevant and were removed.
- **Sparse Cholesky.** In the function *MSK_computesparsecholesky* the argument previously indicating whether to use multiple threads or not is now an integer denoting the number of threads to use, with 0 meaning that **MOSEK** makes the choice.
- **Solve with basis.** The function *MSK_solvewithbasis* changed interface: it separates the input and output number of nonzeros.
- **Surplus.** Remover the `surplus` argument from all `MSK_get` functions using it.
- **OptServer.** The arguments used in remote calls from the **MOSEK** API change from `(server, port)` to `(addr, accesstoken)`, where `addr` is the full URL such as `http://server:port` or `https://server:port`. See the documentation of the relevant functions.
- **MIO initial solution.** In order for the mixed-integer solver to utilize a partial integer solution the parameter *MSK_IPAR_MIO_CONSTRUCT_SOL* must be set. See Sec. 6.8.2 for details. In version 9 this action happened by default.
- **Conic constraints.** The interface introduces affine conic constraints (ACC) as a way of writing directly

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

  where $\mathcal{D}$ is a conic domain, without having to introduce a slack variable $Fx + g - y = 0,\ y \in \mathcal{D}$. See Sec. 6.2 for a tutorial. The current interface through variable cones is still supported but deprecated.

18.2 Changes compared to version 9

18.3 Functions compared to version 9

Added

- *MSK_expirylicenses*
- *MSK_optimizebatch*
- *MSK_resetexpirylicenses*
- *MSK_appendacc*
- *MSK_appendaccs*
- *MSK_appendaccseq*
- *MSK_appendaccsseq*
- *MSK_appendafes*
- *MSK_appenddjcs*
- *MSK_appenddualexpconedomain*
- *MSK_appenddualgeomeanconedomain*
- *MSK_appenddualpowerconedomain*
- *MSK_appendprimalexpconedomain*
- *MSK_appendprimalgeomeanconedomain*
- *MSK_appendprimalpowerconedomain*
- *MSK_appendquadraticconedomain*
- *MSK_appendrdomain*
- *MSK_appendrminusdomain*
- *MSK_appendrplusdomain*
- *MSK_appendrquadraticconedomain*
- *MSK_appendrzerodomain*
- *MSK_appendsvecpsdconedomain*
- *MSK_emptyafebarfrow*
- *MSK_emptyafebarfrowlist*
- *MSK_emptyafefcol*
- *MSK_emptyafefcollist*
- *MSK_emptyafefrow*
- *MSK_emptyafefrowlist*
- *MSK_evaluateacc*
- *MSK_evaluateaccs*

- *MSK_generateaccnames*
- *MSK_generatebarvarnames*
- *MSK_generatedjcnames*
- *MSK_getaccafeidxlist*
- *MSK_getaccb*
- *MSK_getaccbarfblocktriplet*
- *MSK_getaccbarfnumblocktriplets*
- *MSK_getaccdomain*
- *MSK_getaccdoty*
- *MSK_getaccdotys*
- *MSK_getaccfnumnz*
- *MSK_getaccftrip*
- *MSK_getaccgvector*
- *MSK_getaccn*
- *MSK_getaccname*
- *MSK_getaccnamelen*
- *MSK_getaccntot*
- *MSK_getaccs*
- *MSK_getafebarfblocktriplet*
- *MSK_getafebarfnumblocktriplets*
- *MSK_getafebarfnumrowentries*
- *MSK_getafebarfrow*
- *MSK_getafebarfrowinfo*
- *MSK_getafefnumnz*
- *MSK_getafefrow*
- *MSK_getafefrownumnz*
- *MSK_getafeftrip*
- *MSK_getafeg*
- *MSK_getafegslice*
- *MSK_getatrip*
- *MSK_getdjcafeidxlist*
- *MSK_getdjcb*
- *MSK_getdjcdomainidxlist*

- *MSK_getdjcname*
- *MSK_getdjcnamelen*
- *MSK_getdjcnumafe*
- *MSK_getdjcnumafetot*
- *MSK_getdjcnumdomain*
- *MSK_getdjcnumdomaintot*
- *MSK_getdjcnumterm*
- *MSK_getdjcnumtermtot*
- *MSK_getdjcs*
- *MSK_getdjctermsizelist*
- *MSK_getdomainn*
- *MSK_getdomainname*
- *MSK_getdomainnamelen*
- *MSK_getdomaintype*
- *MSK_getdviolacc*
- *MSK_getnumacc*
- *MSK_getnumafe*
- *MSK_getnumdjc*
- *MSK_getnumdomain*
- *MSK_getpowerdomainalpha*
- *MSK_getpowerdomaininfo*
- *MSK_getpviolacc*
- *MSK_getpvioldjc*
- *MSK_getsolutioninfonew*
- *MSK_getsolutionnew*
- *MSK_infeasibilityreport*
- *MSK_putacc*
- *MSK_putaccb*
- *MSK_putaccbj*
- *MSK_putaccdoty*
- *MSK_putacclist*
- *MSK_putaccname*
- *MSK_putafebarfblocktriplet*

- *MSK_putafebarfentry*
- *MSK_putafebarfentrylist*
- *MSK_putafebarfrow*
- *MSK_putafefcol*
- *MSK_putafefentry*
- *MSK_putafefentrylist*
- *MSK_putafefrow*
- *MSK_putafefrowlist*
- *MSK_putafeg*
- *MSK_putafeglist*
- *MSK_putafegslice*
- *MSK_putdjc*
- *MSK_putdjcname*
- *MSK_putdjcslice*
- *MSK_putdomainname*
- *MSK_putmaxnumacc*
- *MSK_putmaxnumafe*
- *MSK_putmaxnumdjc*
- *MSK_putmaxnumdomain*
- *MSK_putsolutionnew*
- *MSK_readbsolution*
- *MSK_readdatacb*
- *MSK_readjsonsol*
- *MSK_readsolutionfile*
- *MSK_writebsolution*
- *MSK_writedatahandle*
- *MSK_writesolutionfile*

Removed

- MSK_makeenvalloc
- MSK_setupthreads
- MSK_strdupdbgtask
- MSK_strduptask

18.4 Parameters compared to version 9

Added

- *MSK_DPAR_MIO_DJC_MAX_BIGM*
- *MSK_DPAR_PRESOLVE_TOL_PRIMAL_INFEAS_PERTURBATION*
- *MSK_IPAR_MIO_CONSTRUCT_SOL*
- *MSK_IPAR_MIO_CUT_LIPRO*
- *MSK_IPAR_MIO_DATA_PERMUTATION_METHOD*
- *MSK_IPAR_MIO_MEMORY_EMPHASIS_LEVEL*
- *MSK_IPAR_MIO_NUMERICAL_EMPHASIS_LEVEL*
- *MSK_IPAR_MIO_PRESOLVE_AGGREGATOR_USE*
- *MSK_IPAR_MIO_QCQO_REFORMULATION_METHOD*
- *MSK_IPAR_MIO_SYMMETRY_LEVEL*
- *MSK_IPAR_NG*
- *MSK_IPAR_PTF_WRITE_PARAMETERS*
- *MSK_IPAR_PTF_WRITE_SOLUTIONS*
- *MSK_IPAR_REMOTE_USE_COMPRESSION*
- *MSK_IPAR_SIM_DETECT_PWL*
- *MSK_IPAR_WRITE_JSON_INDENTATION*
- *MSK_SPAR_REMOTE_OPTSERVER_HOST*
- *MSK_SPAR_REMOTE_TLS_CERT*
- *MSK_SPAR_REMOTE_TLS_CERT_PATH*

Removed

- MSK_IPAR_INTPNT_MULTI_THREAD
- MSK_IPAR_READ_LP_DROP_NEW_VARS_IN_BOU
- MSK_IPAR_READ_LP_QUOTED_NAMES
- MSK_IPAR_WRITE_LP_QUOTED_NAMES
- MSK_IPAR_WRITE_LP_STRICT_FORMAT
- MSK_IPAR_WRITE_LP_TERMS_PER_LINE
- MSK_IPAR_WRITE_PRECISION
- MSK_SPAR_REMOTE_ACCESS_TOKEN
- MSK_SPAR_STAT_FILE_NAME

18.5 Constants compared to version 9

Added

- *MSK_CALLBACK_BEGIN_SOLVE_ROOT_RELAX*
- *MSK_CALLBACK_END_SOLVE_ROOT_RELAX*
- *MSK_CALLBACK_UPDATE_SIMPLEX*
- *MSK_DINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_DENSITY*
- *MSK_DINF_MIO_INITIAL_FEASIBLE_SOLUTION_OBJ*
- *MSK_DINF_MIO_LIPRO_SEPARATION_TIME*
- *MSK_DINF_MIO_ROOT_TIME*
- *MSK_DINF_PRESOLVE_TOTAL_PRIMAL_PERTURBATION*
- *MSK_DINF_READ_DATA_TIME*
- *MSK_DINF_REMOTE_TIME*
- *MSK_DINF_SOL_ITG_PVIOLACC*
- *MSK_DINF_SOL_ITG_PVIOLDJC*
- *MSK_DINF_SOL_ITR_DVIOLACC*
- *MSK_DINF_SOL_ITR_PVIOLACC*
- *MSK_DINF_WRITE_DATA_TIME*
- *MSK_IINF_MIO_INITIAL_FEASIBLE_SOLUTION*
- *MSK_IINF_MIO_NUM_LIPRO_CUTS*
- *MSK_IINF_MIO_NUMDJC*
- *MSK_IINF_MIO_PRESOLVED_NUMDJC*
- *MSK_IINF_PRESOLVE_NUM_PRIMAL_PERTURBATIONS*
- *MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_COLUMNS*
- *MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_NZ*
- *MSK_LIINF_ANA_PRO_SCALARIZED_CONSTRAINT_MATRIX_NUM_ROWS*
- *MSK_LIINF_MIO_NUM_DUAL_ILLPOSED_CER*
- *MSK_LIINF_MIO_NUM_PRIM_ILLPOSED_CER*
- *MSK_LIINF_RD_NUMACC*
- *MSK_LIINF_RD_NUMDJC*
- *MSK_LIINF_SIMPLEX_ITER*

Removed

- MSK_CALLBACKCODE_BEGIN_FULL_CONVEXITY_CHECK
- MSK_CALLBACKCODE_END_FULL_CONVEXITY_CHECK
- MSK_CALLBACKCODE_IM_FULL_CONVEXITY_CHECK
- MSK_DINFITEM_RD_TIME
- MSK_SCALINGTYPE_AGGRESSIVE
- MSK_SCALINGTYPE_MODERATE

18.6 Response Codes compared to version 9

Added

- *MSK_RES_ERR_ACC_AFE_DOMAIN_MISMATCH*
- *MSK_RES_ERR_ACC_INVALID_ENTRY_INDEX*
- *MSK_RES_ERR_ACC_INVALID_INDEX*
- *MSK_RES_ERR_AFE_INVALID_INDEX*
- *MSK_RES_ERR_ARGUMENT_IS_TOO_SMALL*
- *MSK_RES_ERR_AXIS_NAME_SPECIFICATION*
- *MSK_RES_ERR_CBF_DUPLICATE_PSDCON*
- *MSK_RES_ERR_CBF_INVALID_DIMENSION_OF_PSDCON*
- *MSK_RES_ERR_CBF_INVALID_NUM_PSDCON*
- *MSK_RES_ERR_CBF_INVALID_PSDCON_BLOCK_INDEX*
- *MSK_RES_ERR_CBF_INVALID_PSDCON_INDEX*
- *MSK_RES_ERR_CBF_INVALID_PSDCON_VARIABLE_INDEX*
- *MSK_RES_ERR_CBF_UNSUPPORTED_CHANGE*
- *MSK_RES_ERR_DIMENSION_SPECIFICATION*
- *MSK_RES_ERR_DJC_AFE_DOMAIN_MISMATCH*
- *MSK_RES_ERR_DJC_DOMAIN_TERMSIZE_MISMATCH*
- *MSK_RES_ERR_DJC_INVALID_INDEX*
- *MSK_RES_ERR_DJC_INVALID_TERM_SIZE*
- *MSK_RES_ERR_DJC_TOTAL_NUM_TERMS_MISMATCH*
- *MSK_RES_ERR_DJC_UNSUPPORTED_DOMAIN_TYPE*
- *MSK_RES_ERR_DOMAIN_DIMENSION*
- *MSK_RES_ERR_DOMAIN_DIMENSION_PSD*
- *MSK_RES_ERR_DOMAIN_INVALID_INDEX*
- *MSK_RES_ERR_DOMAIN_POWER_INVALID_ALPHA*
- *MSK_RES_ERR_DOMAIN_POWER_NEGATIVE_ALPHA*

- *MSK_RES_ERR_DOMAIN_POWER_NLEFT*
- *MSK_RES_ERR_DUPLICATE_DJC_NAMES*
- *MSK_RES_ERR_DUPLICATE_DOMAIN_NAMES*
- *MSK_RES_ERR_DUPLICATE_FIJ*
- *MSK_RES_ERR_HUGE_FIJ*
- *MSK_RES_ERR_INDEX_IS_NOT_UNIQUE*
- *MSK_RES_ERR_INF_IN_DOUBLE_DATA*
- *MSK_RES_ERR_INVALID_B*
- *MSK_RES_ERR_INVALID_CFIX*
- *MSK_RES_ERR_INVALID_FIJ*
- *MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_AFFINE_CONIC_CONSTRAINTS*
- *MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_DISJUNCTIVE_CONSTRAINTS*
- *MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_QUADRATIC_TERMS*
- *MSK_RES_ERR_INVALID_G*
- *MSK_RES_ERR_LICENSE_OLD_SERVER_VERSION*
- *MSK_RES_ERR_LP_INDICATOR_VAR*
- *MSK_RES_ERR_MISMATCHING_DIMENSION*
- *MSK_RES_ERR_MPS_INVALID_BOUND_KEY*
- *MSK_RES_ERR_MPS_INVALID_CON_KEY*
- *MSK_RES_ERR_MPS_INVALID_INDICATOR_CONSTRAINT*
- *MSK_RES_ERR_MPS_INVALID_INDICATOR_QUADRATIC_CONSTRAINT*
- *MSK_RES_ERR_MPS_INVALID_INDICATOR_VALUE*
- *MSK_RES_ERR_MPS_INVALID_INDICATOR_VARIABLE*
- *MSK_RES_ERR_MPS_INVALID_KEY*
- *MSK_RES_ERR_MPS_INVALID_SEC_NAME*
- *MSK_RES_ERR_MPS_WRITE_CPLEX_INVALID_CONE_TYPE*
- *MSK_RES_ERR_NO_DOTY*
- *MSK_RES_ERR_NOT_POWER_DOMAIN*
- *MSK_RES_ERR_OPF_DUAL_INTEGER_SOLUTION*
- *MSK_RES_ERR_OPF_DUPLICATE_BOUND*
- *MSK_RES_ERR_OPF_DUPLICATE_CONE_ENTRY*
- *MSK_RES_ERR_OPF_DUPLICATE_CONSTRAINT_NAME*
- *MSK_RES_ERR_OPF_INCORRECT_TAG_PARAM*
- *MSK_RES_ERR_OPF_INVALID_CONE_TYPE*
- *MSK_RES_ERR_OPF_INVALID_TAG*
- *MSK_RES_ERR_OPF_MISMATCHED_TAG*

- *MSK_RES_ERR_OPF_SYNTAX*
- *MSK_RES_ERR_OPF_TOO_LARGE*
- *MSK_RES_ERR_PTF_INCOMPATIBILITY*
- *MSK_RES_ERR_PTF_INCONSISTENCY*
- *MSK_RES_ERR_PTF_UNDEFINED_ITEM*
- *MSK_RES_ERR_SERVER_ACCESS_TOKEN*
- *MSK_RES_ERR_SERVER_ADDRESS*
- *MSK_RES_ERR_SERVER_CERTIFICATE*
- *MSK_RES_ERR_SERVER_TLS_CLIENT*
- *MSK_RES_ERR_SPARSITY_SPECIFICATION*
- *MSK_RES_ERR_UNALLOWED_WHICHSOL*
- *MSK_RES_TRM_LOST_RACE*
- *MSK_RES_WRN_INVALID_MPS_NAME*
- *MSK_RES_WRN_INVALID_MPS_OBJ_NAME*
- *MSK_RES_WRN_LARGE_FIJ*
- *MSK_RES_WRN_MODIFIED_DOUBLE_PARAMETER*
- *MSK_RES_WRN_NO_INFEASIBILITY_REPORT_WHEN_MATRIX_VARIABLES*
- *MSK_RES_WRN_PRESOLVE_PRIMAL_PERTUBATIONS*
- *MSK_RES_WRN_WRITE_LP_DUPLICATE_CON_NAMES*
- *MSK_RES_WRN_WRITE_LP_DUPLICATE_VAR_NAMES*
- *MSK_RES_WRN_WRITE_LP_INVALID_CON_NAMES*
- *MSK_RES_WRN_WRITE_LP_INVALID_VAR_NAMES*

Removed

- MSK_RES_ERR_LP_FORMAT
- MSK_RES_ERR_MPS_INV_BOUND_KEY
- MSK_RES_ERR_MPS_INV_CON_KEY
- MSK_RES_ERR_MPS_INV_SEC_NAME
- MSK_RES_ERR_OPF_FORMAT
- MSK_RES_ERR_OPF_NEW_VARIABLE
- MSK_RES_WRN_EXP_CONES_WITH_VARIABLES_FIXED_AT_ZERO
- MSK_RES_WRN_POW_CONES_WITH_ROOT_FIXED_AT_ZERO
- MSK_RES_WRN_QUAD_CONES_WITH_ROOT_FIXED_AT_ZERO
- MSK_RES_WRN_RQUAD_CONES_WITH_ROOT_FIXED_AT_ZERO

Bibliography

[AA95] E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Math. Programming*, 71(2):221–245, 1995.

[AGMeszarosX96] E. D. Andersen, J. Gondzio, Cs. Mészáros, and X. Xu. Implementation of interior point methods for large scale linear programming. In T. Terlaky, editor, *Interior-point methods of mathematical programming*, pages 189–252. Kluwer Academic Publishers, 1996.

[ART03] E. D. Andersen, C. Roos, and T. Terlaky. On implementing a primal-dual interior-point method for conic quadratic optimization. *Math. Programming*, February 2003.

[AY96] E. D. Andersen and Y. Ye. Combining interior-point and pivoting algorithms. *Management Sci.*, 42(12):1719–1731, December 1996.

[And09] Erling D. Andersen. The homogeneous and self-dual model and algorithm for linear optimization. Technical Report TR-1-2009, MOSEK ApS, 2009. URL: http://docs.mosek.com/whitepapers/homolo.pdf.

[And13] Erling D. Andersen. On formulating quadratic functions in optimization models. Technical Report TR-1-2013, MOSEK ApS, 2013. Last revised 23-feb-2016. URL: http://docs.mosek.com/whitepapers/qmodel.pdf.

[BKVH07] S. Boyd, S.J. Kim, L. Vandenberghe, and A. Hassibi. A Tutorial on Geometric Programming. *Optimization and Engineering*, 8(1):67–127, 2007. Available at http://www.stanford.edu/ boyd/gp_tutorial.html.

[Chvatal83] V. Chvátal. *Linear programming.* W.H. Freeman and Company, 1983.

[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

Parameters

Types

Index