MOSEK Optimizer API for C

Release 8.1.0.56

MOSEK ApS

2018

CONTENTS

CHAPTER ONE

INTRODUCTION

The **MOSEK** Optimization Suite 8.1.0.56 is a powerful software package capable of solving large-scale optimization problems of the following kind:

- linear,
- conic quadratic (also known as second-order cone),
- convex quadratic,
- semidefinite,
- and general convex.

Integer constrained variables are supported for all problem classes except for semidefinite and general convex problems. 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

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

where $\mathcal{K} = \{y : y \geq 0\}$, i.e.,

$$\begin{aligned} &Ax - b = y, \\ &y \in \mathcal{K}. \end{aligned}$$

In conic optimization a wider class of convex sets $\mathcal{K}$ is allowed, for example in 3 dimensions $\mathcal{K}$ may correspond to an ice cream cone. The conic optimizer in **MOSEK** supports three structurally different types of cones $\mathcal{K}$, which allows a surprisingly large number of nonlinear relations to be modelled (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)
- Convex Quadratic and Quadratically Constrained Optimization (QCQO)
- Semidefinite Optimization (SDO)
- General and Separable Convex Optimization (SCO)

as well as additional interfaces for:

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

CHAPTER TWO

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

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

CHAPTER THREE

LICENSE AGREEMENT

Before using the **MOSEK** software, please read the license agreement available in the distribution at `<MSKHOME>/mosek/8/mosek-eula.pdf` or on the **MOSEK** website https://mosek.com/products/license-agreement.

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

zlib

MOSEK includes the *zlib* library obtained from the zlib website. The license agreement for *zlib* is shown in Listing 3.1.

Listing 3.1: *zlib* license.

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

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

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

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

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

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

fplib

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

Listing 3.2: *fplib* license.

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

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

CHAPTER

FOUR

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	gcc ($\geq$ 4.5)	glibc ($\geq$ 2.2)
Mac OS 64 bit	Xcode ($\geq$ 5)	MAC OS SDK ($\geq$ 10.7)
Windows 32 and 64 bit	Visual Studio ($\geq$ 2010)	

In many cases older versions can also be used.

Locating Files

The files in 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/8/tools/platform/<PLATFORM>/h`	Header files	`<HEADERDIR>`
`<MSKHOME>/mosek/8/tools/platform/<PLATFORM>/bin`	Libraries and DLLs	`<DLLDIR>`, `<LIBDIR>`
`<MSKHOME>/mosek/8/tools/examples/c`	Examples	`<EXDIR>`
`<MSKHOME>/mosek/8/tools/examples/data`	Additional data	`<MISCDIR>`

where

- `<MSKHOME>` is the folder in which the **MOSEK** package 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/8/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.8.1` must be available at runtime.

Windows, 64bit

```
cl.exe /I<HEADERDIR> file.c /link /LIBPATH:<LIBDIR> /out:file.exe mosek64_8_1.lib
```

The shared library `mosek64_8_1.dll` must be available at runtime.

Windows, 32bit

```
cl.exe /I<HEADERDIR> file.c /link /LIBPATH:<LIBDIR> /out:file.exe mosek8_1.lib
```

The shared library `mosek8_1.dll` must be available at runtime.

Mac OS

```
clang file.c -o file -I<HEADERDIR> -L<LIBDIR> -Wl,-headerpad,128 -lmosek64
install_name_tool -change libmosek64.8.1.dylib <LIBDIR>/libmosek64.8.1.dylib file
```

The shared library `libmosek64.8.1.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_8_1.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_8_1.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 to 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 Mac OS 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

FIVE

DESIGN OVERVIEW

5.1 Modelling

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

Every interaction with **MOSEK** using Optimizer API for C begins 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. An environment can host multiple tasks.

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

int main() {
  MSKrescodee      r;
  MSKenv_t         env  = NULL;
  MSKtask_t        task = NULL;
  double           xx = 0.0;

  MSK_makeenv(&env, NULL);             // Create environment
  MSK_maketask(env, 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_optimize(task);                      // Optimize

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

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

CHAPTER

SIX

OPTIMIZATION TUTORIALS

In this section we demonstrate how to set up basic types of optimization problems. Each short tutorial contains a working example of formulating problems, defining variables and constraints and retrieving solutions.

6.1 Linear Optimization

The simplest optimization problem is a purely linear problem. A *linear optimization problem* is a problem of the following form:

Minimize or maximize the objective function

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

subject to the linear constraints

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

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

$$c = \left[\begin{array}{c} c_0 \\ \vdots \\ c_{n-1} \end{array} \right],$$

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

$$A = \left[\begin{array}{ccc} a_{0,0} & \cdots & a_{0,(n-1)} \\ \vdots & \cdots & \vdots \\ a_{(m-1),0} & \cdots & a_{(m-1),(n-1)} \end{array} \right],$$

- 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. 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.

Before setting up the optimization problem, a **MOSEK** environment must be created. All tasks in the program should share the same environment.

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

Create an optimization task.

Next, an empty task object is created:

```
  /* Create the optimization task. */
  r = MSK_maketask(env, 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.

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

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

and

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

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

The ptrb, ptre, asub, and aval arguments define the constraint matrix A in the column ordered sparse format (for details, see Sec. 16.1.4).

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 $i = 0, \ldots, \mathtt{numcon} - 1$

```
    /* 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_optimize*.

```
    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* or *MSK_SOL_STA_NEAR_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,
                            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 };
MSKenv_t           env  = NULL;
MSKtask_t          task = NULL;
MSKrescodee        r;
MSKint32t          i, j;

/* 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);

  /* 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.
   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;

        if ( r == MSK_RES_OK )
          r = MSK_getsolsta (task,
                             MSK_SOL_BAS,
                             &solsta);
        switch (solsta)
        {
          case MSK_SOL_STA_OPTIMAL:
          case MSK_SOL_STA_NEAR_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:
          case MSK_SOL_STA_NEAR_DUAL_INFEAS_CER:
          case MSK_SOL_STA_NEAR_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)
    {
      /* 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;
}
```

6.2 Quadratic Optimization

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2.1 Example: Quadratic Objective

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

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

The matrix formulation (6.4) has:

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

with the bounds:

$$l^c = 1, u^c = \infty, l^x = \left[\begin{array}{c} 0 \\ 0 \\ 0 \end{array} \right] \text{ and } u^x = \left[\begin{array}{c} \infty \\ \infty \\ \infty \end{array} \right]$$

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

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 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. 16.1.4):

```
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 6.2: Source code implementing problem (6.4).

```
#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;
  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)
        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:
          case MSK_SOL_STA_NEAR_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:
          case MSK_SOL_STA_NEAR_DUAL_INFEAS_CER:
          case MSK_SOL_STA_NEAR_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.\n");
            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 */
```

6.2.2 Example: Quadratic constraints

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

Consider the problem:

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

This is equivalent to

$$
\begin{array}{lccl}
\text{minimize} & \frac{1}{2}x^TQ^ox + c^Tx & & \\
\text{subject to} & \frac{1}{2}x^TQ^0x + Ax & \geq & b, \\
 & x \geq 0, & &
\end{array}
\tag{6.5}
$$

where

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

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

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

Setting up quadratic constraints

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 6.3: Implementation of the quadratically constrained problem (6.5).

```
#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,
                         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]);

      /* 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;
      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:
            case MSK_SOL_STA_NEAR_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:
            case MSK_SOL_STA_NEAR_DUAL_INFEAS_CER:
            case MSK_SOL_STA_NEAR_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.\n");
              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 */
```

6.3 Conic Quadratic Optimization

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

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

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

MOSEK can solve conic quadratic optimization problems of the form

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

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

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

For convenience, a user defining a conic quadratic problem only needs to specify subsets of variables x^t belonging to quadratic cones. These are:

- 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, the following constraint:

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

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

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

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

6.3.1 Example CQO1

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

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

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

A cone is defined using the function *MSK_appendcone*:

```
        csub[0] = 3;
        csub[1] = 0;
        csub[2] = 1;

        r = MSK_appendcone(task,
                           MSK_CT_QUAD,
                           0.0, /* For future use only, can be set to 0.0 */
                           3,
                           csub);
```

The first argument selects the type of quadratic cone, in this case either *MSK_CT_QUAD* for a *quadratic cone* or *MSK_CT_RQUAD* for a *rotated quadratic cone*. The second parameter is currently ignored and passing `0.0` will work.

The next argument denotes the number of variables in the cone, in this case *3*, and the last argument is a list of indexes of the variables appearing in the cone.

Variants of this method are available to append multiple cones at a time.

Source code

Listing 6.4: Source code solving problem (6.6).

```
#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;

  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};


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);

    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.*/

    if ( r == MSK_RES_OK )
    {
      /* Append the first cone. */
      csub[0] = 3;
      csub[1] = 0;
      csub[2] = 1;

      r = MSK_appendcone(task,
                         MSK_CT_QUAD,
                         0.0, /* For future use only, can be set to 0.0 */
                         3,
                         csub);
    }

    if ( r == MSK_RES_OK )
    {
      /* Append the second cone. */
      csub[0] = 4;
      csub[1] = 5;
      csub[2] = 2;

      r = MSK_appendcone(task,
                         MSK_CT_RQUAD,
                         0.0,
                         3,
                         csub);
    }

    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:
        case MSK_SOL_STA_NEAR_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
            {
              r = MSK_RES_ERR_SPACE;
            }
            free(xx);
          }
          break;
        case MSK_SOL_STA_DUAL_INFEAS_CER:
        case MSK_SOL_STA_PRIM_INFEAS_CER:
        case MSK_SOL_STA_NEAR_DUAL_INFEAS_CER:
        case MSK_SOL_STA_NEAR_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.\n");
          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 Semidefinite Optimization

Semidefinite optimization is a generalization of conic quadratic 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 of the form

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

where the problem has p symmetric positive semidefinite variables $\overline{X}_j \in \mathcal{S}_+^{r_j}$ of dimension r_j with symmetric coefficient matrices $\overline{C}_j \in \mathcal{S}^{r_j}$ and $\overline{A}_{i,j} \in \mathcal{S}^{r_j}$. We use standard notation for the matrix inner product, i.e., for $A, B \in \mathbb{R}^{m \times n}$ we have

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

6.4.1 Example SDO1

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

$$\begin{array}{llcl} \text{minimize} & \left\langle \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, \overline{X} \right\rangle + x_0 & & \\ \text{subject to} & \left\langle \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \overline{X} \right\rangle + x_0 & = & 1, \\ & \left\langle \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \overline{X} \right\rangle + x_1 + x_2 & = & 1/2, \\ & x_0 \geq \sqrt{{x_1}^2 + {x_2}^2}, & & \overline{X} \succeq 0, \end{array} \tag{6.7}$$

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 a conic quadratic variable $(x_0, x_1, x_2) \in \mathcal{Q}^3$. 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 quadratic part

The linear and quadratic parts (constraints, variables, objective, cones) are set up using the methods described in the relevant tutorials; Sec. 6.1 and Sec. 6.3. 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 6.5: Source code solving problem (6.7).

```
#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 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};

  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 '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,
                          -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 conic quadratic cone */
    if ( r == MSK_RES_OK )
      r = MSK_appendcone(task,
                         MSK_CT_QUAD,
                         0.0,
                         3,
                         conesub);

    /* 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);

    /* 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:
```

```
          case MSK_SOL_STA_NEAR_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;

          case MSK_SOL_STA_DUAL_INFEAS_CER:
          case MSK_SOL_STA_PRIM_INFEAS_CER:
          case MSK_SOL_STA_NEAR_DUAL_INFEAS_CER:
          case MSK_SOL_STA_NEAR_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.\n");
            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 */
```

6.5 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 and conic quadratic problems. See the previous tutorials for an introduction to how to model these types of problems.

6.5.1 Example MILO1

We use the example

$$
\begin{array}{lccl}
\text{maximize} & x_0 + 0.64 x_1 & & \\
\text{subject to} & 50 x_0 + 31 x_1 & \leq & 250, \\
 & 3 x_0 - 2 x_1 & \geq & -4, \\
 & x_0, x_1 \geq 0 & & \text{and integer}
\end{array}
\tag{6.8}
$$

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. 14 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 6.6. 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 6.6: Source code implementing problem (6.8).

```
#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);

  /* 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;

    /* 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:
          case MSK_SOL_STA_NEAR_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.\n");
                  break;
                default:
                  printf("Other problem status.");
                  break;
              }
            }
            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 */
```

6.5.2 Specifying an initial solution

Solution time of can often be reduced by providing an initial solution for the solver. It is not necessary to specify the whole solution. By setting the *MSK_IPAR_MIO_CONSTRUCT_SOL* parameter to *MSK_ON* and inputting values for the integer variables only, **MOSEK** will be forced to compute the remaining continuous variable values. If the specified integer solution is infeasible or incomplete, **MOSEK** will simply ignore it.

We concentrate on a simple example below.

$$\begin{array}{ll} \text{maximize} & 7x_0 + 10x_1 + x_2 + 5x_3 \\ \text{subject to} & x_0 + x_1 + x_2 + x_3 \leq 2.5 \\ & x_0, x_1, x_2 \in \mathbb{Z} \\ & x_0, x_1, x_2, x_3 \geq 0 \end{array} \tag{6.9}$$

Solution values can be set using *MSK_putxxslice* and related methods.

Listing 6.7: Implementation of problem (6.9) specifying an initial solution.

```
  /* Construct an initial feasible solution from the
     values of the integer variables specified */
  if (r == MSK_RES_OK)
    r = MSK_putintparam(task, MSK_IPAR_MIO_CONSTRUCT_SOL, MSK_ON);

  if (r == MSK_RES_OK)
  {
    double xx[] = {0.0, 2.0, 0.0};

    /* Assign values 0,2,0 to integer variables */
    r = MSK_putxxslice(task, MSK_SOL_ITG, 0, 3, xx);
  }
```

The complete code is not very different from the first example and is available for download as `mioinitsol.c`. For more details about this process see Sec. 14.

6.6 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. 15.3.

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

Parameter settings (see Sec. 7.4) can also be changed between optimizations.

6.6.1 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:

$$\begin{array}{lccccccl} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & & \\ \text{subject to} & 2x_0 & + & 4x_1 & + & 3x_2 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & \leq & 60000, \end{array} \tag{6.10}$$

and

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

Code in Listing 6.8 loads and solves this problem.

Listing 6.8: Setting up and solving problem (6.10)

```
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;
MSKenv_t        env;
MSKtask_t       task;
MSKint32t       varidx, conidx;
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);

  /* 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);

  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);
```

6.6.2 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:

$$\begin{array}{lccccccc} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & & \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & \leq & 60000, \end{array} \tag{6.11}$$

and

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

After this operation we can reoptimize the problem.

6.6.3 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 6.9

Listing 6.9: 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,
                        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:

$$\begin{array}{lccccccccc} \text{maximize} & 1.5x_0 & + & 2.5x_1 & + & 3.0x_2 & + & 1.0x_3 & & \\ \text{subject to} & 3x_0 & + & 4x_1 & + & 3x_2 & + & 4x_3 & \leq & 100000, \\ & 3x_0 & + & 2x_1 & + & 3x_2 & & & \leq & 50000, \\ & 2x_0 & + & 3x_1 & + & 2x_2 & + & 1x_3 & \leq & 60000, \end{array} \tag{6.12}$$

and

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

6.6.4 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 6.10: Adding a new constraint.

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

    /* 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)
  {
    MSKidxt 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.

6.7 Solution Analysis

The main purpose of **MOSEK** is to solve optimization problems and therefore the most fundamental question to be asked is whether the solution reported by **MOSEK** is a solution to the desired optimization problem.

There can be several reasons why it might be not case. The most prominent reasons are:

- A wrong problem. The problem inputted to **MOSEK** is simply not the right problem, i.e. some of the data may have been corrupted or the model has been incorrectly built.
- Numerical issues. The problem is badly scaled or otherwise badly posed.
- Other reasons. E.g. not enough memory or an explicit user request to stop.

The first step in verifying that **MOSEK** reports the expected solution is to inspect the solution summary generated by **MOSEK** (see Sec. 6.7.1). The solution summary provides information about

- the problem and solution statuses,
- objective value and infeasibility measures for the primal solution, and
- objective value and infeasibility measures for the dual solution, where applicable.

By inspecting the solution summary it can be verified that **MOSEK** produces a feasible solution, and, in the continuous case, the optimality can be checked using the dual solution. Furthermore, the problem itself ca be inspected using the problem analyzer discussed in Sec. 15.1.

If the summary reports conflicting information (e.g. a solution status that does not match the actual solution), or the cause for terminating the solver before a solution was found cannot be traced back to the reasons stated above, it may be caused by a bug in the solver; in this case, please contact **MOSEK** support (see Sec. 2).

If it has been verified that **MOSEK** solves the problem correctly but the solution is still not as expected, next step is to verify that the primal solution satisfies all the constraints. Hence, using the original problem it must be determined whether the solution satisfies all the required constraints in the model.

For instance assume that the problem has the constraints

$$\begin{array}{rcl} x_1 + 2x_2 + x_3 & \leq & 1, \\ x_1, x_2, x_3 \geq 0 & & \end{array}$$

and **MOSEK** reports the optimal solution

$$x_1 = x_2 = x_3 = 1.$$

Then clearly the solution violates the constraints. The most likely explanation is that the model does not match the problem entered into **MOSEK**, for instance

$$x_1 - 2x_2 + x_3 \leq 1$$

may have been inputted instead of

$$x_1 + 2x_2 + x_3 \leq 1.$$

A good way to debug such an issue is to dump the problem to *OPF file* and check whether the violated constraint has been specified correctly.

Verifying that a feasible solution is optimal can be harder. However, for continuous problems, i.e. problems without any integer constraints, optimality can verified using a dual solution. Normally, **MOSEK** will report a dual solution; if that is feasible and has the same objective value as the primal solution, then the primal solution must be optimal.

An alternative method is to find another primal solution that has better objective value than the one reported to **MOSEK**. If that is possible then either the problem is badly posed or there is bug in **MOSEK**.

6.7.1 The Solution Summary

Due to **MOSEK** employs finite precision floating point numbers then reported solution is an approximate optimal solution. Therefore after solving an optimization problem it is relevant to investigate how good an approximation the solution is. For a convex optimization problem that is an easy task because the optimality conditions are:

- The primal solution must satisfy all the primal constraints.
- The dual solution much satisfy all the dual constraints.
- The primal and dual objective values must be identical.

Therefore, the **MOSEK** solution summary displays that information that makes it possible to verify the optimality conditions. Indeed the solution summary reports how much primal and dual solutions violate the primal and constraints respectively. In addition the objective values assoctaied with each solution repoted.

In case of a linear optimization problem the solution summary may look like

```
Basic solution summary
  Problem status  : PRIMAL_AND_DUAL_FEASIBLE
  Solution status : OPTIMAL
  Primal.  obj: -4.6475314286e+002  nrm: 5e+002   Viol.  con: 1e-014   var: 1e-014
  Dual.    obj: -4.6475314543e+002  nrm: 1e+001   Viol.  con: 4e-009   var: 4e-016
```

The interpreation of the solution summary is as follows:

- Information for the basic solution is reported.
- The problem status is primal and dual feasible which means the problem has an optimal solution.
- The solution status is optimal.

- Next information about the primal solution is reported. The information consists of the objective value, the infinity norm of the primal solution and violation meassures. The violation for the constraints (`con:`) is the maximal violation in any of the constraints. Whereas the violations for the variables (`var:`) is the maximal bound violation for any of the variables. In this case the primal violations for the constraints and variables are small meaning the solution is an almost feasible solution. Observe due to the rounding errors it can be expected that the violations are proportional to the size (`nrm:`) of the solution.
- Similarly for the dual solution the violations are small and hence the dual solution is almost feasible.
- Finally, it can be seen that the primal and dual objective values are almost identical.

To summarize in this case a primal and a dual solution only violate the primal and dual constraints slightly. Moreover, the primal and dual objective values are almost identical and hence it can be concluded that the reported solution is a good approximation to the optimal solution.

The reason the size (=norms) of the solution are shown is that it shows some about conditioning of the problem because if the primal and/or dual solution has very large norm then the violations and objective values are sensitive to small pertubations in the problem data. Therefore, the problem is unstable and care should be taken before using the solution.

Observe the function *MSK_solutionsummary* will print out the solution summary. In addition

- the problem status can be obtained using *MSK_getprosta*.
- the solution status can be obtained using *MSK_getsolsta*.
- the primal constraint and variable violations can be obtained with *MSK_getpviolcon* and *MSK_getpviolvar*.
- the dual constraint and variable violations can be obtained with *MSK_getdviolcon* and *MSK_getdviolvar* respectively.
- the primal and dual objective values can be obtained with *MSK_getprimalobj* and *MSK_getdualobj*.

Now what happens if the problem does not have an optimal solution e.g. is primal infeasible. In such a case the solution summary may look like

```
Interior-point solution summary
  Problem status  : PRIMAL_INFEASIBLE
  Solution status : PRIMAL_INFEASIBLE_CER
  Dual.    obj: 6.7319732555e+000   nrm: 8e+000   Viol.  con: 3e-010   var: 2e-009
```

i.e. **MOSEK** reports that the solution is a certificate of primal infeasibility but a certificate of primal infeasibility what does that mean? It means that the dual solution is a Farkas type certificate. Recall Farkas' Lemma says

$$\begin{array}{rcl} Ax & = & b, \\ x & \geq & 0 \end{array}$$

if and only if a y exists such that

$$\begin{array}{rcl} A^T y & \leq & 0, \\ b^T y & > & 0. \end{array} \tag{6.13}$$

Observe the infeasibility certificate has the same form as a regular dual solution and therefore the certificate is stored as a dual solution. In order to check quality of the primal infeasibility certificate it should be checked whether satisfies (6.13). Hence, the dual objective value is $b^T y$ should be strictly positive and the maximal violation in $A^T y \leq 0$ should be a small. In this case we conclude the certificate is of high quality because the dual objective is postive and large compared to the violations. Note the Farkas certificate is a ray so any postive multiple of that ray is also certificate. This implies the absolute of the value objective value and the violation is not relevant.

In the case a problem is dual infeasible then the solution summary may look like

```
Basic solution summary
Problem status  : DUAL_INFEASIBLE
Solution status : DUAL_INFEASIBLE_CER
Primal.  obj: -2.0000000000e-002  nrm: 1e+000   Viol.  con: 0e+000   var: 0e+000
```

Observe when a solution is a certificate of dual infeasibility then the primal solution contains the certificate. Moreoever, given the problem is a minimization problem the objective value should be negative and large compared to the worst violation if the certificate is strong.

Listing 6.11 shows how to use these function to determine the quality of the solution.

Listing 6.11: An example of solution quality analysis.

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

static double dmin(double x,
                   double y)
{
  return ( x <= y ) ? ( x ) : ( y );
} /* dmin */

static double dmax(double x,
                   double y)
{
  return ( x >= y ) ? ( x ) : ( y );
} /* dmax */

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


int main (int argc, const char * argv[])
{
  double max_primal_viol, /* maximal primal violation */
         max_dual_viol,  /* maximal dual violation */
         abs_obj_gap,
         rel_obj_gap;

  MSKenv_t     env      = NULL;

  MSKint32t   numvar, j;

  MSKsolstae  solsta;
  MSKsoltypee whichsol = MSK_SOL_BAS;

  MSKrealt    primalobj, pviolcon, pviolvar, pviolbarvar, pviolcones, pviolitg,
              dualobj, dviolcon, dviolvar, dviolbarvar, dviolcones, xj;

  MSKrescodee r        = MSK_RES_OK;
  MSKrescodee trmcode;

  MSKtask_t   task     = NULL;

  int         accepted = 0;

  if ( argc <= 1)
  {
    printf ("Missing argument. The syntax is:\n");
    printf (" solutionquality inputfile\n");
```

```
}
else
{

  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);

  /* We assume that a problem file was given as the first command
     line argument (received in `argv'). */
  if ( r == MSK_RES_OK )
    r = MSK_readdata(task, argv[1]);

  /* Solve the problem */
  if ( r == MSK_RES_OK )
  {
    r = MSK_optimizetrm(task, &trmcode);
  }

  /* Print a summary of the solution. */
  MSK_solutionsummary(task, MSK_STREAM_MSG);

  if ( r == MSK_RES_OK )
  {

    MSK_getsolsta(task, whichsol, &solsta);

    r = MSK_getsolutioninfo(task,
                            whichsol,
                            &primalobj,
                            &pviolcon,
                            &pviolvar,
                            &pviolbarvar,
                            &pviolcones,
                            &pviolitg,
                            &dualobj,
                            &dviolcon,
                            &dviolvar,
                            &dviolbarvar,
                            &dviolcones);

    switch ( solsta )
    {
      case MSK_SOL_STA_OPTIMAL:
      case MSK_SOL_STA_NEAR_OPTIMAL:
        {

          abs_obj_gap     = fabs(dualobj - primalobj);
          rel_obj_gap     = abs_obj_gap / (1.0 + dmin(fabs(primalobj), fabs(dualobj)));
          max_primal_viol = dmax(pviolcon, pviolvar);
          max_primal_viol = dmax(max_primal_viol, pviolbarvar);
          max_primal_viol = dmax(max_primal_viol, pviolcones);

          max_dual_viol   = dmax(dviolcon, dviolvar);
          max_dual_viol   = dmax(max_dual_viol, dviolbarvar);
          max_dual_viol   = dmax(max_dual_viol, dviolcones);

          /* Assume the application needs the solution to be within
             1e-6 optimality in an absolute sense. Another approach
```

```
         would be looking at the relative objective gap */

      printf("\n\n");
      printf("Customized solution information.\n");
      printf("  Absolute objective gap: %e\n", abs_obj_gap);
      printf("  Relative objective gap: %e\n", rel_obj_gap);
      printf("  Max primal violation  : %e\n", max_primal_viol);
      printf("  Max dual violation    : %e\n", max_dual_viol);

      if ( rel_obj_gap > 1e-6 )
      {
        printf("Warning: The relative objective gap is LARGE.\n");
        accepted = 0;
      }

      /* We will accept a primal infeasibility of 1e-8 and
         dual infeasibility of 1e-6. These number should chosen problem
         dependent.
       */

      if ( max_primal_viol > 1e-8 )
      {
        printf("Warning: Primal violation is too LARGE.\n");
        accepted = 0;
      }

      if ( max_dual_viol > 1e-6 )
      {
        printf("Warning: Dual violation is too LARGE.\n");
        accepted = 0;
      }

      if ( accepted )
      {

        if ( MSK_RES_OK == MSK_getnumvar(task, &numvar) )
        {
          printf("Optimal primal solution\n");
          for (j = 0; j < numvar && r == MSK_RES_OK; ++j)
          {
            r = MSK_getxxslice(task, whichsol, j, j + 1, &xj);
            if ( r == MSK_RES_OK )
              printf("x[%d]: %e\n", j, xj);
          }
        }
      }
      else if ( r == MSK_RES_OK )
      {
        /* Print detailed information about the solution */
        r = MSK_analyzesolution(task, MSK_STREAM_LOG, whichsol);
      }
      break;
    }
  case MSK_SOL_STA_DUAL_INFEAS_CER:
  case MSK_SOL_STA_PRIM_INFEAS_CER:
  case MSK_SOL_STA_NEAR_DUAL_INFEAS_CER:
  case MSK_SOL_STA_NEAR_PRIM_INFEAS_CER:
    printf("Primal or dual infeasibility certificate found.\n");
    break;
  case MSK_SOL_STA_UNKNOWN:
    printf("The status of the solution is unknown.\n");
    break;
  default:
```

```
        printf("Other solution status");
        break;
      }
    }

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

6.7.2 The Solution Summary for Mixed-Integer Problems

The solution summary for a mixed-integer problem may look like

Listing 6.12: Example of solution summary for a mixed-integer problem.

```
Integer solution solution summary
Problem status  : PRIMAL_FEASIBLE
Solution status : INTEGER_OPTIMAL
Primal.  obj: 3.4016000000e+005   nrm: 1e+000   Viol.  con: 0e+000   var: 0e+000   itg: 3e-014
```

The main diffrence compared to thecontinous case covered previously is that no information about the dual solution is provided. Simply because there is no dual solution available for a mixed integer problem. In this case it can be seen that the solution is highly feasible because the violations are small. Moreoever, the solution is denoted integer optimal. Observe *itg: 3e-014* implies that all the integer constrained variables are at most $3e - 014$ from being an exact integer.

For a more in-depth treatment see the following sections:

- *Case studies* for more advanced and complicated optimization examples.
- *Problem Formulation and Solutions* for formal mathematical formulations of problems **MOSEK** can solve, dual problems and infeasibility certificates.

CHAPTER

SEVEN

SOLVER INTERACTION TUTORIALS

In this section we cover the interaction with the solver.

7.1 Accessing the solution

This section contains important information about the status of the solver and the status of the solution, which must be checked in order to properly interpret the results of the optimization.

7.1.1 Solver termination

The optimizer provides 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).

When using the method *MSK_optimize*, the response code or termination code most relevant for the user will be returned. To receive both codes separately call the function *MSK_optimizetrm*.

If a runtime error causes the program to crash during optimization, the first debugging step is to enable logging and check the log output. See Sec. 7.3.

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

7.1.2 Available solutions

MOSEK uses three kinds of optimizers and provides three types of solutions:

- basic solution (`BAS`, from the simplex optimizer),
- interior-point solution (`ITR`, from the interior-point optimizer),
- integer solution (`ITG`, 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.

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

7.1.3 Problem and solution status

Assuming that the optimization terminated without errors, the next important step is to check the problem and solution status. There is one for every type of solution, as explained above.

Problem status

Problem status (*MSKprostae*, retrieved with *MSK_getprosta*) 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*, retrieved with *MSK_getsolsta*) 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.
- **near optimal** (*MSK_SOL_STA_NEAR_OPTIMAL*) — the solution values are feasible and they were certified to be at least nearly optimal up to some accuracy.
- **certificate** — the solution is in fact a certificate of infeasibility (primal or dual, depending on the solution).
- **unknown/undefined** — the solver could not solve the problem or this type of solution is not available for a given problem.

The solution status determines the action to be taken. For example, in some cases a suboptimal solution may still be valuable and deserve attention. It is the user's responsibility to check the status and quality of the solution.

Typical status reports

Here are the most typical optimization outcomes described in terms of the problem and solution statuses. Note that these do not cover all possible situations that can occur.

Table 7.2: Continuous problems (solution status for `MSK_SOL_ITR` or `MSK_SOL_BAS`)

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	*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 `MSK_SOL_ITG`, 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*

7.1.4 Retrieving solution values

After the meaning and quality of the solution (or certificate) have been established, we can query for the actual numerical values. They can be accessed with methods such as:

- *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*.

7.1.5 Source code example

Below is a source code example with a simple framework for assessing and retrieving the solution to a conic quadratic 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;
```

```c
char        symname[MSK_MAX_STR_LEN];
char        desc[MSK_MAX_STR_LEN];
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);
    }

    /* 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:
        case MSK_SOL_STA_NEAR_OPTIMAL:
          printf("An optimal interior-point solution is located.\n");

          /* Read a 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]);
          free(xx);
          break;

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

```
          case MSK_SOL_STA_PRIM_INFEAS_CER:
          case MSK_SOL_STA_NEAR_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");
            if ( r != MSK_RES_OK )
            {
              /* Optimization error */
              MSK_getcodedesc(r, symname, desc);
              printf("  Response code: %s %s\n", symname, desc);
            }
            else
            {
              /* 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;
}
```

7.2 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_optimize* is invoked. We will say more about this in Sec. 7.1.

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. 7.3). A typical warning is, for example:

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

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

The user can also register a dedicated callback function to handle all errors and warnings. This is done with *MSK_putresponsefunc*.

7.3 Input/Output

The logging and I/O features are provided mainly by the **MOSEK** task and to some extent by the **MOSEK** environment objects.

7.3.1 Stream logging

By default the solver runs silently and does not produce any output to the console or otherwise. However, the log output can be redirected to a user-defined output stream or stream callback function. The log output is analogous to the one produced by the command-line version of **MOSEK**.

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);
}
```

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*.

7.3.2 Log verbosity

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

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

Each parameter controls the output level of a specific functionality or algorithm. The main switch is *MSK_IPAR_LOG* which 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.

7.3.3 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 filename's extension (unless the parameter *MSK_IPAR_WRITE_DATA_FORMAT* specifies something else). Supported formats are listed in Sec. 17 together with a table of problem types supported by each.

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

```
MSK_writedata(task,"data.opf");
MSK_optimize(task);
```

All formats can be compressed with `gzip` by appending the `.gz` 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.

7.3.4 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);
```

7.4 Setting solver parameters

MOSEK comes with a large number of parameters that allows the user to tune the behavior of the optimizer. The typical settings which can be changed with solver parameters include:

- choice of the optimizer for linear problems,
- choice of primal/dual solver,
- turning presolve on/off,
- turning heuristics in the mixed-integer optimizer on/off,
- level of multi-threading,
- feasibility tolerances,
- solver termination criteria,
- behaviour of the license manager,

and more. All parameters have default settings which will be suitable for most typical users.

The API reference contains:

- *Full list of parameters*
- *List of parameters grouped by topic*

Setting parameters

Each parameter is identified by a unique name. 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 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 7.2: 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);
```

7.5 Retrieving information items

After the optimization the user has access to the solution as well as to a report containing a large amount of additional *information items*. For example, one can obtain information about:

- **timing**: total optimization time, time spent in various optimizer subroutines, number of iterations, etc.
- **solution quality**: feasibility measures, solution norms, constraint and bound violations, etc.
- **problem structure**: counts of variables of different types, constraints, nonzeros, etc.

- **integer optimizer**: integrality gap, objective bound, number of cuts, etc.

and more. Information items are numerical values of integer, long integer or double type. The full list can be found in the API reference:

- *Double*
- *Integer*
- *Long*

Certain information items make sense, and are made available, also *during* the optimization process. They can be accessed from a callback function, see Sec. 7.6 for details.

Remark

For efficiency reasons, not all information items are automatically computed after optimization. To force all information items to be updated use the parameter *MSK_IPAR_AUTO_UPDATE_SOL_INFO*.

Retrieving the values

Values of information items are 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 7.3: 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);
```

7.6 Progress and data callback

Callbacks are a very useful mechanism that allow the caller to track the progress of the **MOSEK** optimizer. A callback function provided by the user is regularly called during the optimization and can be used to

- obtain a customized log of the solver execution,
- collect information for debugging purposes or
- ask the solver to terminate.

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 7.4 shows how to do it by handling the callback code *MSK_CALLBACK_NEW_INT_MIO*.

7.6.1 Data callback

In the data callback **MOSEK** passes a callback code and values of all information items to a user-defined function. The callback function is called, in particular, at the beginning of each iteration of the interior-point optimizer. For the simplex optimizers *MSK_IPAR_LOG_SIM_FREQ* controls how frequently the call-back is called.

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.

7.6.2 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 7.4: 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:
```

```
      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 7.5: 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));

MSK_putcallbackfunc(task,
                    usercallback,
                    (void *) &data);
```

7.7 MOSEK OptServer

MOSEK provides an easy way to offload optimization problem to a remote server in both *synchronous* or *asynchronous* mode. This section describes 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.

7.7.1 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 and requires very few modifications to existing code: instead of *MSK_optimize* the user must invoke *MSK_optimizermt* with the host and port where the server is running and listening as additional arguments. 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 7.6: 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)
  {
    printf ("Missing argument, syntax is:\n");
    printf ("  opt_server_sync inputfile host port\n");
  }
  else
  {
    // 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.
    // 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, argv[1]);

    // Solve the problem remotely
    if ( res == MSK_RES_OK )
      res = MSK_optimizermt (task, argv[2], argv[3], &trm);

    // Print a summary of the solution.
    if ( res == MSK_RES_OK )
      res = MSK_solutionsummary (task, MSK_STREAM_LOG);

    // If an output file was specified, write a solution
    if ( res == MSK_RES_OK && argc >= 3 )
    {
      // We define the output format to be OPF, and tell MOSEK to
      // leave out parameters and problem data from the output file.
      MSK_putintparam (task, MSK_IPAR_WRITE_DATA_FORMAT,    MSK_DATA_FORMAT_OP);
      MSK_putintparam (task, MSK_IPAR_OPF_WRITE_SOLUTIONS,  MSK_ON);
      MSK_putintparam (task, MSK_IPAR_OPF_WRITE_HINTS,      MSK_OFF);
      MSK_putintparam (task, MSK_IPAR_OPF_WRITE_PARAMETERS, MSK_OFF);
      MSK_putintparam (task, MSK_IPAR_OPF_WRITE_PROBLEM,    MSK_OFF);

      res = MSK_writedata (task, argv[2]);
    }

    // Delete task and environment
    MSK_deletetask (&task);
    MSK_deleteenv (&env);
  }
  return res;
}
```

7.7.2 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 7.7: 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 * host     = "karise";
  const char * port     = "30080";

  if (argc < 5)
  {
    fprintf(stderr, "Syntax: opt_server_async filename host port numpolls\n");
    return 0;
  }

  if (argc > 1) filename = argv[1];
  if (argc > 2) host     = argv[2];
  if (argc > 2) port     = argv[3];
  if (argc > 4) numpolls = atoi(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);

  if ( res == MSK_RES_OK )
    res = MSK_readdata (task, filename);

  res = MSK_asyncoptimize(task,
                          host,
```

```
                          port,
                          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 ( 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

    printf("poll %d\n ", i);


    res = MSK_asyncpoll( task,
                         host,
                         port,
                         token,
                         &respavailable,
                         &resp,
                         &trm);

    puts("polling done");

    if (respavailable)
    {
      puts("solution available!");
      res = MSK_asyncgetresult(task,
                               host,
                               port,
                               token,
                               &respavailable,
                               &resp,
                               &trm);

      MSK_solutionsummary (task, MSK_STREAM_LOG);
      break;
    }

  }


  if (i == numpolls)
  {
    printf("max num polls reached, stopping %s", host);
    MSK_asyncstop (task, host, port, token);
  }

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

  printf("%s:%d: Result = %d\n", __FILE__, __LINE__, res); fflush(stdout);
```

```
  return res;
}
```

CHAPTER

EIGHT

NONLINEAR TUTORIALS

This chapter provides information about how to solve general convex nonlinear optimization problems using **MOSEK**. By general nonlinear problems we mean those that cannot be formulated in conic or convex quadratically constrained form.

In general we recommend not to use the general nonlinear optimizer unless absolutely necessary. The reasons are:

- The algorithm employed for nonlinear optimization problems is not as efficient as the one employed for conic problems. Conic problems have special structure that can be exploited to make the optimizer faster and more robust.
- **MOSEK** has no way of checking whether the formulated problem is convex and if this assumption is not satisfied the optimizer will not work.
- The nonlinear optimizer requires 1st and 2nd order derivative information which is often hard to provide correctly.
- The specification of nonlinear problems requires C function callbacks, which cannot be dumped to disk and make issue reporting harder.

Instead, we advise:

- Consider reformulating the problem to a conic quadratic optimization problem if at all possible. In particular many problems involving polynomial terms can easily be reformulated to conic quadratic form.
- Consider reformulating the problem to a separable optimization problem because that simplifies the issue with verifying convexity and computing 1st and 2nd order derivatives significantly. In most cases problems in separable form also solve faster because of the simpler structure of the functions.
- Finally, if the problem cannot be reformulated in separable form use a modelling language like AMPL or GAMS, which will perform all the preprocessing, computing function values and derivatives. This eliminates an important source of errors. Therefore, it is strongly recommended to use a modelling language at the prototype stage.

The Optimizer API for C provides the following nonlinear interfaces:

8.1 Separable Convex (SCopt) Interface

The Optimizer API for C provides a way to add simple non-linear functions composed from a limited set of non-linear terms. Non-linear terms can be mixed with quadratic terms in objective and constraints. We consider problems which can be formulated as:

$$
\begin{array}{lccccl}
\text{minimize} & & & z_0(x) + c^T x & & \\
\text{subject to} & l_i^c & \leq & z_i(x) + a_i^T x & \leq & u_i^c \quad i = 1 \ldots m \\
 & l^x & \leq & x & \leq & u^x,
\end{array}
$$

where $x \in \mathbb{R}^n$ and each $z_i : \mathbb{R}^n \to \mathbb{R}$ is separable, that is can be written as a sum

$$z_i(x) = \sum_{j=1}^{n} z_{i,j}(x_j).$$

The interface implements a limited set of functions which can appear as $z_{i,j}$. They are:

Table 8.1: Functions supported by the SCopt interface.

Separable function	Operator name	Name
$fx\ln(x)$	*ent*	Entropy function
fe^{gx+h}	*exp*	Exponential function
$f\ln(gx+h)$	*log*	Logarithm
$f(x+h)^g$	*pow*	Power function

where $f, g, h \in \mathbb{R}$ are constants. This formulation does not guarantee convexity. For **MOSEK** to be able to solve the problem, the following requirements must be met:

- If the objective is minimized, the sum of non-linear terms must be convex, otherwise it must be concave.
- Any constraint bounded below must be concave, and any constraint bounded above must be convex.
- Each separable term must be twice differentiable within the bounds of the variable it is applied to.

Some simple rules can be followed to ensure that the problem satisfies **MOSEK**'s convexity and differentiability requirements. First of all, for any variable x_i used in a separable term, the variable bounds must define a range within which the function is twice differentiable. These bounds are defined in Table 8.2.

Table 8.2: Safe bounds for functions in the SCopt interface.

Separable function	Operator name	Safe x bounds
$fx\ln(x)$	*ent*	$0 < x$.
fe^{gx+h}	*exp*	$-\infty < x < \infty$.
$f\ln(gx+h)$	*log*	If $g > 0$: $-h/g < x$.
		If $g < 0$: $x < -h/g$.
$f(x+h)^g$	*pow*	If $g > 0$ and integer: $-\infty < x < \infty$.
		If $g < 0$ and integer: either $-h < x$ or $x < -h$.
		Otherwise: $-h < x$.

To ensure convexity, we require that each $z_i(x)$ is either a sum of convex terms or a sum of concave terms. Table 8.3 lists convexity conditions for the relevant ranges for $f > 0$ — changing the sign of f switches concavity/convexity.

Table 8.3: Convexity conditions for functions in the SCopt interface.

Separable function	Operator name	Convexity conditions
$fx\ln(x)$	*ent*	Convex within safe bounds.
fe^{gx+h}	*exp*	Convex for all x.
$f\ln(gx+h)$	*log*	Concave within safe bounds.
$f(x+h)^g$	*pow*	If g is even integer: convex within safe bounds.
		If g is odd integer: • concave if $(-\infty, -h)$, • convex if $(-h, \infty)$ If $0 < g < 1$: concave within safe bounds.
		Otherwise: convex within safe bounds.

A problem involving linear combinations of variables (such as $\ln(x_1+x_2)$), can be converted to a separable problem using slack variables and additional equality constraints.

8.1.1 Example

Consider the following separable convex problem:

$$\begin{array}{ll} \text{minimize} & x_1 - \ln(x_3) \\ \text{subject to} & x_1^2 + x_2^2 \leq 1 \\ & x_1 + 2x_2 - x_3 = 0 \\ & x_3 \geq 0 \end{array} \tag{8.1}$$

Note that all nonlinear functions are well defined for x values satisfying the variable bounds strictly. This assures that function evaluation errors will not occur during the optimization process because **MOSEK**.

The linear part of the problem is specified as usually. The nonlinear part is set using the function *MSK_scbegin*. See the *API reference* for a description of the format. After that a standard invocation of *MSK_optimize* solves the problem. The *API reference* describes additional functions for reading and writing SCopt terms from/to a file.

Note that the code must include the extension `scopt-ext.h` and must be linked the implementation contained in `scopt-ext.c`, both available in `examples/c`.

Listing 8.1: Implementation of problem (8.1).

```c
#include "scopt-ext.h"

#define NUMOPRO  1 /* Number of nonlinear expressions in the obj. */
#define NUMOPRC  2 /* Number of nonlinear expressions in the con. */
#define NUMVAR   3 /* Number of variables.     */
#define NUMCON   2 /* Number of constraints.   */
#define NUMANZ   3 /* Number of non-zeros in A. */

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

int main()
{
  char          buffer[MSK_MAX_STR_LEN];
  double        oprfo[NUMOPRO],oprgo[NUMOPRO],oprho[NUMOPRO],
                oprfc[NUMOPRC],oprgc[NUMOPRC],oprhc[NUMOPRC],
                c[NUMVAR],aval[NUMANZ],
                blc[NUMCON],buc[NUMCON],blx[NUMVAR],bux[NUMVAR];
  int           numopro,numoprc,
                numcon=NUMCON,numvar=NUMVAR,
                opro[NUMOPRO],oprjo[NUMOPRO],
                oprc[NUMOPRC],opric[NUMOPRC],oprjc[NUMOPRC],
                aptrb[NUMVAR],aptre[NUMVAR],asub[NUMANZ];
  MSKboundkeye bkc[NUMCON],bkx[NUMVAR];
  MSKenv_t      env;
  MSKrescodee   r;
  MSKtask_t     task;
  schand_t      sch;

  /* Specify nonlinear terms in the objective. */
  numopro  = NUMOPRO;
  opro[0]  = MSK_OPR_LOG; /* Defined in scopt.h */
  oprjo[0] = 2;
  oprfo[0] = -1.0;
```

```
  oprgo[0] = 1.0;  /* This value is never used. */
  oprho[0] = 0.0;

  /* Specify nonlinear terms in the constraints. */
  numoprc  = NUMOPRC;

  oprc[0]  = MSK_OPR_POW;
  opric[0] = 0;
  oprjc[0] = 0;
  oprfc[0] = 1.0;
  oprgc[0] = 2.0;
  oprhc[0] = 0.0;

  oprc[1]  = MSK_OPR_POW;
  opric[1] = 0;
  oprjc[1] = 1;
  oprfc[1] = 1.0;
  oprgc[1] = 2.0;
  oprhc[1] = 0.0;

  /* Specify c */
  c[0] = 1.0; c[1] = 0.0; c[2] = 0.0;

  /* Specify a. */
  aptrb[0] = 0;   aptrb[1] = 1;   aptrb[2] = 2;
  aptre[0] = 1;   aptre[1] = 2;   aptre[2] = 3;
  asub[0]  = 1;   asub[1]  = 1;   asub[2]  = 1;
  aval[0]  = 1.0; aval[1]  = 2.0; aval[2]  = -1.0;

  /* Specify bounds for constraints. */
  bkc[0] = MSK_BK_UP;     bkc[1] = MSK_BK_FX;
  blc[0] = -MSK_INFINITY; blc[1] = 0.0;
  buc[0] = 1.0;           buc[1] = 0.0;

  /* Specify bounds for variables. */
  bkx[0] = MSK_BK_LO;      bkx[1] = MSK_BK_LO;      bkx[2] = MSK_BK_LO;
  blx[0] = 0.0;            blx[1] = 0.1;            blx[2] = 0.0;
  bux[0] = MSK_INFINITY;   bux[1] = MSK_INFINITY;   bux[2] = MSK_INFINITY;

  /* Create  the 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 )
      MSK_linkfunctotaskstream(task,MSK_STREAM_LOG,NULL,printstr);

    if ( r==MSK_RES_OK )
    {
      /* Setup the linear part of the problem. */
      r = MSK_inputdata(task,
                        numcon,numvar,
                        numcon,numvar,
                        c,0.0,
                        aptrb,aptre,
                        asub,aval,
                        bkc,blc,buc,
                        bkx,blx,bux);
    }

    if ( r== MSK_RES_OK )
```

```
    {
      /* Set-up of nonlinear expressions. */
      r = MSK_scbegin(task,
                      numopro,opro,oprjo,oprfo,oprgo,oprho,
                      numoprc,oprc,opric,oprjc,oprfc,oprgc,oprhc,
                      &sch);

      if ( r==MSK_RES_OK )
      {
        printf("Start optimizing\n");

        r = MSK_optimize(task);

        printf("Done optimizing\n");

        MSK_solutionsummary(task,MSK_STREAM_MSG);
      }

      /* The nonlinear expressions are no longer needed. */
      MSK_scend(task,&sch);
    }
    MSK_deletetask(&task);
  }
  MSK_deleteenv(&env);

  printf("Return code: %d\n",r);
  if ( r!=MSK_RES_OK )
  {
    MSK_getcodedesc(r,buffer,NULL);
    printf("Description: %s\n",buffer);
  }

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

8.2 Exponential Optimization

8.2.1 Problem Definition

An exponential optimization problem has the form

$$
\begin{array}{lrcl}
\text{minimize} & \sum_{k\in J_0} c_k e^{\left\{\sum_{j=0}^{n-1} a_{k,j} x_j\right\}} & & \\
\text{subject to} & \sum_{k\in J_i} c_k e^{\left\{\sum_{j=0}^{n-1} a_{k,j} x_j\right\}} & \leq & 1, \quad i=1,\ldots,m, \\
 & x \in \mathbb{R}^n & &
\end{array}
\tag{8.2}
$$

where it is assumed that

$$\bigcup_{i=0}^{m} J_k = \{1,\ldots,T\}$$

and

$$J_i \cap J_j = \emptyset$$

if $i \neq j$.

Given

$$c_i > 0, \quad i = 1,\ldots,T$$

the problem (8.2) is a convex optimization problem which can be solved using **MOSEK**. We will call

$$c_t e^{\left\{\sum_{j=0}^{n-1} a_{t,j} x_j\right\}} = e^{\left\{\log(c_t)+\sum_{j=0}^{n-1} a_{t,j} x_j\right\}}$$

a single term and hence the number of terms is T.

As stated the problem (8.2) is a nonseparable problem. However, using

$$v_t = \log(c_t) + \sum_{j=0}^{n-1} a_{tj} x_j$$

we obtain the separable problem

$$\begin{array}{lrcll} \text{minimize} & \sum_{t\in J_0} e^{v_t} & & & \\ \text{subject to} & \sum_{t\in J_i} e^{v_t} & \leq & 1, & i = 1,\ldots,m, \\ & \sum_{j=0}^{n-1} a_{t,j} x_j - v_t & = & -\log(c_t), & t = 0,\ldots,T. \end{array} \tag{8.3}$$

A warning about this approach is that computing the function

$$e^x$$

using double-precision floating point numbers is only possible for x of small absolute value. It is also possible to reformulate the exponential optimization problem (8.2) as a dual geometric geometric optimization problem, see Sec. 8.3. This is often the preferred solution approach because it is computationally more efficient and the numerical problems associated with evaluating e^x for moderately large x values are avoided.

Moreover, exponential optimization problems may in some cases have an optimal solution involving infinite values. Consider the simple example

$$\begin{array}{rl} \text{minimize} & e^x \\ \text{subject to} & x \in \mathbb{R}, \end{array}$$

which has the optimal objective value 0 at $x = -\infty$. Similar problems can occur in constraints. Such a solution can not in general be obtained by numerical methods, which means that **MOSEK** will act unpredictably in these situations — possibly failing to find a meaningful solution or simply stalling.

8.2.2 The Command Line tool

In the following we will discuss the program `mskexpopt`, which is included in the **MOSEK** distribution together with its source code. Hence, you can solve exponential optimization problems using the operating system command line or directly from your own C program. The interface enables:

- Reading and writing a data file with an exponential optimization problem.
- Verifying that the input data is reasonable.
- Solving the problem in primal or dual form.
- Writing a solution file.

The Input Format

The program can read a description of an exponential problem from a file in the following format:

$$
\begin{array}{lll}
*\mathtt{Thisisacomment} & & \\
\mathtt{numcon} & & \\
\mathtt{numvar} & & \\
\mathtt{numter} & & \\
c_1 & & \\
c_2 & & \\
\vdots & & \\
c_{\mathtt{numter}} & & \\
i_1 & & \\
i_2 & & \\
\vdots & & \\
i_{\mathtt{numter}} & & \\
t_1 & j_1 & a_{\mathtt{t_1,j_1}} \\
t_2 & j2 & a_{\mathtt{t_2,j_2}} \\
\vdots & \vdots & \vdots
\end{array}
$$

The first line is an optional comment line. In general everything occurring after a `*` is considered a comment. Lines 2 to 4 inclusive define the number of constraints (m), the number of variables (n), and the number of terms T in the problem. Then follows three sections containing the problem data.

The first section

$$
\begin{array}{l}
c_1 \\
c_2 \\
\vdots \\
c_{\mathtt{numter}}
\end{array}
$$

lists the coefficients c_t of each term t in their natural order.

The second section

$$
\begin{array}{l}
i_1 \\
i_2 \\
\vdots \\
i_{\mathtt{numter}}
\end{array}
$$

specifies to which constraint each term belongs. Hence, for instance $i_2 = 5$ means that the term number 2 belongs to constraint 5. $i_t = 0$ means that term number t belongs to the objective.

The third section

$$
\begin{array}{lll}
t_1 & j_1 & a_{t_1,j_1} \\
t_2 & j_2 & a_{t_2,j_2} \\
\vdots & \vdots & \vdots
\end{array}
$$

defines the non-zero $a_{t,j}$ values. For instance the entry

$$
1 \quad 3 \quad 3.3
$$

implies that $a_{t,j} = 3.3$ for $t = 1$ and $j = 3$.

Please note that each $a_{t,j}$ should be specified only once.

Choosing Primal or Dual Form

One can choose to solve the exponential optimization problem directly in the primal form (8.3) or in the dual form. By default `mskexpopt` solves a problem in the dual form since usually this is more efficient. The command line option

```
-primal
```

chooses the primal form.

An Example

Consider the problem:

$$\begin{array}{lccc} \text{minimize} & 40e^{-x_1-\frac{1}{2}x_2-x_3}+20e^{x_1+x_3}+40e^{x_1+x_2+x_3} & & \\ \text{subject to} & \frac{1}{3}e^{-2x_1-2x_2}+\frac{4}{3}e^{\frac{1}{2}x_2-x_3} & \leq & 1. \end{array} \tag{8.4}$$

This small problem can be specified using the input format shown in Listing 8.2.

Listing 8.2: Input file to specify problem (8.4).

```
* File : expopt1.eo

1   * numcon
3   * numvar
5   * numter

* Coefficients of terms

40
20
40
0.3333333
1.3333333

* For each term, the index of the
* constraints to the term belongs

0
0
0
1
1

* Section defining a_kj

0 0 -1
0 1 -0.5
0 2 -1
1 0 1.0
1 2 1.0
2 0 1.0
2 1 1.0
2 2 1.0
3 0 -2
3 1 -2
4 1 0.5
4 2 -1.0
```

Now the command `mskexpopt expopt1.eo` will produce the solution file `expopt1.sol` shown below.

```
PROBLEM STATUS      : PRIMAL_AND_DUAL_FEASIBLE
SOLUTION STATUS     : OPTIMAL
PRIMAL OBJECTIVE    : 1.331371e+02

VARIABLES
INDEX   ACTIVITY
1       6.931471e-01
2       -6.931472e-01
3       3.465736e-01
```

8.2.3 The C interface

The C source code for solving an exponential optimization problem is included in `expopt.h` and `expopt.c`. Setting up an exponential problem begins with a call to *MSK_expoptsetup*, which provides a description of the problem in the form (8.2). The problem can then be solved using *MSK_expoptimize*. For details consult the *API reference* and the source file `examples/c/mskexpopt.c`.

An example that solves (8.4) is included below.

```c
#include <string.h>

#include "expopt.h"

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


int main (int argc, char **argv)
{
  int          r = MSK_RES_OK, numcon = 1, numvar = 3 , numter = 5;

  int          subi[]   = {0,0,0,1,1};
  int          subk[]   = {0,0,0,1,1,2,2,2,3,3,4,4};
  double       c[]      = {40.0,20.0,40.0,0.333333,1.333333};
  int          subj[]   = {0,1,2,0,2,0,1,2,0,1,1,2};
  double       akj[]    = {-1,-0.5,-1.0,1.0,1.0,1.0,1.0,1.0,-2.0,-2.0,0.5,-1.0};
  int          numanz   = 12;
  double       objval;
  double       xx[3];
  double       y[5];
  MSKenv_t     env;
  MSKprostae   prosta;
  MSKsolstae   solsta;
  MSKtask_t    expopttask;
  expopthand_t expopthnd = NULL;
  /* Pointer to data structure that holds nonlinear information */

  if (r == MSK_RES_OK)
    r = MSK_makeenv (&env,NULL);

  if (r == MSK_RES_OK)
    MSK_makeemptytask(env,&expopttask);

  if (r == MSK_RES_OK)
    r = MSK_linkfunctotaskstream(expopttask,MSK_STREAM_LOG,NULL,printcb);

  if (r == MSK_RES_OK)
  {
    /* Initialize expopttask with problem data */
```

```
    r =  MSK_expoptsetup(expopttask,
                         1, /* Solve the dual formulation */
                         numcon,
                         numvar,
                         numter,
                         subi,
                         c,
                         subk,
                         subj,
                         akj,
                         numanz,
                         &expopthnd
                         /* Pointer to data structure holding nonlinear data */
                         );
  }

  /* Any parameter can now be changed with standard mosek function calls */
  if (r == MSK_RES_OK)
    r = MSK_putintparam(expopttask,MSK_IPAR_INTPNT_MAX_ITERATIONS,200);

  /* Optimize, xx holds the primal optimal solution,
   y holds solution to the dual problem if the dual formulation is used
  */

  if (r == MSK_RES_OK)
    r = MSK_expoptimize(expopttask,
                        &prosta,
                        &solsta,
                        &objval,
                        xx,
                        y,
                        &expopthnd);

  /* Free data allocated by expoptsetup */
  if (expopthnd)
    MSK_expoptfree(expopttask,
                   &expopthnd);

  MSK_deletetask(&expopttask);
  MSK_deleteenv(&env);

}
```

8.3 Dual Geometric Optimization

Dual geometric is a special class of nonlinear optimization problems involving a nonlinear and non-separable objective function. In this section we will show how to solve dual geometric optimization problems using **MOSEK**. For a thorough discussion of geometric optimization see *[BSS93]*.

8.3.1 Problem Definition

Consider the dual geometric optimization problem

$$
\begin{array}{lccc}
\text{maximize} & f(x) & & \\
\text{subject to} & Ax & = & b, \\
 & x \geq 0, & &
\end{array}
$$

where $A \in \mathbb{R}^{m \times n}$ and all other quantities have conforming dimensions. Let t be an integer and p be a vector of $t+1$ integers satisfying the conditions

$$\begin{array}{rcl} p_0 & = & 0, \\ p_i & < & p_{i+1}, \quad i = 1, \ldots, t, \\ p_t & = & n. \end{array}$$

Then f can be of the form

$$f(x) = \sum_{j=0}^{n-1} x_j \ln\left(\frac{v_j}{x_j}\right) + \sum_{i=1}^{t}\left(\sum_{j=p_i}^{p_{i+1}-1} x_j\right) \ln\left(\sum_{j=p_i}^{p_{i+1}-1} x_j\right)$$

where $v \in \mathbb{R}^n_+$. Given these assumptions, it can be proven that f is a concave function and therefore the dual geometric optimization problem can be solved using **MOSEK**. We will introduce the following definitions:

$$x^i := \begin{bmatrix} x_{p_i} \\ x_{p_i+1} \\ \vdots \\ x_{p_{i+1}-1} \end{bmatrix}, X^i := \operatorname{diag}(x^i), \text{ and } e^i := \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^{p_{i+1}-p_i}.$$

which make it possible to state f on the form

$$f(x) = \sum_{j=0}^{n-1} x_j \ln\left(\frac{v_j}{x_j}\right) + \sum_{i=1}^{t}\left((e^i)^T x^i\right) \ln\left((e^i)^T x^i\right).$$

Furthermore, we have that

$$\nabla f(x) = \begin{bmatrix} \ln(v_0) - 1 - \ln(x_0) \\ \vdots \\ \ln(v_j) - 1 - \ln(x_j) \\ \vdots \\ \ln(v_{n-1}) - 1 - \ln(x_{n-1}) \end{bmatrix} + \begin{bmatrix} 0e^0 \\ (1 + \ln((e^1)^T x^1))e^1 \\ \vdots \\ (1 + \ln((e^i)^T x^i))e^i \\ \vdots \\ (1 + \ln((e^t)^T x^t))e^t \end{bmatrix}$$

and

$$\nabla^2 f(x) = \begin{bmatrix} -(X^0)^{-1} & 0 & 0 & \cdots & 0 \\ 0 & \frac{e^1(e^1)^T}{(e^1)^T x^1} - (X^1)^{-1} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \frac{e^t(e^t)^T}{(e^t)^T x^t} - (X^t)^{-1} \end{bmatrix}.$$

Please note that the Hessian is a block diagonal matrix and, especially if t is large, it is very sparse — **MOSEK** will automatically exploit these features to speed up computations. Moreover, the Hessian can be computed cheaply, specifically in

$$O\left(\sum_{i=0}^{t}(p_{i+1} - p_i)^2\right)$$

operations.

8.3.2 `dgopt`: A Program for Dual Geometric Optimization

Input format

A dual geometric optimization problem input consists of two files. Since the constraints of the optimization problem are linear, they can be specified using an MPS file as in the purely linear case. The objective f is defined in a separate file as a list of the following values:

$$\begin{array}{c} t \\ v_0 \\ v_1 \\ \vdots \\ v_{n-1} \\ p_1 - p_0 \\ p_2 - p_1 \\ \vdots \\ p_t - p_{t-1} \end{array}$$

For example, the function f given by

$$\begin{aligned} f(x) = \quad & x_0 \ln\left(\frac{40}{x_0}\right) + x_1 \ln\left(\frac{20}{x_1}\right) + x_2 \ln\left(\frac{40}{x_2}\right) + x_3 \ln\left(\frac{1}{3x_3}\right) + x_4 \ln\left(\frac{4}{3x_4}\right) \\ & +(x_3 + x_4)\ln(x_3 + x_4) \end{aligned}$$

would be represented as:

Listing 8.3: File containing the specification for the non-linear part.

```
2
40.0
20.0
40.0
0.33333333333333
1.33333333333333
3
2
```

The example is solved by executing the command line

```
mskdgopt examp/data/dgo.mps examples/data/dgo.f
```

The C interface

The source code for the dual geometric optimizer consists of the files `dgopt.h` and `dgopt.c`. To define an optimization problem the user should set up an ordinary task containing the linear part of the data and call *MSK_dgosetup* to append the nonlinear objective data. After that the standard method *MSK_optimize* solves the problem. See the file `mskdgopt.c` provided in `examples/c` for more information and the *API reference* for details.

8.4 General Convex Optimization

MOSEK provides an interface for general convex optimization which is discussed in this section.

Warning: Using the general convex optimization interface in **MOSEK** is (very) complicated. It is recommended to use the conic solver, the quadratic solver or the `scopt` interface whenever possible. Alternatively GAMS or AMPL with **MOSEK** as solver are well-suited for general convex optimization problems.

8.4.1 The problem

A general nonlinear convex optimization problem is to minimize or maximize an objective function of the form

$$f(x) + \frac{1}{2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} q^o_{i,j} x_i x_j + \sum_{j=0}^{n-1} c_j x_j + c^f$$

subject to the functional constraints

$$l^c_k \leq g_k(x) + \frac{1}{2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} q^k_{i,j} x_i x_j + \sum_{j=0}^{n-1} a_{k,j} x_j \leq u^c_k, \quad k = 0, \ldots, m-1,$$

and the bounds

$$l^x_j \leq x_j \leq u^x_j, \quad j = 0, \ldots, n-1.$$

Please note that this problem is a generalization of linear and quadratic optimization. This implies that the parameters c, A, Q^o, Q, and so forth denote the same data as in the case of linear and quadratic optimization. All linear and quadratic terms should be inputted to **MOSEK** as described for these problem classes. The general convex part of the problems is defined by the functions $f(x)$ and $g_k(x)$, which must be general nonlinear, twice differentiable functions.

8.4.2 Assumptions About a Nonlinear Optimization Problem

MOSEK makes two assumptions about the optimization problem.

The first assumption is that all functions are at least twice differentiable on their domain. More precisely, $f(x)$ and $g(x)$ must be at least twice differentiable for all $l^x < x < u^x$.

The second assumption is that

$$f(x) + \frac{1}{2}x^T Q^o x$$

must be a convex function if the objective is minimized. Otherwise if the objective is maximized it must be a concave function. Moreover,

$$g_k(x) + \frac{1}{2}x^T Q^k x$$

must be a convex function if

$$u^c_k < \infty$$

and a concave function if

$$l^c_k > -\infty.$$

Note in particular that nonlinear equalities are not allowed. **If these two assumptions are not satisfied, then it cannot be guaranteed that MOSEK produces correct results or works at all**.

8.4.3 Specifying General Convex Terms

MOSEK receives information about the general convex terms via two call-back functions implemented by the user:

- *MSKnlgetspfunc*: Provides structural information about f and g.
- *MSKnlgetvafunc*: Provides numerical information about f and g.

These call-back functions are passed to **MOSEK** using *MSK_putnlfunc*. For an example of using the general convex framework see Sec. 8.3.

CHAPTER

NINE

ADVANCED NUMERICAL TUTORIALS

MOSEK provides access to numerical linear algebra tools essential for more advanced applications. They are described in this section.

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

$$[\ A \quad -I\ ].$$

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(task,basis);
```

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

$$\mathtt{basis}[i] < \mathtt{numcon},$$

then the i-th basis variable is x_i^c. Moreover, the i-th column in B will be the i-th column of $-I$. On the other hand if

$$\mathtt{basis}[i] \geq \mathtt{numcon},$$

then the i-th basis variable is the variable

$$x_{\mathtt{basis}[i]-\mathtt{numcon}}$$

and the i-th column of B is the column

$$A_{:,(\mathtt{basis}[i]-\mathtt{numcon})}.$$

For instance if $\mathtt{basis}[0] = 4$ and $\mathtt{numcon} = 5$, then since $\mathtt{basis}[0] < \mathtt{numcon}$, the first basis variable is x_4^c. Therefore, the first column of B is the fourth column of $-I$. Similarly, if $\mathtt{basis}[1] = 7$, then the second variable in the basis is $x_{\mathtt{basis}[1]-\mathtt{numcon}} = x_2$. 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

$$\begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix}.$$

Please note the ordering of the columns in B .

Listing 9.1: A program showing how to identify the basis.

```
#include "mosek.h"

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

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

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

  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);
  }
```

```
  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);

  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);

  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:

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 */


MSKrescodee put_a(MSKtask_t task,
                  double *aval,
                  MSKidxt *asub,
                  MSKidxt *ptrb,
                  MSKidxt *ptre,
                  int numvar,
                  MSKidxt *basis
                 )

{
  MSKrescodee r = MSK_RES_OK;
  int i;
  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);

/* Define a basic solution by specifying
   status keys for variables & constraints. */
for (i = 0; i < numvar && r == MSK_RES_OK; ++i)
  r = MSK_putsolutioni (
        task,
        MSK_ACC_VAR,
        i,
        MSK_SOL_BAS,
        skx[i],
        0.0,
        0.0,
        0.0,
        0.0);

for (i = 0; i < numvar && r == MSK_RES_OK; ++i)
  r = MSK_putsolutioni (
        task,
        MSK_ACC_CON,
        i,
        MSK_SOL_BAS,
        skc[i],
        0.0,
        0.0,
        0.0,
        0.0);
```

```
  if (r == MSK_RES_OK)
    r = MSK_initbasissolve(task, basis);

  free (skx);
  free (skc);

  return ( r );

}

#define NUMCON 2
#define NUMVAR 2

int main(int argc, const char *argv[])
{
  MSKenv_t  env;
  MSKtask_t task;
  MSKrescodee r = MSK_RES_OK;
  MSKintt   numvar = NUMCON;
  MSKintt   numcon = NUMVAR;   /* we must have numvar == numcon */
  int       i, nz;
  double    aval[] = { -1.0, 1.0, 1.0};
  MSKidxt   asub[] = {1, 0, 1};
  MSKidxt   ptrb[] = {0, 1};
  MSKidxt   ptre[] = {1, 3};

  MSKidxt   bsub[NUMCON];
  double    b[NUMCON];

  MSKidxt   *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 = (MSKidxt *) calloc(numcon, sizeof(MSKidxt));
  if ( basis == NULL && numvar)
    r = MSK_RES_ERR_SPACE;

  /* Put A matrix and factor A.
     Call this function only once for a given task. */
  if (r == MSK_RES_OK)
    r = put_a( task,
               aval,
               asub,
               ptrb,
               ptre,
               numvar,
               basis
             );

  /* now solve rhs */
  b[0] = 1;
  b[1] = -2;
  bsub[0] = 0;
  bsub[1] = 1;
  nz = 2;
```

```
  if (r == MSK_RES_OK)
    r = MSK_solvewithbasis(task, 0, &nz, bsub, b);

  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);

  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:

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 of a symmetric matrix
SYEIG	*MSK_syeig*	Eigenvalues and eigenvectors 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");
  }

}

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

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

```c
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);
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,          /* Disable multithreading */
                              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.

9.4 Converting a quadratically constrained problem to conic form

MOSEK employs the following form of quadratic problems:

$$\begin{array}{lccccll} \text{minimize} & & & \frac{1}{2}x^TQ^ox + c^Tx + c^f & & & \\ \text{subject to} & l_k^c & \leq & \frac{1}{2}x^TQ^kx + \sum_{j=0}^{n-1} a_{k,j}x_j & \leq & u_k^c, & k = 0, \ldots, m-1, \\ & l_j^x & \leq & x_j & \leq & u_j^x, & j = 0, \ldots, n-1. \end{array} \tag{9.8}$$

A conic quadratic constraint has the form

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

in its most basic form where

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

A quadratic problem such as (9.8), if convex, can be reformulated in conic form. This is in fact the reformulation **MOSEK** performs internally. It has many advantages:

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

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

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

In particular:

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

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

Note: `MSK_toconic` modifies the input task in-place: this means that if the reformulation is not possible, i.e. the problem is not conic representable, the state of the task is in general undefined. The user should consider cloning the original task.

9.4.1 Quadratic Constraint Reformulation

Let us assume we want to convert the following quadratic constraint

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

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

$$\frac{1}{2}x^TQx + \sum_{j=0}^{n-1} a_j^T x_j \leq u. \tag{9.9}$$

Introducing an additional variable w such that

$$w = u - \sum_{j=0}^{n-1} a_j^T x_j \tag{9.10}$$

we obtain the equivalent form

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

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

$$Q = FF^T \tag{9.11}$$

and therefore we can write

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

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

$$\begin{array}{ll} (w, z, y) & \in \mathcal{Q}_r, \\ z = 1 & \\ y = Fx & \\ w = u - a^T x. & \end{array} \tag{9.12}$$

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

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

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

Example

Next we consider a simple problem with quadratic objective function:

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

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

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

[variables]
x0 x1 x2
[/variables]

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

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

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

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

The model generated by *MSK_toconic* is

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

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

[variables disallow_new_variables]
  x0000_x0 x0001_x1 x0002_x2 x0003 x0004
  x0005 x0006 x0007 x0008
[/variables]

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

[constraints]
  [con c0000]  3.605551275463989e+00 x0000_x0 - 5.547001962252291e-01 x0002_x2 + 3.
↪328201177351375e+00 x0001_x1 - x0006 = 0e+00 [/con]
  [con c0001]  3.419401657060442e+00 x0002_x2 + 2.294598480395823e+00 x0001_x1 - x0007 = 0e+00␣
↪[/con]
```

```
  [con c0002]  8.111071056538127e-01 x0001_x1 - x0008 = 0e+00 [/con]
  [con c0003]  - x0003 + x0004 = 0e+00 [/con]
[/constraints]

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

We can clearly see that constraints c0000, c0001 and c0002 represent the original linear constraints as in (9.11), while c0003 corresponds to (9.10). The cone roots are x0005 and x0004.

CHAPTER

TEN

TECHNICAL GUIDELINES

This section contains some technical guidelines for the Optimizer API for C users.

For modelling guidelines check one of the following sections:

- Sec. 13 for how to address numerical issues in modelling and how to tune the continuous optimizers.
- Sec. 14 for how to tune the mixed-integer optimizer.

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 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. See Sec. 13 and Sec. 14 for more details for the two optimizer types.

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.

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.

10.3 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_putmaxnumcone*. Estimate for the number of cones.
- *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.

10.4 The license system

MOSEK is a commercial product that **always** needs a valid license to work. **MOSEK** uses a third party license manager to implement license checking. The number of license tokens provided determines the number of optimizations that can be run simultaneously.

By default a license token remains checked out from the first optimization until the end of the **MOSEK** session, i.e.

- a license token is checked out when *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**.

10.5 Deployment

When redistributing a C application using the **MOSEK** Optimizer API for C 8.1.0.56, the following libraries must be included:

64-bit Linux	64-bit Windows	32-bit Windows	64-bit Mac OS
libmosek64.so.8.1	mosek64_8_1.dll	mosek8_1.dll	libmosek64.8.1.dylib
libiomp5.so	libomp5md.dll	libomp5md.dll	
libcilkrts.so.5	cilkrts20.dll	cilkrts20.dll	libcilkrts.5.dylib

CHAPTER

ELEVEN

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 modelling 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.

Case Studies	Type	Int.	Keywords
Portofolio Optimization	CQO	NO	Markowitz, Slippage, Market Impact

11.1 Portfolio Optimization

In this section the Markowitz portfolio optimization problem and variants are implemented using the **MOSEK** optimizer API.

11.1.1 A Basic Portfolio Optimization 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 (or risk)

$$(y - \mathbf{E}y)^2 = x^T \Sigma x.$$

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, or the risk, is bounded by γ^2. Therefore, γ specifies an upper bound of the standard deviation the investor is willing to undertake. Finally, the constraint

$$x_j \geq 0$$

excludes the possibility of short-selling. This constraint can of course be excluded if short-selling is allowed.

The covariance matrix Σ is positive semidefinite by definition and therefore there exist a matrix G such that

$$\Sigma = GG^T. \tag{11.2}$$

In general the choice of G is **not** unique and one possible choice of G is the Cholesky factorization of Σ. However, in many cases another choice is better for efficiency reasons as discussed in Sec. 11.1.3.

For a given G we have that

$$\begin{array}{rcl} x^T \Sigma x & = & x^T GG^T x \\ & = & \left\|G^T x\right\|^2. \end{array}$$

Hence, we may write the risk constraint as

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

or equivalently

$$[\gamma; G^T x] \in \mathcal{Q}^{n+1}.$$

where $\mathcal{Q}^{n+1}$ is the $n+1$ dimensional quadratic cone. Therefore, problem (11.1) can be written as

$$\begin{array}{lrcl} \text{maximize} & \mu^T x & & \\ \text{subject to} & e^T x & = & w + e^T x^0, \\ & [\gamma; G^T x] & \in & \mathcal{Q}^{n+1}, \\ & x & \geq & 0, \end{array} \tag{11.3}$$

which is a conic quadratic optimization problem that can easily be solved using **MOSEK**.

Example data

Subsequently we will use the following sample input taken from *[CT07]*. We set

$$\mu = \begin{bmatrix} 0.1073 \\ 0.0737 \\ 0.0627 \end{bmatrix}$$

and

$$\Sigma = 0.1 \begin{bmatrix} 0.2778 & 0.0387 & 0.0021 \\ 0.0387 & 0.1112 & -0.0020 \\ 0.0021 & -0.0020 & 0.0115 \end{bmatrix}$$

This implies

$$G^T = \sqrt{0.1} \begin{bmatrix} 0.5271 & 0.0734 & 0.0040 \\ 0 & 0.3253 & -0.0070 \\ 0 & 0 & 0.1069 \end{bmatrix}$$

using 5 significant digits. Moreover, let

$$x^0 = \begin{bmatrix} 0.0 \\ 0.0 \\ 0.0 \end{bmatrix}$$

and

$$w = 1.0.$$

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.

Implementing the Portfolio Model

Creating a matrix formulation

The Optimizer API for C requires that an optimization problem is entered in the following standard form:

$$\begin{array}{lccccc} \text{maximize} & & & c^T \hat{x} & & \\ \text{subject to} & l^c & \leq & A\hat{x} & \leq & u^c, \\ & l^x & \leq & \hat{x} & \leq & u^x, \\ & & & \hat{x} \in \mathcal{K}. & & \end{array} \tag{11.4}$$

We refer to $\hat{x}$ as the API variable. It means we need to reformulate (11.3). The first step is to introduce auxiliary variables so that the conic constraint involves only unique variables:

$$
\begin{array}{lrcl}
\text{maximize} & \mu^T x & & \\
\text{subject to} & e^T x & = & w + e^T x^0, \\
& G^T x - t & = & 0, \\
& [s; t] & \in & \mathcal{Q}^{n+1}, \\
& x & \geq & 0, \\
& s & = & \gamma.
\end{array} \tag{11.5}
$$

Here s is an additional scalar variable and t is a vector variable of dimension n. The next step is to concatenate all the variables into one long variable vector:

$$
\hat{x} = [x; s; t] = \left[\begin{array}{c} x \\ s \\ t \end{array} \right] \tag{11.6}
$$

The details of the concatenation are specified below.

Table 11.1: Storage layout of the $\hat{x}$ variable.

Variable	Length	Offset
x	n	0
s	1	n
t	n	$n+1$

The offset determines where the variable starts. (Note that all variables are indexed from 0). For instance

$$
\hat{x}_{n+1+i} = t_i.
$$

because the offset of the t variable is $n+1$.

Given the ordering of the variables specified by (11.6) it is useful to visualize the linear constraints (11.4) in an explicit block matrix form:

$$
\left[\begin{array}{c|c|ccc}
1 & 0 & & 0 & \\
\hline
 & & -1 & & \\
G^T & 0 & & -1 & \\
 & & & & -1
\end{array} \right]
\cdot
\left[\begin{array}{c} x \\ \hline s \\ \hline t \end{array} \right]
=
\left[\begin{array}{c} w + e^T x_0 \\ \hline 0 \end{array} \right]. \tag{11.7}
$$

In other words, we should define the specific components of the problem description as follows:

$$
\begin{array}{rcl}
c & = & \left[\begin{array}{ccc} \mu^T & 0 & 0_n \end{array} \right]^T, \\
A & = & \left[\begin{array}{ccc} e^T & 0 & 0_n \\ G^T & 0_n & -I_n \end{array} \right], \\
l^c & = & \left[\begin{array}{cc} w + e^T x^0 & 0_n \end{array} \right]^T, \\
u^c & = & \left[\begin{array}{cc} w + e^T x^0 & 0_n \end{array} \right]^T, \\
l^x & = & \left[\begin{array}{ccc} 0_n & \gamma & -\infty_n \end{array} \right]^T, \\
u^x & = & \left[\begin{array}{ccc} \infty_n & \gamma & \infty_n \end{array} \right]^T.
\end{array} \tag{11.8}
$$

Source code example

From the block matrix form (11.7) and the explicit specification (11.8), using the offset information in Table 11.1 it is easy to calculate the index and value of each entry of the linear constraint matrix. The code below sets up the general optimization problem (11.3) and solves it for the example data. Of course it is only necessary to set non-zero entries of the linear constraint matrix.

Listing 11.1: Code implementing model (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       = 3;
  const MSKrealt  gamma   = 0.05;
  const MSKrealt  mu[]    = {0.1073,  0.0737,  0.0627};
  const MSKrealt  GT[][3] = {{0.1667,  0.0232,  0.0013},
                             {0.0000,  0.1033, -0.0022},
                             {0.0000,  0.0000,  0.0338}};
  const MSKrealt  x0[3]   = {0.0, 0.0, 0.0};
  const MSKrealt  w       = 1.0;
  MSKrealt        rtemp;
  MSKenv_t        env;
  MSKint32t       k, i, j, offsetx, offsets, offsett, *sub;
  MSKrescodee     res = MSK_RES_OK;
  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));

  /* Total budget */
  rtemp = w;
  for (j = 0; j < n; ++j)
    rtemp += x0[j];

  /* Constraints. */
  MOSEKCALL(res, MSK_appendcons(task, 1 + n));

  /* Total budget constraint - set bounds l^c = u^c */
  MOSEKCALL(res, MSK_putconbound(task, 0, MSK_BK_FX, rtemp, rtemp));
  sprintf(buf, "%s", "budget");
  MOSEKCALL(res, MSK_putconname(task, 0, buf));

  /* The remaining constraints GT * x - t = 0 - set bounds l^c = u^c*/
  for (i = 0; i < n; ++i)
  {
    MOSEKCALL(res, MSK_putconbound(task, 1 + i, MSK_BK_FX, 0.0, 0.0));
    sprintf(buf, "GT[%d]", 1 + i);
    MOSEKCALL(res, MSK_putconname(task, 1 + i, buf));
```

```
}

/* Variables. */
MOSEKCALL(res, MSK_appendvars(task, 1 + 2 * n));

offsetx = 0;   /* Offset of variable x into the API variable. */
offsets = n;   /* Offset of variable x into the API variable. */
offsett = n + 1; /* Offset of variable t into the API variable. */

/* x variables. */
for (j = 0; j < n; ++j)
{
  /* Return of asset j in the objective */
  MOSEKCALL(res, MSK_putcj(task, offsetx + j, mu[j]));
  /* Coefficients in the first row of A */
  MOSEKCALL(res, MSK_putaij(task, 0, offsetx + j, 1.0));
  /* No short-selling - x^l = 0, x^u = inf */
  MOSEKCALL(res, MSK_putvarbound(task, offsetx + j, MSK_BK_LO, 0.0, MSK_INFINITY));
  sprintf(buf, "x[%d]", 1 + j);
  MOSEKCALL(res, MSK_putvarname(task, offsetx + j, buf));
}

/* s variable is a constant equal to gamma. */
MOSEKCALL(res, MSK_putvarbound(task, offsets + 0, MSK_BK_FX, gamma, gamma));
sprintf(buf, "s");
MOSEKCALL(res, MSK_putvarname(task, offsets + 0, buf));

/* t variables (t = GT*x). */
for (j = 0; j < n; ++j)
{
  /* Copying the GT matrix in the appropriate block of A */
  for (k = 0; k < n; ++k)
    if ( GT[k][j] != 0.0 )
      MOSEKCALL(res, MSK_putaij(task, 1 + k, offsetx + j, GT[k][j]));
  /* Diagonal -1 entries in a block of A */
  MOSEKCALL(res, MSK_putaij(task, 1 + j, offsett + j, -1.0));
  /* Free - no bounds */
  MOSEKCALL(res, MSK_putvarbound(task, offsett + j, MSK_BK_FR, -MSK_INFINITY, MSK_INFINITY));
  sprintf(buf, "t[%d]", 1 + j);
  MOSEKCALL(res, MSK_putvarname(task, offsett + j, buf));
}

if ( res == MSK_RES_OK )
{
  /* Define the cone spanned by variables (s, t), i.e. dimension = n + 1 */
  MSKint32t *sub = (MSKint32t *) MSK_calloctask(task, n + 1, sizeof(MSKint32t));

  if ( sub )
  {
    /* Copy indices of variables involved in the conic constraint */
    sub[0] = offsets + 0;
    for (j = 0; j < n; ++j)
      sub[j + 1] = offsett + j;

    MOSEKCALL(res, MSK_appendcone(task, MSK_CT_QUAD, 0.0, n + 1, sub));
    MOSEKCALL(res, MSK_putconename(task, 0, "stddev"));

    MSK_freetask(task, sub);
  }
  else
    res = MSK_RES_ERR_SPACE;
}
```

```
  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.opf"));
#endif

  MOSEKCALL(res, MSK_optimize(task));

#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;
    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, offsetx + j, offsetx + j + 1, &xj));
      expret += mu[j] * xj;
    }

    /* Read the value of s. This should be gamma. */
    MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITR, offsets + 0, offsets + 1, &stddev));

    printf("\nExpected return %e for gamma %e\n", expret, stddev);
  }

  if ( task != NULL )
    MSK_deletetask(&task);

  if ( env != NULL )
    MSK_deleteenv(&env);

  return ( 0 );
}
```

The above code produces the result:

Listing 11.2: Output from the solver.

```
Interior-point solution summary
  Problem status  : PRIMAL_AND_DUAL_FEASIBLE
  Solution status : OPTIMAL
  Primal.  obj: 7.4766507287e-02    nrm: 1e+00    Viol.  con: 2e-08    var: 0e+00    cones: 2e-
↪08
  Dual.    obj: 7.4766554102e-02    nrm: 3e-01    Viol.  con: 0e+00    var: 3e-08    cones:␣
↪0e+00

Expected return 7.476651e-02 for gamma 5.000000e-02
```

Source code comments

The source code is a direct translation of the model (11.5) using the explicit block matrix specification (11.8) but a few comments are nevertheless in place.

The code uses a macro called MOSEKCALL which is defined as follows

```
#define MOSEKCALL(_r,_call)  ( (_r)==MSK_RES_OK ? ( (_r) = (_call) ) : ( (_r) = (_r) ) );
```

so for instance

```
MOSEKCALL(res, MSK_optimize());
```

is the same as

```
if ( res == MSK_RES_OK )
  res = MSK_optimize()
```

so MOSEKCALL is a method for hiding `if` statements and hence making the code more compact.

In the lines

```
  offsetx = 0;   /* Offset of variable x into the API variable. */
  offsets = n;   /* Offset of variable x into the API variable. */
  offsett = n + 1; /* Offset of variable t into the API variable. */
```

offsets into the **MOSEK** API variable are stored as in Table 11.1. The code

```
  for (j = 0; j < n; ++j)
  {
    /* Return of asset j in the objective */
    MOSEKCALL(res, MSK_putcj(task, offsetx + j, mu[j]));
    /* Coefficients in the first row of A */
    MOSEKCALL(res, MSK_putaij(task, 0, offsetx + j, 1.0));
    /* No short-selling - x^l = 0, x^u = inf */
    MOSEKCALL(res, MSK_putvarbound(task, offsetx + j, MSK_BK_LO, 0.0, MSK_INFINITY));
    sprintf(buf, "x[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, offsetx + j, buf));
  }
```

sets up the data for x variables. For instance

```
    /* Return of asset j in the objective */
    MOSEKCALL(res, MSK_putcj(task, offsetx + j, mu[j]));
```

inputs the objective coefficients for the x variables. Moreover, the code

```
    sprintf(buf, "x[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, offsetx + j, buf));
```

assigns meaningful names to the API variables. This is not needed but it makes debugging easier.

Note that the solution values are only accessed for the interesting variables; for instance the auxiliary variable t is omitted from this process.

Debugging Tips

Implementing an optimization model in Optimizer API for C can be error-prone. In order to check the code for accidental errors it is very useful to dump the problem to a file in a human readable form for visual inspection. The line

```
/* Dump the problem to a human readable OPF file. */
MOSEKCALL(res, MSK_writedata(task, "dump.opf"));
```

does that and it produces a file with the content:

Listing 11.3: Problem (11.5) stored in OPF format.

```
[comment]
   Written by MOSEK version 8.1.0.24
   Date 11-09-17
   Time 14:34:24
[/comment]

[hints]
  [hint NUMVAR] 7 [/hint]
  [hint NUMCON] 4 [/hint]
  [hint NUMANZ] 12 [/hint]
  [hint NUMQNZ] 0 [/hint]
  [hint NUMCONE] 1 [/hint]
[/hints]

[variables disallow_new_variables]
  'x[1]' 'x[2]' 'x[3]' s 't[1]'
  't[2]' 't[3]'
[/variables]

[objective maximize]
   1.073e-01 'x[1]' + 7.37e-02 'x[2]' + 6.270000000000001e-02 'x[3]'
[/objective]

[constraints]
  [con 'budget']  'x[1]' + 'x[2]' + 'x[3]' = 1e+00 [/con]
  [con 'GT[1]']  1.667e-01 'x[1]' + 2.32e-02 'x[2]' + 1.3e-03 'x[3]' - 't[1]' = 0e+00 [/con]
  [con 'GT[2]']  1.033e-01 'x[2]' - 2.2e-03 'x[3]' - 't[2]' = 0e+00 [/con]
  [con 'GT[3]']  3.38e-02 'x[3]' - 't[3]' = 0e+00 [/con]
[/constraints]

[bounds]
  [b] 0e+00      <= 'x[1]','x[2]','x[3]' [/b]
  [b]               s =  5e-02 [/b]
  [b]               't[1]','t[2]','t[3]' free [/b]
  [cone quad 'stddev'] s, 't[1]', 't[2]', 't[3]' [/cone]
[/bounds]
```

Since the API variables have been given meaningful names it is easy to verify by hand that the model is correct.

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. This leads to the concept of efficient frontier.

Given a nonnegative α the optimization problem

$$\begin{array}{lccc} \text{maximize} & \mu^T x - \alpha s & & \\ \text{subject to} & e^T x & = & w + e^T x^0, \\ & [s; G^T x] & \in & \mathcal{Q}^{n+1}, \\ & x & \geq & 0. \end{array} \tag{11.9}$$

computes an efficient portfolio which maximizes expected return while minimizing risk, where the tradeoff between the two is controlled by α. Ideally the problem (11.9) should be solved for all values $\alpha \geq 0$ but in practice that is impossible.

For the example data from Sec. 11.1.1, the optimal values of return and risk for a range of αs are listed below:

Listing 11.4: Results obtained solving problem (11.9) for different values of α.

```
alpha          exp ret        std dev
0.000e+000     1.073e-001     7.261e-001
2.500e-001     1.033e-001     1.499e-001
5.000e-001     6.976e-002     3.735e-002
7.500e-001     6.766e-002     3.383e-002
1.000e+000     6.679e-002     3.281e-002
1.500e+000     6.599e-002     3.214e-002
2.000e+000     6.560e-002     3.192e-002
2.500e+000     6.537e-002     3.181e-002
3.000e+000     6.522e-002     3.176e-002
3.500e+000     6.512e-002     3.173e-002
4.000e+000     6.503e-002     3.170e-002
4.500e+000     6.497e-002     3.169e-002
```

Source code example

The example code in Listing 11.5 demonstrates how to compute the efficient portfolios for several values of α. The code is mostly similar to the one in Sec. 11.1.1, except the problem is re-optimized in a loop for varying α.

Listing 11.5: Code implementing model (11.9).

```
#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           = 3,
                  numalpha    = 12;
  const double    mu[]        = {0.1073,  0.0737,  0.0627},
                  x0[3]       = {0.0, 0.0, 0.0},
                  w           = 1.0,
                  alphas[12]  = {0.0, 0.25, 0.5, 0.75, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5},
                  GT[][3]     = {{0.1667,  0.0232,  0.0013},
                                 {0.0000,  0.1033, -0.0022},
                                 {0.0000,  0.0000,  0.0338}};
  double          expret,
                  stddev,
                  alpha;
  MSKenv_t        env;
  MSKint32t       k, i, j, offsetx, offsets, offsett;
  MSKrescodee     res = MSK_RES_OK;
  MSKtask_t       task;
  MSKrealt        xj;
  MSKsolstae      solsta;

  /* 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));

  /* Constraints. */
  MOSEKCALL(res, MSK_appendcons(task, 1 + n));
  MOSEKCALL(res, MSK_putconbound(task, 0, MSK_BK_FX, 1.0, 1.0));
  sprintf(buf, "%s", "budget");
  MOSEKCALL(res, MSK_putconname(task, 0, buf));

  for (i = 0; i < n; ++i)
  {
    MOSEKCALL(res, MSK_putconbound(task, 1 + i, MSK_BK_FX, 0.0, 0.0));
    sprintf(buf, "GT[%d]", 1 + i);
    MOSEKCALL(res, MSK_putconname(task, 1 + i, buf));
  }

  /* Variables. */
  MOSEKCALL(res, MSK_appendvars(task, 1 + 2 * n));

  offsetx = 0;   /* Offset of variable x into the API variable. */
  offsets = n;   /* Offset of variable x into the API variable. */
  offsett = n + 1; /* Offset of variable t into the API variable. */

  /* x variables. */
  for (j = 0; j < n; ++j)
  {
    MOSEKCALL(res, MSK_putcj(task, offsetx + j, mu[j]));
    MOSEKCALL(res, MSK_putaij(task, 0, offsetx + j, 1.0));
    for (k = 0; k < n; ++k)
      if ( GT[k][j] != 0.0 )
        MOSEKCALL(res, MSK_putaij(task, 1 + k, offsetx + j, GT[k][j]));

    MOSEKCALL(res, MSK_putvarbound(task, offsetx + j, MSK_BK_LO, 0.0, MSK_INFINITY));
    sprintf(buf, "x[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, offsetx + j, buf));
  }

  /* s variable. */
  MOSEKCALL(res, MSK_putvarbound(task, offsets + 0, MSK_BK_FR, -MSK_INFINITY, MSK_INFINITY));
```

```
sprintf(buf, "s");
MOSEKCALL(res, MSK_putvarname(task, offsets + 0, buf));

/* t variables. */
for (j = 0; j < n; ++j)
{
  MOSEKCALL(res, MSK_putaij(task, 1 + j, offsett + j, -1.0));
  MOSEKCALL(res, MSK_putvarbound(task, offsett + j, MSK_BK_FR, -MSK_INFINITY, MSK_INFINITY));
  sprintf(buf, "t[%d]", 1 + j);
  MOSEKCALL(res, MSK_putvarname(task, offsett + j, buf));
}

if ( res == MSK_RES_OK )
{
  /* sub should be n+1 long i.e. the dimmension of the cone. */
  MSKint32t *sub = (MSKint32t *) MSK_calloctask(task, n + 1, sizeof(MSKint32t));

  if ( sub )
  {
    sub[0] = offsets + 0;
    for (j = 0; j < n; ++j)
      sub[j + 1] = offsett + j;

    MOSEKCALL(res, MSK_appendcone(task, MSK_CT_QUAD, 0.0, n + 1, sub));
    MOSEKCALL(res, MSK_putconename(task, 0, "stddev"));

    MSK_freetask(task, sub);
  }
  else
    res = MSK_RES_ERR_SPACE;
}

MOSEKCALL(res, MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE));

/* Turn all log output off. */
MOSEKCALL(res, MSK_putintparam(task, MSK_IPAR_LOG, 0));

printf("%-12s  %-12s  %-12s\n", "alpha", "exp ret", "std dev");

for (k = 0; k < numalpha; ++k)
{
  alpha = alphas[k];

  /* Sets the objective function coefficient for s. */
  MOSEKCALL(res, MSK_putcj(task, offsets + 0, -alpha));

  MOSEKCALL(res, MSK_optimize(task));

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

  if ( solsta == MSK_SOL_STA_OPTIMAL || solsta == MSK_SOL_STA_NEAR_OPTIMAL )
  {
    expret = 0.0;
    for (j = 0; j < n; ++j)
    {
      MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITR, offsetx + j, offsetx + j + 1, &xj));
      expret += mu[j] * xj;
    }

    MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITR, offsets + 0, offsets + 1, &stddev));

    printf("%-12.3e  %-12.3e  %-12.3e\n", alpha, expret, stddev);
  }
```

```
    else
    {
      printf("An error occurred when solving for alpha=%e\n", alpha);
    }
  }

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

  return ( 0 );
}
```

11.1.3 Improving the Computational 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 modelling stage.

The computational cost is of course to some extent dependent on the number of constraints and variables in the optimization problem. However, in practice a more important factor is the sparsity: the number of nonzeros used to represent the problem. Indeed it is often better to focus on the number of nonzeros in G see (11.2) and try to reduce that number by for instance changing the choice of G.

In other words if the computational efficiency should be improved then it is always good idea to start with focusing at the covariance matrix. As an example assume that

$$\Sigma = D + VV^T$$

where D is a positive definite diagonal matrix. Moreover, V is a matrix with n rows and p columns. Such a model for the covariance matrix is called a factor model and usually p is much smaller than n. In practice p tends to be a small number independent of n, say less than 100.

One possible choice for G is the Cholesky factorization of Σ which requires storage proportional to $n(n+1)/2$. However, another choice is

$$G^T = \left[\begin{array}{c} D^{1/2} \\ V^T \end{array} \right]$$

because then

$$GG^T = D + VV^T.$$

This choice requires storage proportional to $n + pn$ which is much less than for the Cholesky choice of G. Indeed assuming p is a constant storage requirements are reduced by a factor of n.

The example above exploits the so-called factor structure and demonstrates that an alternative choice of G may lead to a significant reduction in the amount of storage used to represent the problem. This will in most cases also lead to a significant reduction in the solution time.

The lesson to be learned is that it is important to investigate how the covariance matrix is formed. Given this knowledge it might be possible to make a special choice for G that helps reducing the storage requirements and enhance the computational efficiency. More details about this process can be found in *[And13]*.

11.1.4 Slippage Cost

The basic Markowitz model assumes that there are no costs associated with trading the assets and that the returns of the assets are independent of the amount traded. Neither of those assumptions is usually valid in practice. Therefore, a more realistic model is

$$
\begin{array}{lrcl}
\text{maximize} & \mu^T x & & \\
\text{subject to} & e^T x + \sum_{j=1}^{n} C_j(x_j - x_j^0) & = & w + e^T x^0, \\
 & x^T \Sigma x & \leq & \gamma^2, \\
 & x & \geq & 0,
\end{array} \tag{11.10}
$$

where the function

$$C_j(x_j - x_j^0)$$

specifies the transaction costs when the holding of asset j is changed from its initial value.

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 modelled by

$$C_j = m_j \sqrt{|x_j - x_j^0|}$$

where m_j is a constant that is estimated in some way by the trader. See *[GK00]* [p. 452] for details. Hence, we have

$$C_j(x_j - x_j^0) = m_j |x_j - x_j^0| \sqrt{|x_j - x_j^0|} = m_j |x_j - x_j^0|^{3/2}.$$

From *[MOSEKApS12]* it is known that

$$\{(c, z) : c \geq z^{3/2}, z \geq 0\} = \{(c, z) : (v, c, z), (z, 1/8, v) \in \mathcal{Q}_r^3\}$$

where $\mathcal{Q}_r^3$ is the 3-dimensional rotated quadratic cone. Hence, it follows

$$
\begin{array}{rl}
z_j & = |x_j - x_j^0|, \\
(v_j, c_j, z_j), (z_j, 1/8, v_j) & \in \mathcal{Q}_r^3, \\
\sum_{j=1}^{n} C_j(x_j - x_j^0) & = \sum_{j=1}^{n} c_j.
\end{array}
$$

Unfortunately this set of constraints is nonconvex due to the constraint

$$z_j = |x_j - x_j^0| \tag{11.11}$$

but in many cases the constraint may be replaced by the relaxed constraint

$$z_j \geq |x_j - x_j^0|, \tag{11.12}$$

which is equivalent to

$$
\begin{array}{l}
z_j \geq x_j - x_j^0, \\
z_j \geq -(x_j - x_j^0).
\end{array} \tag{11.13}
$$

For instance if the universe of assets contains a risk free asset then

$$z_j > |x_j - x_j^0| \tag{11.14}$$

cannot hold for an optimal solution.

If the optimal solution has the property (11.14) 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 (11.11) and (11.12) are equivalent.

The above observations lead to

$$
\begin{array}{lrcll}
\text{maximize} & \mu^T x & & & \\
\text{subject to} & e^T x + m^T c & = & w + e^T x^0, & \\
& [\gamma; G^T x] & \in & \mathcal{Q}^{n+1}, & \\
& z_j & \geq & x_j - x_j^0, & j = 1, \ldots, n, \\
& z_j & \geq & x_j^0 - x_j, & j = 1, \ldots, n, \\
& [v_j; c_j; z_j], [z_j; 1/8; v_j] & \in & \mathcal{Q}_r^3, & j = 1, \ldots, n, \\
& x & \geq & 0. &
\end{array}
\tag{11.15}
$$

The revised budget constraint

$$
e^T x + m^T c = w + e^T x^0
$$

specifies that the initial wealth covers the investment and the transaction costs. Moreover, v and z are auxiliary variables that model the market impact cost so that $z_j \geq |x_j - x_j^0|$ and $c_j \geq z_j^{3/2}$.

It should be mentioned that transaction costs of the form

$$
c_j \geq z_j^{p/q}
$$

where p and q are both integers and $p \geq q$ can be modelled using quadratic cones. See *[MOSEKApS12]* for details.

Creating a matrix formulation

One more reformulation of (11.15) is needed to bring it to the standard form (11.4).

$$
\begin{array}{lrcll}
\text{maximize} & \mu^T x & & & \\
\text{subject to} & e^T x + m^T c & = & w + e^T x^0, & \\
& G^T x - t & = & 0, & \\
& z_j - x_j & \geq & -x_j^0, & j = 1, \ldots, n, \\
& z_j + x_j & \geq & x_j^0, & j = 1, \ldots, n, \\
& [v_j; c_j; z_j] - [f_{j,1}; f_{j,2}; f_{j,3}] & = & 0, & j = 1, \ldots, n, \\
& [z_j; 0; v_j] - [g_{j,1}; g_{j,2}; g_{j,3}] & = & [0; -1/8; 0], & j = 1, \ldots, n, \\
& [s; t] & \in & \mathcal{Q}^{n+1}, & \\
& [f_{j,1}; f_{j,2}; f_{j,3}] & \in & \mathcal{Q}_r^3, & j = 1, \ldots, n, \\
& [g_{j,1}; g_{j,2}; g_{j,3}] & \in & \mathcal{Q}_r^3, & j = 1, \ldots, n, \\
& x & \geq & 0, & \\
& s & = & \gamma, &
\end{array}
\tag{11.16}
$$

where $f, g \in \mathbb{R}^{n \times 3}$. The additional variables f and g are introduced to ensure that each variable appears at most once in any cone.

The formulation (11.16) is not the most compact possible, but it is easy to implement. **MOSEK** presolve will automatically simplify it.

The first step in developing the implementation is to chose an ordering of the variables. We will choose the following ordering:

$$
\hat{x} = [x; s; t; c; v; z; f; g]
$$

Table 11.2 shows the mapping between the $\hat{x}$ vector and the model variables.

Table 11.2: Storage layout for the $\hat{x}$

Variable	Length	Offset
x	n	0
s	1	n
t	n	$n+1$
c	n	$2n+1$
v	n	$3n+1$
z	n	$4n+1$
$f(:)^T$	$3n$	$5n+1$
$g(:)^T$	$3n$	$8n+1$

The next step is to consider how the linear constraint matrix A and the remaining data vectors are laid out. Reusing the idea in Sec. 11.1.1 we can write the data in block matrix form and read off all the required coordinates. This extension of the code setting up the constraint $G^T x - t = 0$ from Sec. 11.1.1 is shown below.

Source code example

The example code in Listing 11.6 demonstrates how to implement the model (11.16).

Listing 11.6: Code implementing model (11.16).

```
#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       = 3;
  const double    w       = 1.0,
                  x0[]    = {0.0, 0.0, 0.0},
                  gamma   = 0.05,
                  mu[]    = {0.1073,  0.0737,  0.0627},
                  m[]     = {0.01, 0.01, 0.01},
                  GT[][3] = {{0.1667,  0.0232,  0.0013},
                   {0.0000,  0.1033, -0.0022},
                   {0.0000,  0.0000,  0.0338}
                  };
  double          b[3]    = {0.0, -1.0 / 8.0, 0.0};
  double          rtemp,
                  expret,
                  stddev,
                  xj;
  MSKenv_t        env;
  MSKint32t       k, i, j,
                  offsetx, offsets, offsett, offsetc,
                  offsetv, offsetz, offsetf, offsetg;
  MSKrescodee     res = MSK_RES_OK;
  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));

rtemp = w;
for (k = 0; k < n; ++k)
  rtemp += x0[k];

/* Constraints. */
MOSEKCALL(res, MSK_appendcons(task, 1 + 9 * n));
MOSEKCALL(res, MSK_putconbound(task, 0, MSK_BK_FX, w, w));
sprintf(buf, "%s", "budget");
MOSEKCALL(res, MSK_putconname(task, 0, buf));

for (i = 0; i < n; ++i)
{
  MOSEKCALL(res, MSK_putconbound(task, 1 + i, MSK_BK_FX, 0.0, 0.0));
  sprintf(buf, "GT[%d]", 1 + i);
  MOSEKCALL(res, MSK_putconname(task, 1 + i, buf));
}

for (i = 0; i < n; ++i)
{
  MOSEKCALL(res, MSK_putconbound(task, 1 + n + i, MSK_BK_LO, -x0[i], MSK_INFINITY));
  sprintf(buf, "zabs1[%d]", 1 + i);
  MOSEKCALL(res, MSK_putconname(task, 1 + n + i, buf));
}

for (i = 0; i < n; ++i)
{
  MOSEKCALL(res, MSK_putconbound(task, 1 + 2 * n + i, MSK_BK_LO, x0[i], MSK_INFINITY));
  sprintf(buf, "zabs2[%d]", 1 + i);
  MOSEKCALL(res, MSK_putconname(task, 1 + 2 * n + i, buf));
}

for (i = 0; i < n; ++i)
{
  for (k = 0; k < 3; ++k)
  {
    MOSEKCALL(res, MSK_putconbound(task, 1 + 3 * n + 3 * i + k, MSK_BK_FX, 0.0, 0.0));
    sprintf(buf, "f[%d,%d]", 1 + i, 1 + k);
    MOSEKCALL(res, MSK_putconname(task, 1 + 3 * n + 3 * i + k, buf));
  }
}

for (i = 0; i < n; ++i)
{
  for (k = 0; k < 3; ++k)
  {
    MOSEKCALL(res, MSK_putconbound(task, 1 + 6 * n + 3 * i + k, MSK_BK_FX, b[k], b[k]));
    sprintf(buf, "g[%d,%d]", 1 + i, 1 + k);
    MOSEKCALL(res, MSK_putconname(task, 1 + 6 * n + 3 * i + k, buf));
  }
}

/* Offsets of variables into the (serialized) API variable. */
offsetx = 0;
offsets = n;
offsett = n + 1;
```

```
offsetc = 2 * n + 1;
offsetv = 3 * n + 1;
offsetz = 4 * n + 1;
offsetf = 5 * n + 1;
offsetg = 8 * n + 1;


/* Variables. */
MOSEKCALL(res, MSK_appendvars(task, 11 * n + 1));

/* x variables. */
for (j = 0; j < n; ++j)
{
  MOSEKCALL(res, MSK_putcj(task, offsetx + j, mu[j]));
  MOSEKCALL(res, MSK_putaij(task, 0, offsetx + j, 1.0));
  for (k = 0; k < n; ++k)
    if ( GT[k][j] != 0.0 )
      MOSEKCALL(res, MSK_putaij(task, 1 + k, offsetx + j, GT[k][j]));
  MOSEKCALL(res, MSK_putaij(task, 1 + n + j, offsetx + j, -1.0));
  MOSEKCALL(res, MSK_putaij(task, 1 + 2 * n + j, offsetx + j, 1.0));

  MOSEKCALL(res, MSK_putvarbound(task, offsetx + j, MSK_BK_LO, 0.0, MSK_INFINITY));
  sprintf(buf, "x[%d]", 1 + j);
  MOSEKCALL(res, MSK_putvarname(task, offsetx + j, buf));
}

/* s variable. */
MOSEKCALL(res, MSK_putvarbound(task, offsets + 0, MSK_BK_FX, gamma, gamma));
sprintf(buf, "s");
MOSEKCALL(res, MSK_putvarname(task, offsets + 0, buf));

/* t variables. */
for (j = 0; j < n; ++j)
{
  MOSEKCALL(res, MSK_putaij(task, 1 + j, offsett + j, -1.0));
  MOSEKCALL(res, MSK_putvarbound(task, offsett + j, MSK_BK_FR, -MSK_INFINITY, MSK_INFINITY));
  sprintf(buf, "t[%d]", 1 + j);
  MOSEKCALL(res, MSK_putvarname(task, offsett + j, buf));
}

/* c variables. */
for (j = 0; j < n; ++j)
{
  MOSEKCALL(res, MSK_putaij(task, 0, offsetc + j, m[j]));
  MOSEKCALL(res, MSK_putaij(task, 1 + 3 * n + 3 * j + 1, offsetc + j, 1.0));
  MOSEKCALL(res, MSK_putvarbound(task, offsetc + j, MSK_BK_FR, -MSK_INFINITY, MSK_INFINITY));
  sprintf(buf, "c[%d]", 1 + j);
  MOSEKCALL(res, MSK_putvarname(task, offsetc + j, buf));
}

/* v variables. */
for (j = 0; j < n; ++j)
{
  MOSEKCALL(res, MSK_putaij(task, 1 + 3 * n + 3 * j + 0, offsetv + j, 1.0));
  MOSEKCALL(res, MSK_putaij(task, 1 + 6 * n + 3 * j + 2, offsetv + j, 1.0));
  MOSEKCALL(res, MSK_putvarbound(task, offsetv + j, MSK_BK_FR, -MSK_INFINITY, MSK_INFINITY));
  sprintf(buf, "v[%d]", 1 + j);
  MOSEKCALL(res, MSK_putvarname(task, offsetv + j, buf));
}

/* z variables. */
for (j = 0; j < n; ++j)
{
```

```
    MOSEKCALL(res, MSK_putaij(task, 1 + 1 * n + j, offsetz + j, 1.0));
    MOSEKCALL(res, MSK_putaij(task, 1 + 2 * n + j, offsetz + j, 1.0));
    MOSEKCALL(res, MSK_putaij(task, 1 + 3 * n + 3 * j + 2, offsetz + j, 1.0));
    MOSEKCALL(res, MSK_putaij(task, 1 + 6 * n + 3 * j + 0, offsetz + j, 1.0));
    MOSEKCALL(res, MSK_putvarbound(task, offsetz + j, MSK_BK_FR, -MSK_INFINITY, MSK_INFINITY));
    sprintf(buf, "z[%d]", 1 + j);
    MOSEKCALL(res, MSK_putvarname(task, offsetz + j, buf));
  }

  /* f variables. */
  for (j = 0; j < n; ++j)
  {
    for (k = 0; k < 3; ++k)
    {
      MOSEKCALL(res, MSK_putaij(task, 1 + 3 * n + 3 * j + k, offsetf + 3 * j + k, -1.0));
      MOSEKCALL(res, MSK_putvarbound(task, offsetf + 3 * j + k, MSK_BK_FR, -MSK_INFINITY, MSK_
↪INFINITY));
      sprintf(buf, "f[%d,%d]", 1 + j, 1 + k);
      MOSEKCALL(res, MSK_putvarname(task, offsetf + 3 * j + k, buf));
    }
  }

  /* g variables. */
  for (j = 0; j < n; ++j)
  {
    for (k = 0; k < 3; ++k)
    {
      MOSEKCALL(res, MSK_putaij(task, 1 + 6 * n + 3 * j + k, offsetg + 3 * j + k, -1.0));
      MOSEKCALL(res, MSK_putvarbound(task, offsetg + 3 * j + k, MSK_BK_FR, -MSK_INFINITY, MSK_
↪INFINITY));
      sprintf(buf, "g[%d,%d]", 1 + j, 1 + k);
      MOSEKCALL(res, MSK_putvarname(task, offsetg + 3 * j + k, buf));
    }
  }
  if ( res == MSK_RES_OK )
  {
    /* sub should be n+1 long i.e. the dimmension of the cone. */
    MSKint32t *sub = (MSKint32t *) MSK_calloctask(task, 3 >= n + 1 ? 3 : n + 1,␣
↪sizeof(MSKint32t));

    if ( sub )
    {
      sub[0] = offsets + 0;
      for (j = 0; j < n; ++j)
        sub[j + 1] = offsett + j;

      MOSEKCALL(res, MSK_appendcone(task, MSK_CT_QUAD, 0.0, n + 1, sub));
      MOSEKCALL(res, MSK_putconename(task, 0, "stddev"));

      for (k = 0; k < n; ++k)
      {
        MOSEKCALL(res, MSK_appendconeseq(task, MSK_CT_RQUAD, 0.0, 3, offsetf + k * 3));
        sprintf(buf, "f[%d]", 1 + k);
        MOSEKCALL(res, MSK_putconename(task, 1 + k, buf));
      }

      for (k = 0; k < n; ++k)
      {
        MOSEKCALL(res, MSK_appendconeseq(task, MSK_CT_RQUAD, 0.0, 3, offsetg + k * 3));
        sprintf(buf, "g[%d]", 1 + k);
        MOSEKCALL(res, MSK_putconename(task, 1 + n + k, buf));
      }
```

```
      MSK_freetask(task, sub);
    }
    else
      res = MSK_RES_ERR_SPACE;
  }

  MOSEKCALL(res, MSK_putobjsense(task, MSK_OBJECTIVE_SENSE_MAXIMIZE));

#if 1
  /* no log output. */
#else
  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.opf"));
#endif

  MOSEKCALL(res, MSK_optimize(task));

  /* 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, offsetx + j, offsetx + j + 1, &xj));
      expret += mu[j] * xj;
    }

    MOSEKCALL(res, MSK_getxxslice(task, MSK_SOL_ITR, offsets + 0, offsets + 1, &stddev));

    printf("\nExpected return %e for gamma %e\n", expret, stddev);
  }

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

  return ( 0 );
}
```

The example code above produces the result

```
Interior-point solution summary
  Problem status  : PRIMAL_AND_DUAL_FEASIBLE
  Solution status : OPTIMAL
  Primal.  obj: 7.4390660847e-02    nrm: 1e+00    Viol.  con: 6e-09    var: 0e+00    cones: 4e-
↪09
  Dual.    obj: 7.4390675795e-02    nrm: 3e-01    Viol.  con: 1e-19    var: 8e-09    cones:␣
↪0e+00

Expected return 7.439066e-02 for gamma 5.000000e-02
```

If the problem is dumped to an OPF file, it has the following content.

Listing 11.7: OPF file for problem (11.16).

```
[comment]
   Written by MOSEK version 8.1.0.24
   Date 12-09-17
   Time 12:34:27
[/comment]

[hints]
  [hint NUMVAR] 34 [/hint]
  [hint NUMCON] 28 [/hint]
  [hint NUMANZ] 60 [/hint]
  [hint NUMQNZ] 0 [/hint]
  [hint NUMCONE] 7 [/hint]
[/hints]

[variables disallow_new_variables]
  'x[1]' 'x[2]' 'x[3]' s 't[1]'
  't[2]' 't[3]' 'c[1]' 'c[2]' 'c[3]'
  'v[1]' 'v[2]' 'v[3]' 'z[1]' 'z[2]'
  'z[3]' 'f[1,1]' 'f[1,2]' 'f[1,3]' 'f[2,1]'
  'f[2,2]' 'f[2,3]' 'f[3,1]' 'f[3,2]' 'f[3,3]'
  'g[1,1]' 'g[1,2]' 'g[1,3]' 'g[2,1]' 'g[2,2]'
  'g[2,3]' 'g[3,1]' 'g[3,2]' 'g[3,3]'
[/variables]

[objective maximize]
   1.073e-01 'x[1]' + 7.37e-02 'x[2]' + 6.270000000000001e-02 'x[3]'
[/objective]

[constraints]
  [con 'budget']  'x[1]' + 'x[2]' + 'x[3]' + 1e-02 'c[1]' + 1e-02 'c[2]'
     + 1e-02 'c[3]' = 1e+00 [/con]
  [con 'GT[1]']  1.667e-01 'x[1]' + 2.32e-02 'x[2]' + 1.3e-03 'x[3]' - 't[1]' = 0e+00 [/con]
  [con 'GT[2]']  1.033e-01 'x[2]' - 2.2e-03 'x[3]' - 't[2]' = 0e+00 [/con]
  [con 'GT[3]']  3.38e-02 'x[3]' - 't[3]' = 0e+00 [/con]
  [con 'zabs1[1]'] 0e+00 <= - 'x[1]' + 'z[1]' [/con]
  [con 'zabs1[2]'] 0e+00 <= - 'x[2]' + 'z[2]' [/con]
  [con 'zabs1[3]'] 0e+00 <= - 'x[3]' + 'z[3]' [/con]
  [con 'zabs2[1]'] 0e+00 <= 'x[1]' + 'z[1]' [/con]
  [con 'zabs2[2]'] 0e+00 <= 'x[2]' + 'z[2]' [/con]
  [con 'zabs2[3]'] 0e+00 <= 'x[3]' + 'z[3]' [/con]
  [con 'f[1,1]']  'v[1]' - 'f[1,1]' = 0e+00 [/con]
  [con 'f[1,2]']  'c[1]' - 'f[1,2]' = 0e+00 [/con]
  [con 'f[1,3]']  'z[1]' - 'f[1,3]' = 0e+00 [/con]
  [con 'f[2,1]']  'v[2]' - 'f[2,1]' = 0e+00 [/con]
  [con 'f[2,2]']  'c[2]' - 'f[2,2]' = 0e+00 [/con]
  [con 'f[2,3]']  'z[2]' - 'f[2,3]' = 0e+00 [/con]
  [con 'f[3,1]']  'v[3]' - 'f[3,1]' = 0e+00 [/con]
  [con 'f[3,2]']  'c[3]' - 'f[3,2]' = 0e+00 [/con]
  [con 'f[3,3]']  'z[3]' - 'f[3,3]' = 0e+00 [/con]
  [con 'g[1,1]']  'z[1]' - 'g[1,1]' = 0e+00 [/con]
  [con 'g[1,2]']  - 'g[1,2]' = -1.25e-01 [/con]
  [con 'g[1,3]']  'v[1]' - 'g[1,3]' = 0e+00 [/con]
  [con 'g[2,1]']  'z[2]' - 'g[2,1]' = 0e+00 [/con]
  [con 'g[2,2]']  - 'g[2,2]' = -1.25e-01 [/con]
  [con 'g[2,3]']  'v[2]' - 'g[2,3]' = 0e+00 [/con]
  [con 'g[3,1]']  'z[3]' - 'g[3,1]' = 0e+00 [/con]
  [con 'g[3,2]']  - 'g[3,2]' = -1.25e-01 [/con]
  [con 'g[3,3]']  'v[3]' - 'g[3,3]' = 0e+00 [/con]
[/constraints]
```

```
[bounds]
  [b] 0e+00      <= 'x[1]','x[2]','x[3]' [/b]
  [b]               s =  5e-02 [/b]
  [b]               't[1]','t[2]','t[3]','c[1]','c[2]','c[3]' free [/b]
  [b]               'v[1]','v[2]','v[3]','z[1]','z[2]','z[3]' free [/b]
  [b]               'f[1,1]','f[1,2]','f[1,3]','f[2,1]','f[2,2]','f[2,3]' free [/b]
  [b]               'f[3,1]','f[3,2]','f[3,3]','g[1,1]','g[1,2]','g[1,3]' free [/b]
  [b]               'g[2,1]','g[2,2]','g[2,3]','g[3,1]','g[3,2]','g[3,3]' free [/b]
  [cone quad 'stddev'] s, 't[1]', 't[2]', 't[3]' [/cone]
  [cone rquad 'f[1]'] 'f[1,1]', 'f[1,2]', 'f[1,3]' [/cone]
  [cone rquad 'f[2]'] 'f[2,1]', 'f[2,2]', 'f[2,3]' [/cone]
  [cone rquad 'f[3]'] 'f[3,1]', 'f[3,2]', 'f[3,3]' [/cone]
  [cone rquad 'g[1]'] 'g[1,1]', 'g[1,2]', 'g[1,3]' [/cone]
  [cone rquad 'g[2]'] 'g[2,1]', 'g[2,2]', 'g[2,3]' [/cone]
  [cone rquad 'g[3]'] 'g[3,1]', 'g[3,2]', 'g[3,3]' [/cone]
[/bounds]
```

The file verifies that the correct problem has been set up.

CHAPTER

TWELVE

PROBLEM FORMULATION AND SOLUTIONS

In this chapter we will discuss the following issues:

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

12.1 Linear Optimization

A linear optimization problem can be written as

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

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.
- $A \in \mathbb{R}^{m \times n}$ 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.

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

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

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. E.g.

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

This is equivalent to removing variable $(s_l^x)_j$ from the dual problem. 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 Primal-dual Feasible Solution

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).

The Duality Gap

Let

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

be a primal-dual feasible solution, and let

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

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.

An Optimal Solution

It is well-known that a linear optimization problem has an optimal solution if and only if there exist feasible primal and dual solutions 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 its dual is infeasible. As the constraints to the dual of (12.4) are identical to the constraints of problem (12.1), we thus have that problem (12.1) is also 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 its dual is infeasible. As the constraints to the dual of (12.5) are identical to the constraints of problem (12.2), we thus have that problem (12.2) is also infeasible.

Primal and Dual Infeasible Case

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).

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 Quadratic Optimization

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

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

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

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

where $x = (x^1, \ldots, x^p)$ is a partition of the problem variables. Please note that the n-dimensional Euclidean space $\mathbb{R}^n$ is a cone itself, so simple linear variables are still allowed.

MOSEK supports only a limited number of cones, specifically:

- The $\mathbb{R}^n$ set.
- The quadratic cone:

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

- The rotated quadratic cone:

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

Although these cones may seem to provide only limited expressive power they can be used to model a wide range of problems as demonstrated in *[MOSEKApS12]*.

12.2.1 Duality for Conic Quadratic Optimization

The dual problem corresponding to the conic quadratic optimization problem (12.7) is given by

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

where the dual cone $\mathcal{K}^*$ is a Cartesian product of the cones

$$\mathcal{K}^* = \mathcal{K}_1^* \times \cdots \times \mathcal{K}_p^*,$$

where each $\mathcal{K}_t^*$ is the dual cone of $\mathcal{K}_t$. For the cone types **MOSEK** can handle, the relation between the primal and dual cone is given as follows:

- The $\mathbb{R}^n$ set:

$$\mathcal{K}_t = \mathbb{R}^{n_t} \quad \Leftrightarrow \quad \mathcal{K}_t^* = \{ s \in \mathbb{R}^{n_t} : \quad s = 0 \}.$$

- The quadratic cone:

$$\mathcal{K}_t = \mathcal{Q}^{n_t} \quad \Leftrightarrow \quad \mathcal{K}_t^* = \mathcal{Q}^{n_t} = \left\{ s \in \mathbb{R}^{n_t} : s_1 \geq \sqrt{\sum_{j=2}^{n_t} s_j^2} \right\}.$$

- The rotated quadratic cone:

$$\mathcal{K}_t = \mathcal{Q}_r^{n_t} \quad \Leftrightarrow \quad \mathcal{K}_t^* = \mathcal{Q}_r^{n_t} = \left\{ s \in \mathbb{R}^{n_t} : 2s_1s_2 \geq \sum_{j=3}^{n_t} s_j^2, \quad s_1 \geq 0, \quad s_2 \geq 0 \right\}.$$

Please note that the dual problem of the dual problem is identical to the original primal problem.

12.2.2 Infeasibility for Conic Quadratic Optimization

In case **MOSEK** finds a problem to be infeasible it reports a certificate of infeasibility. This works exactly as for linear problems (see Sec. 12.1.2).

Primal Infeasible Problems

If the problem (12.7) is infeasible, **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 problem

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

such that the objective value is strictly positive.

Dual infeasible problems

If the problem (12.8) is infeasible, **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 problem

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

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 the objective value is strictly negative.

12.3 Semidefinite Optimization

Semidefinite optimization is an extension of conic quadratic optimization (see Sec. 12.2) allowing positive semidefinite matrix variables to be used in addition to the usual scalar variables. A semidefinite optimization problem can be written as

$$\begin{array}{lcccll} \text{minimize} & & \sum_{j=0}^{n-1} c_j x_j + \sum_{j=0}^{p-1} \left\langle \overline{C}_j, \overline{X}_j \right\rangle + c^f & & & \\ \text{subject to} & l_i^c \leq & \sum_{j=0}^{n-1} a_{ij} x_j + \sum_{j=0}^{p-1} \left\langle \overline{A}_{ij}, \overline{X}_j \right\rangle & \leq & u_i^c, & i = 0, \ldots, m-1 \\ & l_j^x \leq & x_j & \leq & u_j^x, & j = 0, \ldots, n-1 \\ & & x \in \mathcal{K}, \overline{X}_j \in \mathcal{S}_+^{r_j}, & & & j = 0, \ldots, p-1 \end{array} \tag{12.9}$$

where the problem has p symmetric positive semidefinite variables $\overline{X}_j \in \mathcal{S}_+^{r_j}$ of dimension r_j with symmetric coefficient matrices $\overline{C}_j \in \mathcal{S}^{r_j}$ and $\overline{A}_{i,j} \in \mathcal{S}^{r_j}$. We use standard notation for the matrix inner product, i.e., for $U, V \in \mathbb{R}^{m \times n}$ we have

$$\langle U, V \rangle := \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} U_{ij} V_{ij}.$$

With semidefinite optimization we can model a wide range of problems as demonstrated in *[MOSEKApS12]*.

12.3.1 Duality for Semidefinite Optimization

The dual problem corresponding to the semidefinite optimization problem (12.9) is given by

$$\begin{array}{lll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c^f & \\ \text{subject to} & & \\ & c - A^T y + s_u^x - s_l^x = s_n^x, & \\ & \overline{C}_j - \sum_{i=0}^{m} y_i \overline{A}_{ij} = \overline{S}_j, & j = 0, \ldots, p-1 \\ & s_l^c - s_u^c = y, & \\ & s_l^c, s_u^c, s_l^x, s_u^x \geq 0, & \\ & s_n^x \in \mathcal{K}^*, \quad \overline{S}_j \in \mathcal{S}_+^{r_j}, & j = 0, \ldots, p-1 \end{array} \tag{12.10}$$

where $A \in \mathbb{R}^{m\times n}$, $A_{ij} = a_{ij}$, which is similar to the dual problem for conic quadratic optimization (see Sec. 12.2.1), except for the addition of dual constraints

$$\left(\overline{C}_j - \sum_{i=0}^{m} y_i \overline{A}_{ij}\right) \in \mathcal{S}_+^{r_j}.$$

Note that the dual of the dual problem is identical to the original primal problem.

12.3.2 Infeasibility for Semidefinite Optimization

In case **MOSEK** finds a problem to be infeasible it reports a certificate of the infeasibility. This works exactly as for linear problems (see Sec. 12.1.2).

Primal Infeasible Problems

If the problem (12.9) is infeasible, **MOSEK** will report a certificate of primal infeasibility: The dual solution reported is a certificate of infeasibility, and the primal solution is undefined.

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

$$\begin{array}{ll}
\text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x \\
\text{subject to} & \\
& A^T y + s_l^x - s_u^x + s_n^x = 0, \\
& \sum_{i=0}^{m-1} y_i \overline{A}_{ij} + \overline{S}_j = 0, \qquad\qquad j = 0,\ldots,p-1 \\
& -y + s_l^c - s_u^c = 0, \\
& s_l^c, s_u^c, s_l^x, s_u^x \geq 0, \\
& s_n^x \in \mathcal{K}^*, \quad \overline{S}_j \in \mathcal{S}_+^{r_j}, \qquad\qquad j = 0,\ldots,p-1
\end{array}$$

such that the objective value is strictly positive.

Dual Infeasible Problems

If the problem (12.10) is infeasible, **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 problem

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

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 the objective value is strictly negative.

12.4 Quadratic and Quadratically Constrained Optimization

A convex quadratic and quadratically constrained optimization problem has the form

$$
\begin{array}{lccccl}
\text{minimize} & & & \frac{1}{2}x^TQ^ox + c^Tx + c^f & & \\
\text{subject to} & l_k^c & \leq & \frac{1}{2}x^TQ^kx + \sum_{j=0}^{n-1} a_{kj}x_j & \leq & u_k^c, \quad k = 0,\ldots,m-1, \\
 & l_j^x & \leq & x_j & \leq & u_j^x, \quad j = 0,\ldots,n-1,
\end{array}
\tag{12.11}
$$

where 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 *[MOSEKApS12]* and in particular *[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 modeller can do a better reformulation than the automatic method because the modeller 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 (12.11) is given by

$$
\begin{array}{ll}
\text{maximize} & (l^c)^Ts_l^c - (u^c)^Ts_u^c + (l^x)^Ts_l^x - (u^x)^Ts_u^x + \frac{1}{2}x^T\left\{\sum_{k=0}^{m-1} y_kQ^k - Q^o\right\}x + c^f \\
\text{subject to} & A^Ty + s_l^x - s_u^x + \left\{\sum_{k=0}^{m-1} y_kQ^k - Q^o\right\}x = c, \\
 & -y + s_l^c - s_u^c = 0, \\
 & s_l^c, s_u^c, s_l^x, s_u^x \geq 0.
\end{array}
\tag{12.12}
$$

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 (12.11) and the dual problem (12.12).

12.4.3 Infeasibility for Quadratic and Quadratically Constrained Optimization

In case **MOSEK** finds a problem to be infeasible it reports a certificate of infeasibility. This works exactly as for linear problems (see Sec. 12.1.2).

Primal Infeasible Problems

If the problem (12.11) with all $Q^k = 0$ is infeasible, **MOSEK** will report a certificate of primal infeasibility. As the constraints are the same as for a linear problem, the certificate of infeasibility is the same as for linear optimization (see Sec. 12.1.2).

Dual Infeasible Problems

If the problem (12.12) with all $Q^k = 0$ is dual infeasible, **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 problem

$$
\begin{array}{lccccc}
\text{minimize} & & & c^T x & & \\
\text{subject to} & \hat{l}^c & \leq & Ax & \leq & \hat{u}^c, \\
 & 0 & \leq & Q^o x & \leq & 0, \\
 & \hat{l}^x & \leq & x & \leq & \hat{u}^x,
\end{array}
$$

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 the objective value is strictly negative.

12.5 General Convex Optimization

The general nonlinear optimizer (which may be available for all or some types of nonlinear problems depending on the interface), solves smooth (twice differentiable) convex nonlinear optimization problems of the form

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

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 objective function.
- $A \in \mathbb{R}^{m \times n}$ 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.
- $f : \mathbb{R}^n \to \mathbb{R}$ is a nonlinear function.
- $g : \mathbb{R}^n \to \mathbb{R}^m$ is a nonlinear vector function.

This means that the i-th constraint has the form

$$l_i^c \leq g_i(x) + \sum_{j=1}^{n} a_{ij} x_j \leq u_i^c.$$

The linear term Ax is not included in $g(x)$ since it can be handled much more efficiently as a separate entity when optimizing.

The nonlinear functions f and g must be smooth in all $x \in [l^x; u^x]$. Moreover, $f(x)$ must be a convex function and $g_i(x)$ must satisfy

$$\begin{array}{rcl} -\infty < l_i^c & \Rightarrow & g_i(x) \text{ is concave}, \\ u_i^c < \infty & \Rightarrow & g_i(x) \text{ is convex}, \\ -\infty < l_i^c \leq u_i^c < \infty & \Rightarrow & g_i(x) = 0. \end{array}$$

12.5.1 Duality for General convex Optimization

Similarly to the linear case, **MOSEK** reports dual information in the general nonlinear case. Indeed in this case the Lagrange function is defined by

$$\begin{array}{rcl} L(x, s_l^c, s_u^c, s_l^x, s_u^x) & := & f(x) + c^T x + c^f \\ & & -(s_l^c)^T (g(x) + Ax - l^c) - (s_u^c)^T (u^c - g(x) - Ax) \\ & & -(s_l^x)^T (x - l^x) - (s_u^x)^T (u^x - x), \end{array}$$

and the dual problem is given by

$$\begin{array}{lccc} \text{maximize} & L(x, s_l^c, s_u^c, s_l^x, s_u^x) & & \\ \text{subject to} & \nabla_x L(x, s_l^c, s_u^c, s_l^x, s_u^x)^T & = & 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x \geq 0, & & \end{array}$$

which is equivalent to

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

In this context we use the following definition for scalar functions

$$\nabla f(x) \quad = \quad \left[\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_n} \right],$$

and accordingly for vector functions

$$\nabla g(x) \quad = \quad \left[\begin{array}{c} \nabla g_1(x) \\ \vdots \\ \nabla g_m(x) \end{array} \right].$$

CHAPTER

THIRTEEN

THE OPTIMIZERS FOR CONTINUOUS PROBLEMS

The most essential part of **MOSEK** are the optimizers. This chapter describes the optimizers for the class of *continuous problems* without integer variables, that is:

- linear problems,
- conic problems (quadratic and semidefinite),
- general convex problems.

MOSEK offers an interior-point optimizer for each class of problems and also a simplex optimizer for linear problems. 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 *[AGMX96]*.

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*. The two most time-consuming steps of the presolve are

- the eliminator, and
- the linear dependency check.

Therefore, in some cases it is worthwhile to disable one or both of these.

Numerical issues in the presolve

During the presolve the problem is reformulated so that it hopefully solves faster. However, in rare cases the presolved problem may be harder to solve 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 modelling 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 quadratic 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 Using Multiple Threads in an Optimizer

Multithreading in interior-point optimizers

The interior-point optimizers in **MOSEK** have been parallelized. This means that if you solve linear, quadratic, conic, or general convex optimization problem using the interior-point optimizer, you can take advantage of multiple CPU's. By default **MOSEK** will automatically select the number of threads to be employed when solving the problem. However, the maximum number of threads employed can be changed by setting the parameter *MSK_IPAR_NUM_THREADS*. This should never exceed the number of cores on the computer.

The speed-up obtained when using multiple threads is highly problem and hardware dependent, and consequently, it is advisable to compare single threaded and multi threaded performance for the given problem type to determine the optimal settings. For small problems, using multiple threads is not be worthwhile and may even be counter productive because of the additional coordination overhead. Therefore, it may be advantageous to disable multithreading using the parameter *MSK_IPAR_INTPNT_MULTI_THREAD*.

The interior-point optimizer parallelizes big tasks such linear algebra computations.

Thread Safety

The **MOSEK** API is thread-safe provided that a task is only modified or accessed from one thread at any given time. Also accessing two or more separate tasks from threads at the same time is safe. Sharing an environment between threads is safe.

Determinism

The optimizers are run-to-run deterministic which means if a problem is solved twice on the same computer using the same parameter setting and exactly the same input then exactly the same results is obtained. One restriction is that no time limits must be imposed because the time taken to perform an operation on a computer is dependent on many factors such as the current workload.

13.3 Linear Optimization

13.3.1 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.

13.3.2 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

$$\begin{array}{lccl} \mbox{minimize} & c^T x & & \\ \mbox{subject to} & Ax & = & b, \\ & x \geq 0. & & \end{array} \tag{13.1}$$

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

$$\begin{aligned} \left\| A\frac{x^k}{\tau^k} - b \right\|_\infty &\leq \epsilon_p (1 + \|b\|_\infty), \\ \left\| A^T\frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c \right\|_\infty &\leq \epsilon_d (1 + \|c\|_\infty), \text{ and} \\ \min\left(\frac{(x^k)^T s^k}{(\tau^k)^2}, \left|\frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k}\right|\right) &\leq \epsilon_g \max\left(1, \frac{\min(|c^T x^k|, |b^T y^k|)}{\tau^k}\right), \end{aligned} \tag{13.5}$$

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

$$-\epsilon_i c^T x^k > \frac{\|c\|_\infty}{\max(1, \|b\|_\infty)} \left\| A x^k \right\|_\infty$$

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 $\left\| A x^k \right\|_\infty = 0$; then x^k is an exact certificate of dual infeasibility. Next assume that this is not the case, i.e.

$$\left\| A x^k \right\|_\infty > 0,$$

and define

$$\bar{x} := \epsilon_i \frac{\max(1, \|b\|_\infty)}{\|A x^k\|_\infty \|c\|_\infty} x^k.$$

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 and near optimality

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.

In some cases the interior-point method terminates having found a solution not too far from meeting the optimality condition (13.5). A solution is defined as *near optimal* if scaling the termination tolerances ε_p, ε_d, ε_g and ε_g by the same factor $\varepsilon_n \in [1.0, +\infty]$ makes the condition (13.5) satisfied. A near optimal solution is therefore of lower quality but still potentially valuable. If for instance the solver stalls, i.e. it can make no more significant progress towards the optimal solution, a near optimal solution could be available and be good enough for the user. Near infeasibility certificates are defined similarly. The value of ε_n can be adjusted with the parameter *MSK_DPAR_INTPNT_CO_TOL_NEAR_REL* .

The basis identification discussed in Sec. 13.3.2 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:

$$
\begin{array}{lccc}
\text{minimize} & x+y & & \\
\text{subject to} & x+y & = & 1, \\
 & x, y \geq 0. & &
\end{array}
$$

It is easy to see that all feasible solutions are also optimal. In particular, there are two basic solutions, namely

$$
\begin{array}{rcl}
(x_1^*, y_1^*) & = & (1,0), \\
(x_2^*, y_2^*) & = & (0,1).
\end{array}
$$

The interior point algorithm will actually converge to the center of the optimal set, i.e. to $(x^*, y^*) = (1/2, 1/2)$ (to see this in **MOSEK** deactivate *Presolve*).

In practice, when the algorithm gets close to the optimal solution, it is possible to construct in polynomial time an initial basis for the simplex algorithm from the current interior point solution. This basis is used to warm-start the simplex algorithm that will provide the optimal basic solution. In most cases the constructed basis is optimal, or very few iterations are required by the simplex algorithm to make it optimal and hence the final *clean-up* phase be short. However, for some cases of ill-conditioned problems the additional simplex clean up phase may take of lot a time.

By default **MOSEK** performs a basis identification. However, if a basic solution is not needed, the basis identification procedure can be turned off. The parameters

- *MSK_IPAR_INTPNT_BASIS*,
- *MSK_IPAR_BI_IGNORE_MAX_ITER*, and
- *MSK_IPAR_BI_IGNORE_NUM_ERROR*

control when basis identification is performed.

The type of simplex algorithm to be used (primal/dual) can be tuned with the parameter *MSK_IPAR_BI_CLEAN_OPTIMIZER*, and the maximum number of iterations can be set with *MSK_IPAR_BI_MAX_ITERATIONS*.

Finally, it should be mentioned that there is no guarantee on which basic solution will be returned.

The Interior-point Log

Below is a typical log output from the interior-point optimizer:

```
Optimizer  - threads                : 1
Optimizer  - solved problem         : the dual
Optimizer  - Constraints            : 2
Optimizer  - Cones                  : 0
Optimizer  - Scalar variables       : 6                 conic                  : 0
Optimizer  - Semi-definite variables: 0                 scalarized             : 0
Factor     - setup time             : 0.00              dense det. time        : 0.00
Factor     - ML order time          : 0.00              GP order time          : 0.00
Factor     - nonzeros before factor : 3                 after factor           : 3
Factor     - dense dim.             : 0                 flops                  : 7.00e+001
ITE PFEAS    DFEAS    GFEAS    PRSTATUS   POBJ               DOBJ               MU       TIME
0   1.0e+000 8.6e+000 6.1e+000 1.00e+000  0.000000000e+000  -2.208000000e+003 1.0e+000 0.00
1   1.1e+000 2.5e+000 1.6e-001 0.00e+000  -7.901380925e+003 -7.394611417e+003 2.5e+000 0.00
2   1.4e-001 3.4e-001 2.1e-002 8.36e-001  -8.113031650e+003 -8.055866001e+003 3.3e-001 0.00
3   2.4e-002 5.8e-002 3.6e-003 1.27e+000  -7.777530698e+003 -7.766471080e+003 5.7e-002 0.01
4   1.3e-004 3.2e-004 2.0e-005 1.08e+000  -7.668323435e+003 -7.668207177e+003 3.2e-004 0.01
5   1.3e-008 3.2e-008 2.0e-009 1.00e+000  -7.668000027e+003 -7.668000015e+003 3.2e-008 0.01
6   1.3e-012 3.2e-012 2.0e-013 1.00e+000  -7.667999994e+003 -7.667999994e+003 3.2e-012 0.01
```

The first line displays the number of threads used by the optimizer and the second line tells that the optimizer chose to solve the dual problem rather than the primal problem. The next line displays 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. 13.3.2 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.

13.3.3 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. 13.3.1 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
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).

13.4 Conic Optimization

For conic optimization problems only an interior-point type optimizer is available.

13.4.1 The Interior-point optimizer

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

$$\begin{array}{lccc} \text{maximize} & b^T y & & \\ \text{subject to} & A^T y + s & = & c, \\ & x \in \mathcal{K}^* & & \end{array} \tag{13.7}$$

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

$$\begin{array}{rcl}
Ax^* & = & 0, \\
A^T y^* + s^* & = & 0, \\
-c^T x^* + b^T y^* & = & \kappa^*, \\
x^* & \in & \mathcal{K}, \\
s^* & \in & \mathcal{K}^*.
\end{array}$$

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]*.

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:

$$\begin{array}{rcl}
\rho_p^k & = & \arg\min_\rho \left\{ \rho \mid \left\| A\frac{x^k}{\tau^k} - b \right\|_\infty \leq \rho\varepsilon_p (1 + \|b\|_\infty) \right\}, \\
\rho_d^k & = & \arg\min_\rho \left\{ \rho \mid \left\| A^T\frac{y^k}{\tau^k} + \frac{s^k}{\tau^k} - c \right\|_\infty \leq \rho\varepsilon_d (1 + \|c\|_\infty) \right\}, \\
\rho_g^k & = & \arg\min_\rho \left\{ \rho \mid \left(\frac{(x^k)^T s^k}{(\tau^k)^2}, \left| \frac{c^T x^k}{\tau^k} - \frac{b^T y^k}{\tau^k} \right| \right) \leq \rho\varepsilon_g \max\left(1, \frac{\min\left(\left|c^T x^k\right|, \left|b^T y^k\right|\right)}{\tau^k}\right) \right\}, \\
\rho_{pi}^k & = & \arg\min_\rho \left\{ \rho \mid \left\| A^T y^k + s^k \right\|_\infty \leq \rho\varepsilon_i b^T y^k,\ b^T y^k > 0 \right\} \text{ and} \\
\rho_{di}^k & = & \arg\min_\rho \left\{ \rho \mid \left\| A x^k \right\|_\infty \leq -\rho\varepsilon_i c^T x^k,\ c^T x^k < 0 \right\}.
\end{array}$$

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

$$\left\| A x^k \right\|_\infty \leq -\varepsilon_i c^T x^k \text{ and } -c^T x^k > 0$$

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.

Adjusting optimality criteria and near optimality

It is possible to adjust the tolerances ε_p, ε_d, ε_g and ε_i using parameters; see table for details.

Table 13.2: Parameters employed in termination criterion

ToleranceParameter	name
ε_p	*MSK_DPAR_INTPNT_CO_TOL_PFEAS*
ε_d	*MSK_DPAR_INTPNT_CO_TOL_DFEAS*
ε_g	*MSK_DPAR_INTPNT_CO_TOL_REL_GAP*
ε_i	*MSK_DPAR_INTPNT_CO_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.

In some cases the interior-point method terminates having found a solution not too far from meeting the optimality condition (13.11). A solution is defined as *near optimal* if scaling the termination tolerances ε_p, ε_d, ε_g and ε_g by the same factor $\varepsilon_n \in [1.0, +\infty]$ makes the condition (13.11) satisfied. A near optimal solution is therefore of lower quality but still potentially valuable. If for instance the solver stalls, i.e. it can make no more significant progress towards the optimal solution, a near optimal solution could be available and be good enough for the user. Near infeasibility certificates are defined similarly. The value of ε_n can be adjusted with the parameter *MSK_DPAR_INTPNT_CO_TOL_NEAR_REL* .

To conclude the discussion in this section, relaxing the termination criterion is usually not worthwhile.

The Interior-point Log

Below is a typical log output from the interior-point optimizer:

```
Optimizer  - threads                : 20
Optimizer  - solved problem         : the primal
Optimizer  - Constraints            : 1
Optimizer  - Cones                  : 2
```

```
Optimizer  - Scalar variables       : 6                 conic                 : 6
Optimizer  - Semi-definite variables: 0                 scalarized            : 0
Factor     - setup time             : 0.00              dense det. time       : 0.00
Factor     - ML order time          : 0.00              GP order time         : 0.00
Factor     - nonzeros before factor : 1                 after factor          : 1
Factor     - dense dim.             : 0                 flops                 : 1.70e+01
ITE PFEAS    DFEAS    GFEAS    PRSTATUS   POBJ              DOBJ              MU       TIME
0   1.0e+00  2.9e-01  3.4e+00  0.00e+00   2.414213562e+00   0.000000000e+00   1.0e+00  0.01
1   2.7e-01  7.9e-02  2.2e+00  8.83e-01   6.969257574e-01   -9.685901771e-03  2.7e-01  0.01
2   6.5e-02  1.9e-02  1.2e+00  1.16e+00   7.606090061e-01   6.046141322e-01   6.5e-02  0.01
3   1.7e-03  5.0e-04  2.2e-01  1.12e+00   7.084385672e-01   7.045122560e-01   1.7e-03  0.01
4   1.4e-08  4.2e-09  4.9e-08  1.00e+00   7.071067941e-01   7.071067599e-01   1.4e-08  0.01
```

The first line displays the number of threads used by the optimizer and the second line tells that the optimizer chose to solve the dual problem rather than the primal problem. The next line displays 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. 13.4.1 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`: $\left\|A^Ty^k + s^k - c\tau^k\right\|_\infty$. The numbers in this column should converge monotonically towards zero but may stall at low level due to rounding errors.
- `GFEAS`: $| -c^Tx^k + b^Ty^k - \kappa^k|$. 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^Tx^k/τ^k. An estimate for the primal objective value.
- `DOBJ`: b^Ty^k/τ^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).

13.5 Nonlinear Convex Optimization

13.5.1 The Interior-point Optimizer

For general convex optimization problems an interior-point type optimizer is available. The interior-point optimizer is an implementation of the homogeneous and self-dual algorithm. For a detailed description of the algorithm, please see *[AY98]*, *[AY99]*.

The Convexity Requirement

Continuous nonlinear problems are required to be convex. For quadratic problems **MOSEK** tests this requirement before optimizing. Specifying a non-convex problem results in an error message.

The following parameters are available to control the convexity check:

- *MSK_IPAR_CHECK_CONVEXITY*: Turn convexity check on/off.
- *MSK_DPAR_CHECK_CONVEXITY_REL_TOL*: Tolerance for convexity check.
- *MSK_IPAR_LOG_CHECK_CONVEXITY*: Turn on more log information for debugging.

The Differentiability Requirement

The nonlinear optimizer in **MOSEK** requires both first order and second order derivatives. This of course implies care should be taken when solving problems involving non-differentiable functions.

For instance, the function

$$f(x) = x^2$$

is differentiable everywhere whereas the function

$$f(x) = \sqrt{x}$$

is only differentiable for $x > 0$. In order to make sure that **MOSEK** evaluates the functions at points where they are differentiable, the function domains must be defined by setting appropriate variable bounds.

In general, if a variable is not ranged **MOSEK** will only evaluate that variable at points strictly within the bounds. Hence, imposing the bound

$$x \geq 0$$

in the case of $\sqrt{x}$ is sufficient to guarantee that the function will only be evaluated in points where it is differentiable.

However, if a function is defined on a closed range, specifying the variable bounds is not sufficient. Consider the function

$$f(x) = \frac{1}{x} + \frac{1}{1-x}. \tag{13.12}$$

In this case the bounds

$$0 \leq x \leq 1$$

will not guarantee that **MOSEK** only evaluates the function for x strictly between 0 and 1 . To force **MOSEK** to strictly satisfy both bounds on ranged variables set the parameter *MSK_IPAR_INTPNT_STARTING_POINT* to *MSK_STARTING_POINT_SATISFY_BOUNDS* .

For efficiency reasons it may be better to reformulate the problem than to force **MOSEK** to observe ranged bounds strictly. For instance, (13.12) can be reformulated as follows

$$\begin{array}{rcl} f(x) & = & \frac{1}{x} + \frac{1}{y} \\ 0 & = & 1 - x - y \\ 0 & \leq & x \\ 0 & \leq & y. \end{array}$$

Interior-point Termination Criteria

The parameters controlling when the general convex interior-point optimizer terminates are shown in Table 13.3.

Table 13.3: Parameters employed in termination criteria.

Parameter name	Purpose
MSK_DPAR_INTPNT_NL_TOL_PFEAS	Controls primal feasibility
MSK_DPAR_INTPNT_NL_TOL_DFEAS	Controls dual feasibility
MSK_DPAR_INTPNT_NL_TOL_REL_GAP	Controls relative gap
MSK_DPAR_INTPNT_TOL_INFEAS	Controls when the problem is declared infeasible
MSK_DPAR_INTPNT_NL_TOL_MU_RED	Controls when the complementarity is reduced enough

CHAPTER

FOURTEEN

THE OPTIMIZER FOR MIXED-INTEGER PROBLEMS

A problem is a mixed-integer optimization problem when one or more of the variables are constrained to be integer valued. Readers unfamiliar with integer optimization are recommended to consult some relevant literature, e.g. the book *[Wol98]* by Wolsey.

14.1 The Mixed-integer Optimizer Overview

MOSEK can solve mixed-integer

- linear,
- quadratic and quadratically constrained, and
- conic quadratic

problems, at least as long as they do not contain both quadratic objective or constraints and conic constraints at the same time. The mixed-integer optimizer is specialized for solving linear and conic optimization problems. Pure quadratic and quadratically constrained problems are automatically converted to conic form.

By default the mixed-integer optimizer is run-to-run deterministic. This means that if a problem is solved twice on the same computer with identical parameter settings and no time limit then the obtained solutions will be identical. If a time limit is set then this may not be case since the time taken to solve a problem is not deterministic. The mixed-integer optimizer is parallelized i.e. it can exploit multiple cores during the optimization.

The solution process can be split into these phases:

1. **Presolve:** See Sec. 13.1.
2. **Cut generation:** Valid inequalities (cuts) are added to improve the lower bound.
3. **Heuristic:** Using heuristics the optimizer tries to guess a good feasible solution. Heuristics can be controlled by the parameter *MSK_IPAR_MIO_HEURISTIC_LEVEL* .
4. **Search:** The optimal solution is located by branching on integer variables.

14.2 Relaxations and bounds

It is important to understand that, in a worst-case scenario, the time required to solve integer optimization problems grows exponentially with the size of the problem (solving mixed-integer problems is NP-hard). For instance, a problem with n binary variables, may require time proportional to 2^n . The value of 2^n is huge even for moderate values of n.

In practice this implies that the focus should be on computing a near-optimal solution quickly rather than on locating an optimal solution. Even if the problem is only solved approximately, it is important to know how far the approximate solution is from an optimal one. In order to say something about the quality of an approximate solution the concept of *relaxation* is important.

Consider for example a mixed-integer optimization problem

$$\begin{array}{rcllcl} z^* & = & \text{minimize} & c^T x & & \\ & & \text{subject to} & Ax & = & b, \\ & & & x \geq 0 & & \\ & & & x_j \in \mathbb{Z}, & & \forall j \in \mathcal{J}. \end{array} \tag{14.1}$$

It has the continuous relaxation

$$\begin{array}{rcllcl} \underline{z} & = & \text{minimize} & c^T x & & \\ & & \text{subject to} & Ax & = & b, \\ & & & x \geq 0 & & \end{array} \tag{14.2}$$

obtained simply by ignoring the integrality restrictions. The relaxation is a continuous problem, and therefore much faster to solve to optimality with a linear (or, in the general case, conic) optimizer. We call the optimal value $\underline{z}$ the *objective bound*. The objective bound $\underline{z}$ normally increases during the solution search process when the continuous relaxation is gradually refined.

Moreover, if $\hat{x}$ is any feasible solution to (14.1) and

$$\bar{z} := c^T \hat{x}$$

then

$$\underline{z} \leq z^* \leq \bar{z}.$$

These two inequalities allow us to estimate the quality of the integer solution: it is no further away from the optimum than $\bar{z} - \underline{z}$ in terms of the objective value. Whenever a mixed-integer problem is solved **MOSEK** reports this lower bound so that the quality of the reported solution can be evaluated.

14.3 Termination Criterion

In general, it is time consuming to find an exact feasible and optimal solution to an integer optimization problem, though in many practical cases it may be possible to find a sufficiently good solution. The issue of terminating the mixed-integer optimizer is rather delicate and the user has numerous possibilities of influencing it with various parameters. The mixed-integer optimizer employs a relaxed feasibility and optimality criterion to determine when a satisfactory solution is located.

A candidate solution that is feasible for the continuous relaxation is said to be an *integer feasible solution* if the criterion

$$\min(x_j - \lfloor x_j \rfloor, \lceil x_j \rceil - x_j) \leq \delta_1 \quad \forall j \in \mathcal{J}$$

is satisfied, meaning that x_j is at most δ_1 from the nearest integer.

Whenever the integer optimizer locates an integer feasible solution it will check if the criterion

$$\bar{z} - \underline{z} \leq \max(\delta_2, \delta_3 \max(10^{-10}, |\bar{z}|))$$

is satisfied. If this is the case, the integer optimizer terminates and reports the integer feasible solution as an optimal solution. If an optimal solution cannot be located after the time specified by the parameter `MSK_DPAR_MIO_DISABLE_TERM_TIME` (in seconds), it may be advantageous to relax the termination criteria, and they become replaced with

$$\bar{z} - \underline{z} \leq \max(\delta_4, \delta_5 \max(10^{-10}, |\bar{z}|)).$$

Any solution satisfying those will now be reported as **near optimal** and the solver will be terminated (note that since this criterion depends on timing, the optimizer will not be run to run deterministic).

All the δ tolerances discussed above can be adjusted using suitable parameters — see Table 14.1.

Table 14.1: Tolerances for the mixed-integer optimizer.

Tolerance	Parameter name
δ_1	*MSK_DPAR_MIO_TOL_ABS_RELAX_INT*
δ_2	*MSK_DPAR_MIO_TOL_ABS_GAP*
δ_3	*MSK_DPAR_MIO_TOL_REL_GAP*
δ_4	*MSK_DPAR_MIO_NEAR_TOL_ABS_GAP*
δ_5	*MSK_DPAR_MIO_NEAR_TOL_REL_GAP*

In Table 14.2 some other common parameters affecting the integer optimizer termination criterion are shown. Please note that if the effect of a parameter is delayed, the associated termination criterion is applied only after some time, specified by the *MSK_DPAR_MIO_DISABLE_TERM_TIME* parameter.

Table 14.2: Other parameters affecting the integer optimizer termination criterion.

Parameter name	Delayed	Explanation
MSK_IPAR_MIO_MAX_NUM_BRANCHES	Yes	Maximum number of branches allowed.
MSK_IPAR_MIO_MAX_NUM_RELAXS	Yes	Maximum number of relaxations allowed.
MSK_IPAR_MIO_MAX_NUM_SOLUTIONS	Yes	Maximum number of feasible integer solutions allowed.

14.4 Speeding Up the Solution Process

As mentioned previously, in many cases it is not possible to find an optimal solution to an integer optimization problem in a reasonable amount of time. Some suggestions to reduce the solution time are:

- Relax the termination criterion: In case the run time is not acceptable, the first thing to do is to relax the termination criterion — see Sec. 14.3 for details.
- Specify a good initial solution: In many cases a good feasible solution is either known or easily computed using problem-specific knowledge. If a good feasible solution is known, it is usually worthwhile to use this as a starting point for the integer optimizer.
- Improve the formulation: A mixed-integer optimization problem may be impossible to solve in one form and quite easy in another form. However, it is beyond the scope of this manual to discuss good formulations for mixed-integer problems. For discussions on this topic see for example *[Wol98]*.

14.5 Understanding Solution Quality

To determine the quality of the solution one should check the following:

- The problem status and solution status returned by **MOSEK**, as well as constraint violations in case of suboptimal solutions.
- The *optimality gap* defined as

  $$\epsilon = |(\text{objective value of feasible solution}) - (\text{objective bound})| = |\bar{z} - \underline{z}|.$$

  which measures how much the located solution can deviate from the optimal solution to the problem. The optimality gap can be retrieved through the information item *MSK_DINF_MIO_OBJ_ABS_GAP*. Often it is more meaningful to look at the relative optimality gap normalized against the magnitude of the solution.

  $$\epsilon_{\text{rel}} = \frac{|\bar{z} - \underline{z}|}{\max(10^{-10}, |\bar{z}|)}.$$

  The relative optimality gap is available in *MSK_DINF_MIO_OBJ_REL_GAP*.

14.6 The Optimizer Log

Below is a typical log output from the mixed-integer optimizer:

```
Presolved problem: 6573 variables, 35728 constraints, 101258 non-zeros
Presolved problem: 0 general integer, 4294 binary, 2279 continuous
Clique table size: 1636
BRANCHES RELAXS   ACT_NDS  DEPTH    BEST_INT_OBJ         BEST_RELAX_OBJ       REL_GAP(%)  TIME
0        1        0        0        NA                   1.8218819866e+07     NA          1.6
0        1        0        0        1.8331557950e+07     1.8218819866e+07     0.61        3.5
0        1        0        0        1.8300507546e+07     1.8218819866e+07     0.45        4.3
Cut generation started.
0        2        0        0        1.8300507546e+07     1.8218819866e+07     0.45        5.3
Cut generation terminated. Time = 1.43
0        3        0        0        1.8286893047e+07     1.8231580587e+07     0.30        7.5
15       18       1        0        1.8286893047e+07     1.8231580587e+07     0.30        10.5
31       34       1        0        1.8286893047e+07     1.8231580587e+07     0.30        11.1
51       54       1        0        1.8286893047e+07     1.8231580587e+07     0.30        11.6
91       94       1        0        1.8286893047e+07     1.8231580587e+07     0.30        12.4
171      174      1        0        1.8286893047e+07     1.8231580587e+07     0.30        14.3
331      334      1        0        1.8286893047e+07     1.8231580587e+07     0.30        17.9

[ ... ]

Objective of best integer solution : 1.825846762609e+07
Best objective bound               : 1.823311032986e+07
Construct solution objective       : Not employed
Construct solution # roundings     : 0
User objective cut value           : 0
Number of cuts generated           : 117
  Number of Gomory cuts            : 108
  Number of CMIR cuts              : 9
Number of branches                 : 4425
Number of relaxations solved       : 4410
Number of interior point iterations: 25
Number of simplex iterations       : 221131
```

The first lines contain a summary of the problem as seen by the optimizer. This is followed by the iteration log. The columns have the following meaning:

- `BRANCHES`: Number of branches generated.
- `RELAXS`: Number of relaxations solved.
- `ACT_NDS`: Number of active branch bound nodes.
- `DEPTH`: Depth of the recently solved node.
- `BEST_INT_OBJ`: The best integer objective value, $\bar{z}$.
- `BEST_RELAX_OBJ`: The best objective bound, $\underline{z}$.
- `REL_GAP(%)`: Relative optimality gap, $100\% \cdot \epsilon_{\text{rel}}$
- `TIME`: Time (in seconds) from the start of optimization.

Following that a summary of the optimization process is printed.

CHAPTER

FIFTEEN

ADDITIONAL FEATURES

In this section we describe additional features and tools which enable more detailed analysis of optimization problems with **MOSEK**.

15.1 Problem Analyzer

The problem analyzer prints a detailed survey of the

- linear constraints and objective
- quadratic constraints
- conic constraints
- variables

of the model.

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 produces output similar to the one below (this is the problem survey of the `aflow30a` problem from the MIPLIB 2003 collection).

```
Analyzing the problem

Constraints                Bounds                     Variables
 upper bd:       421        ranged  : all              cont:       421
 fixed   :        58                                   bin :       421

-------------------------------------------------------------------------------

Objective, min cx
   range: min |c|: 0.00000   min |c|>0: 11.0000     max |c|: 500.000
 distrib:        |c|        vars
                   0         421
           [11, 100)         150
          [100, 500]         271

-------------------------------------------------------------------------------

Constraint matrix A has
       479 rows (constraints)
       842 columns (variables)
      2091 (0.518449%) nonzero entries (coefficients)

Row nonzeros, A_i
   range: min A_i: 2 (0.23753%)    max A_i: 34 (4.038%)
```

```
 distrib:        A_i        rows       rows%        acc%
                   2         421       87.89       87.89
             [8, 15]          20        4.18       92.07
            [16, 31]          30        6.26       98.33
            [32, 34]           8        1.67      100.00

Column nonzeros, A|j
   range: min A|j: 2 (0.417537%)    max A|j: 3 (0.626305%)
 distrib:        A|j        cols       cols%        acc%
                   2         435       51.66       51.66
                   3         407       48.34      100.00

A nonzeros, A(ij)
   range: min |A(ij)|: 1.00000     max |A(ij)|: 100.000
 distrib:      A(ij)      coeffs
             [1, 10)        1670
           [10, 100]         421

-------------------------------------------------------------------------------

Constraint bounds, lb <= Ax <= ub
 distrib:        |b|            lbs             ubs
                   0                            421
             [1, 10]             58              58

Variable bounds, lb <= x <= ub
 distrib:        |b|            lbs             ubs
                   0            842
             [1, 10)                            421
           [10, 100]                            421

-------------------------------------------------------------------------------
```

The survey is divided into six different sections, each described below. To keep the presentation short with focus on key elements. The analyzer generally attempts to display information on issues relevant for the current model only: e.g., if the model does not have any conic constraints (this is the case in the example above) or any integer variables, those parts of the analysis will not appear.

General Characteristics

The first part of the survey consists of a brief summary of the model's linear and quadratic constraints (indexed by i) and variables (indexed by j). The summary is divided into three subsections:

`Constraints`

- `upper bd` The number of upper bounded constraints, $\sum_{j=0}^{n-1} a_{ij}x_j \leq u_i^c$
- `lower bd` The number of lower bounded constraints, $l_i^c \leq \sum_{j=0}^{n-1} a_{ij}x_j$
- `ranged` The number of ranged constraints, $l_i^c \leq \sum_{j=0}^{n-1} a_{ij}x_j \leq u_i^c$
- `fixed` The number of fixed constraints, $l_i^c = \sum_{j=0}^{n-1} a_{ij}x_j = u_i^c$
- `free` The number of free constraints

`Bounds`

- `upper bd` The number of upper bounded variables, $x_j \leq u_j^x$

- `lower bd` The number of lower bounded variables, $l_k^x \leq x_j$
- `ranged` The number of ranged variables, $l_k^x \leq x_j \leq u_j^x$
- `fixed` The number of fixed variables, $l_k^x = x_j = u_j^x$
- `free` The number of free variables

`Variables`

- `cont` The number of continuous variables, $x_j \in \mathbb{R}$
- `bin` The number of binary variables, $x_j \in \{0, 1\}$
- `int` The number of general integer variables, $x_j \in \mathbb{Z}$

Only constraints, bounds and domains actually in the model will be reported on; if all entities in a section turn out to be of the same kind, the number will be replaced by `all` for brevity.

Objective

The second part of the survey focuses on (the linear part of) the objective, summarizing the optimization sense and the coefficients' absolute value range and distribution. The number of 0 (zero) coefficients is singled out (if any such variables are in the problem).

The range is displayed using three terms:

- `min |c|` The minimum absolute value among all coeffecients
- `min |c|>0` The minimum absolute value among the nonzero coefficients
- `max |c|` The maximum absolute value among the coefficients

If some of these extrema turn out to be equal, the display is shortened accordingly:

- If `min |c|` is greater than zero, the `min |c|>0` term is obsolete and will not be displayed
- If only one or two different coefficients occur this will be displayed using `all` and an explicit listing of the coefficients

The absolute value distribution is displayed as a table summarizing the numbers by orders of magnitude (with a ratio of 10). Again, the number of variables with a coefficient of 0 (if any) is singled out. Each line of the table is headed by an interval (half-open intervals including their lower bounds), and is followed by the number of variables with their objective coefficient in this interval. Intervals with no elements are skipped.

Linear Constraints

The third part of the survey displays information on the nonzero coefficients of the linear constraint matrix.

Following a brief summary of the matrix dimensions and the number of nonzero coefficients in total, three sections provide further details on how the nonzero coefficients are distributed by row-wise count (`A_i`), by column-wise count (`A|j`), and by absolute value (`|A(ij)|`). Each section is headed by a brief display of the distribution's range (`min` and `max`), and for the row/column-wise counts the corresponding densities are displayed too (in parentheses).

The distribution tables single out three particularly interesting counts: zero, one, and two nonzeros per row/column; the remaining row/column nonzeros are displayed by orders of magnitude (ratio 2). For each interval the relative and accumulated relative counts are also displayed.

Note that constraints may have both linear and quadratic terms, but the empty rows and columns reported in this part of the survey relate to the linear terms only. If empty rows and/or columns are found in the linear constraint matrix, the problem is analyzed further in order to determine if the corresponding constraints have any quadratic terms or the corresponding variables are used in conic or quadratic constraints.

The distribution of the absolute values, `|A(ij)|`, is displayed just as for the objective coefficients described above.

Constraint and Variable Bounds

The fourth part of the survey displays distributions for the absolute values of the finite lower and upper bounds for both constraints and variables. The number of bounds at 0 is singled out and, otherwise, displayed by orders of magnitude (with a ratio of 10).

Quadratic Constraints

The fifth part of the survey displays distributions for the nonzero elements in the gradient of the quadratic constraints, i.e. the nonzero row counts for the column vectors Qx . The table is similar to the tables for the linear constraints' nonzero row and column counts described in the survey's third part.

Quadratic constraints may also have a linear part, but that will be included in the linear constraints survey; this means that if a problem has one or more pure quadratic constraints, part three of the survey will report the number of linear constraint rows with 0 (zero) nonzeros. Likewise, variables that appear in quadratic terms only will be reported as empty columns (0 nonzeros) in the linear constraint report.

Conic Constraints

The last part of the survey summarizes the model's conic constraints. For each of the two types of cones, quadratic and rotated quadratic, the total number of cones are reported, and the distribution of the cones' dimensions are displayed using intervals. Cones dimensions of 2, 3, and 4 are singled out.

15.2 Analyzing Infeasible Problems

When developing and implementing a new optimization model, the first attempts will often be either infeasible, due to specification of inconsistent constraints, or unbounded, if important constraints have been left out.

In this section we will

- go over an example demonstrating how to locate infeasible constraints using the **MOSEK** infeasibility report tool,
- discuss in more general terms which properties may cause infeasibilities, and
- present the more formal theory of infeasible and unbounded problems.

Furthermore, Sec. 15.2.7 contains a discussion on a specific method for repairing infeasibility problems where infeasibilities are caused by model parameters rather than errors in the model or the implementation.

15.2.1 Example: Primal Infeasibility

A problem is said to be *primal infeasible* if no solution exists that satisfies all the constraints of the problem.

As an example of a primal infeasible problem consider the problem of 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. 15.1.

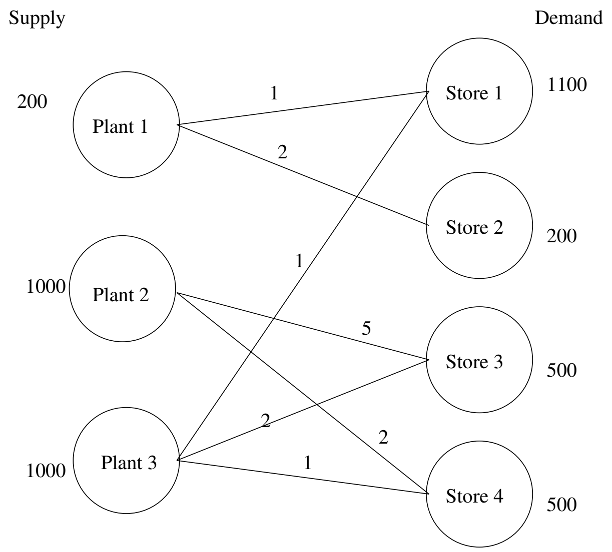

Fig. 15.1: Supply, demand and cost of transportation.

The problem represented in Fig. 15.1 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:

$$
\begin{array}{lccccccccccccccc}
\text{minimize} & x_{11} & + & 2x_{12} & + & 5x_{23} & + & 2x_{24} & + & x_{31} & + & 2x_{33} & + & x_{34} & & \\
\text{subject to} & x_{11} & + & x_{12} & & & & & & & & & & & \leq & 200, \\
 & & & & & x_{23} & + & x_{24} & & & & & & & \leq & 1000, \\
 & & & & & & & & & x_{31} & + & x_{33} & + & x_{34} & \leq & 1000, \\
 & x_{11} & & & & & & & + & x_{31} & & & & & = & 1100, \\
 & & & x_{12} & & & & & & & & & & & = & 200, \\
 & & & & & x_{23} & + & & & & & x_{33} & & & = & 500, \\
 & & & & & & & x_{24} & + & & & & & x_{34} & = & 500, \\
 & x_{ij} \geq 0. & & & & & & & & & & & & & &
\end{array}
\tag{15.1}
$$

Solving problem (15.1) using **MOSEK** will result in a solution, a solution status and a problem status. Among the log output from the execution of **MOSEK** on the above problem are the lines:

```
Basic solution
Problem status  : PRIMAL_INFEASIBLE
Solution status : PRIMAL_INFEASIBLE_CER
```

The first line indicates that the problem status is primal infeasible. The second line says that a *certificate of the infeasibility* was found. The certificate is returned in place of the solution to the problem.

15.2.2 Locating the cause of Primal Infeasibility

Usually a primal infeasible problem status is caused by a mistake in formulating the problem and therefore the question arises: *What is the cause of the infeasible status?* When trying to answer this question, it is often advantageous to follow these steps:

- 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.

- Consider whether your problem has some 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.

If the problem is still primal infeasible, some of the constraints must be relaxed or removed completely. The **MOSEK** infeasibility report (Sec. 15.2.4) may assist you in finding the constraints causing the infeasibility.

Possible ways of relaxing your problem nclude:

- Increasing (decreasing) upper (lower) bounds on variables and constraints.
- Removing suspected constraints from the problem.

Returning to the transportation example, we discover that removing the fifth constraint

$$x_{12} = 200$$

makes the problem feasible.

15.2.3 Locating the Cause of Dual Infeasibility

A problem may also be *dual infeasible*. In this case the primal problem is often unbounded, meaning that feasbile solutions exists such that the objective tends towards infinity. An example of a dual infeasible and primal unbounded problem is:

$$\begin{array}{ll} \text{minimize} & x_1 \\ \text{subject to} & x_1 \leq 5. \end{array}$$

To resolve a dual infeasibility the primal problem must be made more restricted by

- Adding upper or lower bounds on variables or constraints.
- Removing variables.
- Changing the objective.

A cautionary note

The problem

$$\begin{array}{lll} \text{minimize} & 0 & \\ \text{subject to} & 0 \leq x_1, & \\ & x_j \leq x_{j+1}, & j = 1, \ldots, n-1, \\ & x_n \leq -1 & \end{array}$$

is clearly infeasible. Moreover, if any one of the constraints is dropped, then the problem becomes feasible.

This illustrates the worst case scenario where all, or at least a significant portion of the constraints are involved in causing infeasibility. Hence, it may not always be easy or possible to pinpoint a few constraints responsible for infeasibility.

15.2.4 The Infeasibility Report

MOSEK includes functionality for diagnosing the cause of a primal or a dual infeasibility. It can be turned on by setting the *MSK_IPAR_INFEAS_REPORT_AUTO* to *MSK_ON*. This causes **MOSEK** to print a report on variables and constraints involved in the infeasibility.

The *MSK_IPAR_INFEAS_REPORT_LEVEL* parameter controls the amount of information presented in the infeasibility report. The default value is 1.

Example: Primal Infeasibility

We will keep working with the problem (15.1) written in LP format:

Listing 15.1: The code for problem (15.1).

```
\
\ An example of an infeasible linear problem.
\
minimize
 obj: + 1 x11 + 2 x12
      + 5 x23 + 2 x24
      + 1 x31 + 2 x33 + 1 x34
st
  s0: + x11 + x12       <= 200
  s1: + x23 + x24       <= 1000
  s2: + x31 + x33 + x34 <= 1000
  d1: + x11 + x31        = 1100
  d2: + x12              = 200
  d3: + x23 + x33        = 500
  d4: + x24 + x34        = 500
bounds
end
```

Example: Dual Infeasibility

The following problem is dual to (15.1) and therefore it is dual infeasible.

Listing 15.2: The dual of problem (15.1).

```
maximize + 200 y1 + 1000 y2 + 1000 y3 + 1100 y4 + 200 y5 + 500 y6 + 500 y7
subject to
   x11: y1+y4 < 1
   x12: y1+y5 < 2
   x23: y2+y6 < 5
   x24: y2+y7 < 2
   x31: y3+y4 < 1
   x33: y3+y6 < 2
   x34: y3+y7 < 1
bounds
   -inf <= y1 < 0
   -inf <= y2 < 0
   -inf <= y3 < 0
   y4 free
   y5 free
   y6 free
   y7 free
end
```

This can be verified by proving that

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

is a certificate of dual infeasibility (see Sec. 12.1.2) as we can see from this report:

```
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
```

Let y^* denote the reported primal solution. **MOSEK** states

- that the problem is *dual infeasible*,
- that the reported solution is a certificate of dual infeasibility, and
- that the infeasibility measure for y^* is approximately zero.

Since the original objective was maximization, we have that $c^T y^* > 0$. See Sec. 12.1.2 for how to interpret the parameter values in the infeasibility report for a linear program. We see that the variables `y1`, `y3`, `y4`, `y5` and the constraints `x33` and `x34` contribute to infeasibility with non-zero values in the `Activity` column.

One possible strategy to *fix* the infeasibility is to modify the problem so that the certificate of infeasibility becomes invalid. In this case we could do one the following things:

- Add a lower bound on `y3`. This will directly invalidate the certificate of dual infeasibility.
- Increase the object coefficient of `y3`. Changing the coefficients sufficiently will invalidate the inequality $c^T y^* > 0$ and thus the certificate.
- Add lower bounds on `x11` or `x31`. This will directly invalidate the certificate of infeasibility.

Please note that modifying the problem to invalidate the reported certificate does *not* imply that the problem becomes dual 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.

15.2.5 Theory Concerning Infeasible Problems

This section discusses the theory of infeasibility certificates and how **MOSEK** uses a certificate to produce an infeasibility report. In general, **MOSEK** solves the problem

$$\begin{array}{lccccc} \text{minimize} & & & c^T x + c^f & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x \end{array} \tag{15.2}$$

where the corresponding dual problem is

$$\begin{array}{lccc} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c & & \\ & +(l^x)^T s_l^x - (u^x)^T s_u^x + c^f & & \\ \text{subject to} & A^T y + s_l^x - s_u^x & = & c, \\ & -y + s_l^c - s_u^c & = & 0, \\ & s_l^c, s_u^c, s_l^x, s_u^x \leq 0. & & \end{array} \tag{15.3}$$

We use the convension that for any bound that is not finite, the corresponding dual variable is fixed at zero (and thus will have no influence on the dual problem). For example

$$l_j^x = -\infty \quad \Rightarrow \quad (s_l^x)_j = 0$$

15.2.6 The Certificate of Primal Infeasibility

A certificate of primal infeasibility is *any* solution to the homogenized dual problem

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

with a positive objective value. That is, $(s_l^{c*}, s_u^{c*}, s_l^{x*}, s_u^{x*})$ is a certificate of primal infeasibility if

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

and

$$\begin{array}{lcc}
A^T y + s_l^{x*} - s_u^{x*} & = & 0, \\
-y + s_l^{c*} - s_u^{c*} & = & 0, \\
s_l^{c*}, s_u^{c*}, s_l^{x*}, s_u^{x*} \leq 0. & &
\end{array}$$

The well-known *Farkas Lemma* tells us that (15.2) is infeasible if and only if a certificate of primal infeasibility exists.

Let $(s_l^{c*}, s_u^{c*}, s_l^{x*}, s_u^{x*})$ be a certificate of primal infeasibility then

$$(s_l^{c*})_i > 0((s_u^{c*})_i > 0)$$

implies that the lower (upper) bound on the i th constraint is important for the infeasibility. Furthermore,

$$(s_l^{x*})_j > 0((s_u^{x*})_i > 0)$$

implies that the lower (upper) bound on the j th variable is important for the infeasibility.

15.2.7 The certificate of dual infeasibility

A certificate of dual infeasibility is *any* solution to the problem

$$\begin{array}{lccccc}
\text{minimize} & & & c^T x & & \\
\text{subject to} & \bar{l}^c & \leq & Ax & \leq & \bar{u}^c, \\
 & \bar{l}^x & \leq & x & \leq & \bar{u}^x
\end{array}$$

with negative objective value, where we use the definitions

$$\bar{l}_i^c := \left\{ \begin{array}{ll} 0, & l_i^c > -\infty, \\ -\infty, & \text{otherwise}, \end{array} \right\}, \; \bar{u}_i^c := \left\{ \begin{array}{ll} 0, & u_i^c < \infty, \\ \infty, & \text{otherwise}, \end{array} \right\}$$

and

$$\bar{l}_i^x := \left\{ \begin{array}{ll} 0, & l_i^x > -\infty, \\ -\infty, & \text{otherwise}, \end{array} \right\} \quad \text{and} \quad \bar{u}_i^x := \left\{ \begin{array}{ll} 0, & u_i^x < \infty, \\ \infty, & \text{otherwise}. \end{array} \right\}$$

Stated differently, a certificate of dual infeasibility is any x^* such that

$$\begin{array}{ccccc}
 & & c^T x^* & < & 0, \\
\bar{l}^c & \leq & Ax^* & \leq & \bar{u}^c, \\
\bar{l}^x & \leq & x^* & \leq & \bar{u}^x
\end{array} \tag{15.4}$$

The well-known Farkas Lemma tells us that (15.3) is infeasible if and only if a certificate of dual infeasibility exists.

Note that if x^* is a certificate of dual infeasibility then for any j such that

$$x_j^* \leq 0,$$

variable j is involved in the dual infeasibility.

Primal Feasibility Repair

Sec. 15.2.2 discusses how **MOSEK** treats infeasible problems. In particular, it is discussed which information **MOSEK** returns when a problem is infeasible and how this information can be used to pinpoint the cause of the infeasibility.

In this section we discuss how to repair a primal infeasible problem by relaxing the constraints in a controlled way. For the sake of simplicity we discuss the method in the context of linear optimization.

Manual repair

Subsequently we discuss an automatic method for repairing an infeasible optimization problem. However, it should be observed that the best way to repair an infeasible problem usually depends on what the optimization problem models. For instance in many optimization problem it does not make sense to relax the constraints $x \geq 0$ e.g. it is not possible to produce a negative quantity. Hence, whatever automatic method **MOSEK** provides it will never be as good as a method that exploits knowledge about what is being modelled. This implies that it is usually better to remove the underlying cause of infeasibility at the modelling stage.

Indeed consider the example

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

then if we add the equalties together we obtain the implied equality

$$0 = \varepsilon$$

which is infeasible for any $\varepsilon \geq 0$. Here the infeasibility is caused by a linear dependency in the constraint matrix and that the right-hand side does not match if $\varepsilon \geq 0$.

Observe even if the problem is feasible then just a tiny perturbation to the right-hand side will make the problem infeasible. Therefore, even though the problem can be repaired then a much more robust solution is to avoid problems with linear dependent constraints. 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. Note that most network flow models usually is formulated with one linear dependent constraint.

Next consider the problem

$$\begin{array}{lrcl}
\text{minimize} & & & \\
\text{subject to} & x_1 - 0.01x_2 & = & 0 \\
 & x_2 - 0.01x_3 & = & 0 \\
 & x_3 - 0.01x_4 & = & 0 \\
 & x_1 & \geq & -1.0e-9 \\
 & x_1 & \geq & 1.0e-9 \\
 & x_4 & \geq & -1.0e-4
\end{array}$$

Now the **MOSEK** presolve for the sake of efficiency fix variables (and constraints) that has tight bounds where tightness is controlled by the parameter *MSK_DPAR_PRESOLVE_TOL_X*. Since, the bounds

$$-1.0e-9 \leq x_1 \leq 1.0e-9$$

are tight then the **MOSEK** presolve will fix variable x_1 at the mid point between the bounds i.e. at 0. It easy to see that this implies $x_4 = 0$ too which leads to the incorrect conclusion that the problem is infeasible. Observe tiny change of the size 1.0e-9 make the problem switch from feasible to infeasible. Such a problem is inherently unstable and is hard to solve. We normally call such a problem ill-posed.

In general it is recommended to avoid ill-posed problems, but if that is not possible then one solution to this issue is is to reduce the parameter to say *MSK_DPAR_PRESOLVE_TOL_X* to say 1.0e-10. This will at least make sure that the presolve does not make the wrong conclusion.

Automatic Repair

In this section we will describe the idea behind a method that automatically can repair an infeasible probem. The main idea can be described as follows. Consider the linear optimization problem with m constraints and n variables

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

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

$$\begin{array}{lccccl} \text{minimize} & & & p(v_l^c, v_u^c, v_l^x, v_u^x) & & \\ \text{subject to} & l^c & \leq & Ax + v_l^c - v_u^c & \leq & u^c, \\ & l^x & \leq & x + v_l^x - v_u^x & \leq & u^x, \\ & & & v_l^c, v_u^c, v_l^x, v_u^x \geq 0 & & \end{array} \tag{15.5}$$

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 penalize 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),

then 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 relaxation and therefore the problem (15.5) keeps the amount of relaxation at a minimum. Please observe that

- the problem (15.5) is always feasible.
- a negative weight implies problem (15.5) 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 may imply that it is not possible repair the problem.

A simple choice of weights is to let them all to be 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 be unbounded. Hence, a repaired problem may not have an optimal solution.

Another and more important caveat is that only a minimial repair is perfomed i.e. the repair that just make the problem feasible. Hence, the repaired problem is barely feasible and that sometimes make the repaired problem hard to solve.

Feasibility Repair

MOSEK includes a function that repair an infeasible problem using the idea described in the previous section simply by passing a set of weights to **MOSEK**. This can be used for linear and conic optimization problems, possibly having integer constrained variables.

An example

Consider the example linear optimization

$$\begin{array}{lccccc} \text{minimize} & -10x_1 & & -9x_2, & & \\ \text{subject to} & 7/10x_1 & + & 1x_2 & \geq & 630, \\ & 1/2x_1 & + & 5/6x_2 & \geq & 600, \\ & 1x_1 & + & 2/3x_2 & \geq & 708, \\ & 1/10x_1 & + & 1/4x_2 & \geq & 135, \\ & x_1, & & x_2 & \geq & 0, \\ & & x_2 \geq 650 & & & \end{array} \tag{15.6}$$

which is infeasible. Now suppose we wish to use **MOSEK** to suggest a modification to the bounds that makes the problem feasible.

The function *MSK_primalrepair* can be used to repair an infeasible problem. Details about the function *MSK_primalrepair* can be seen in the reference.

Listing 15.3: An example of feasibility repair applied to problem (15.6).

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

#include "mosek.h"

static void MSKAPI printstr(void *handle,
                            const char str[])
{
  fputs(str, stdout);
} /* printstr */

int main(int argc, const char *argv[])
{
  const char  *filename = "../data/feasrepair.lp";
  MSKenv_t    env;
  MSKrescodee r;
  MSKtask_t   task;

  if ( argc > 1 )
    filename = argv[1];
```

```
  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_readdata(task, filename); /* Read file from current dir */

  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_optimize(task); /* Optimize the repaired task. */

      MSK_solutionsummary(task, MSK_STREAM_MSG);
    }
  }

  printf("Return code: %d\n", r);

  return ( r );
}
```

will produce the following

```
Copyright (c) MOSEK ApS, Denmark. WWW: mosek.com

Open file 'feasrepair.lp'

Read summary
Type             : LO (linear optimization problem)
Objective sense  : min
Constraints      : 4
Scalar variables : 2
Matrix variables : 0
Time             : 0.0

Computer
Platform               : Windows/64-X86
Cores                  : 4

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.
Interior-point optimizer started.
Presolve started.
Linear dependency checker started.
Linear dependency checker terminated.
Eliminator started.
Total number of eliminations : 2
Eliminator terminated.
Eliminator - tries                  : 1                 time                   : 0.00
Eliminator - elim's                 : 2
Lin. dep.  - tries                  : 1                 time                   : 0.00
Lin. dep.  - number                 : 0
Presolve terminated. Time: 0.00
Optimizer  - threads                : 1
Optimizer  - solved problem         : the primal
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                  : 5.40e+001
ITE PFEAS    DFEAS    GFEAS    PRSTATUS   POBJ              DOBJ              MU       TIME
0   2.7e+001 1.0e+000 4.8e+000 1.00e+000  4.195228609e+000  0.000000000e+000  1.0e+000 0.00
1   2.4e+001 8.6e-001 1.5e+000 0.00e+000  1.227497414e+001  1.504971820e+001  2.6e+000 0.00
2   2.6e+000 9.7e-002 1.7e-001 -6.19e-001 4.363064729e+001  4.648523094e+001  3.0e-001 0.00
3   4.7e-001 1.7e-002 3.1e-002 1.24e+000  4.256803136e+001  4.298540657e+001  5.2e-002 0.00
4   8.7e-004 3.2e-005 5.7e-005 1.08e+000  4.249989892e+001  4.250078747e+001  9.7e-005 0.00
5   8.7e-008 3.2e-009 5.7e-009 1.00e+000  4.249999999e+001  4.250000008e+001  9.7e-009 0.00
6   8.7e-012 3.2e-013 5.7e-013 1.00e+000  4.250000000e+001  4.250000000e+001  9.7e-013 0.00
Basis identification started.
Primal basis identification phase started.
ITER      TIME
0         0.00
Primal basis identification phase terminated. Time: 0.00
Dual basis identification phase started.
ITER      TIME
0         0.00
Dual basis identification phase terminated. Time: 0.00
Basis identification terminated. Time: 0.00
Interior-point optimizer terminated. Time: 0.00.

Optimizer terminated. Time: 0.03
Basic solution summary
Problem status  : PRIMAL_AND_DUAL_FEASIBLE
Solution status : OPTIMAL
Primal.  obj: 4.2500000000e+001   Viol.  con: 1e-013   var: 0e+000
Dual.    obj: 4.2500000000e+001   Viol.  con: 0e+000   var: 5e-013
Optimal objective value of the penalty problem: 4.250000000000e+001

Repairing bounds.
Increasing the upper bound -2.25e+001 on constraint 'c4' (3) with 1.35e+002.
Decreasing the lower bound 6.50e+002 on variable 'x2' (4) with 2.00e+001.
Primal feasibility repair terminated.
```

```
Optimizer started.
Interior-point optimizer started.
Presolve started.
Presolve terminated. Time: 0.00
Interior-point optimizer terminated. Time: 0.00.

Optimizer terminated. Time: 0.00

Interior-point solution summary
Problem status  : PRIMAL_AND_DUAL_FEASIBLE
Solution status : OPTIMAL
Primal.  obj: -5.6700000000e+003  Viol.  con: 0e+000   var: 0e+000
Dual.    obj: -5.6700000000e+003  Viol.  con: 0e+000   var: 0e+000

Basic solution summary
Problem status  : PRIMAL_AND_DUAL_FEASIBLE
Solution status : OPTIMAL
Primal.  obj: -5.6700000000e+003  Viol.  con: 0e+000   var: 0e+000
Dual.    obj: -5.6700000000e+003  Viol.  con: 0e+000   var: 0e+000

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
Clean primal-dual   - iterations : 0        time: 0.00
Simplex                 -                        time: 0.00
Primal simplex        - iterations : 0        time: 0.00
Dual simplex          - iterations : 0        time: 0.00
Primal-dual simplex   - iterations : 0        time: 0.00
Mixed integer           - relaxations: 0        time: 0.00
```

reports the optimal repair. In this case it is to increase the upper bound on constraint `c4` by *1.35e2* and decrease the lower bound on variable `x2` by *20*.

15.3 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 *[Chv83]* 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.

15.3.1 Sensitivity Analysis for Linear Problems

The Optimal Objective Value Function

Assume that we are given the problem

$$
\begin{array}{rclccccc}
z(l^c, u^c, l^x, u^x, c) & = & \text{minimize} & & & c^T x & & \\
& & \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\
& & & l^x & \leq & x & \leq & u^x,
\end{array}
\tag{15.7}
$$

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

$$
f_{l_i^c}(\beta) \tag{15.8}
$$

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 (15.8) 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 (15.8) is a piecewise linear and convex function, i.e. its graph may look like in Fig. 15.2 and Fig. 15.3.

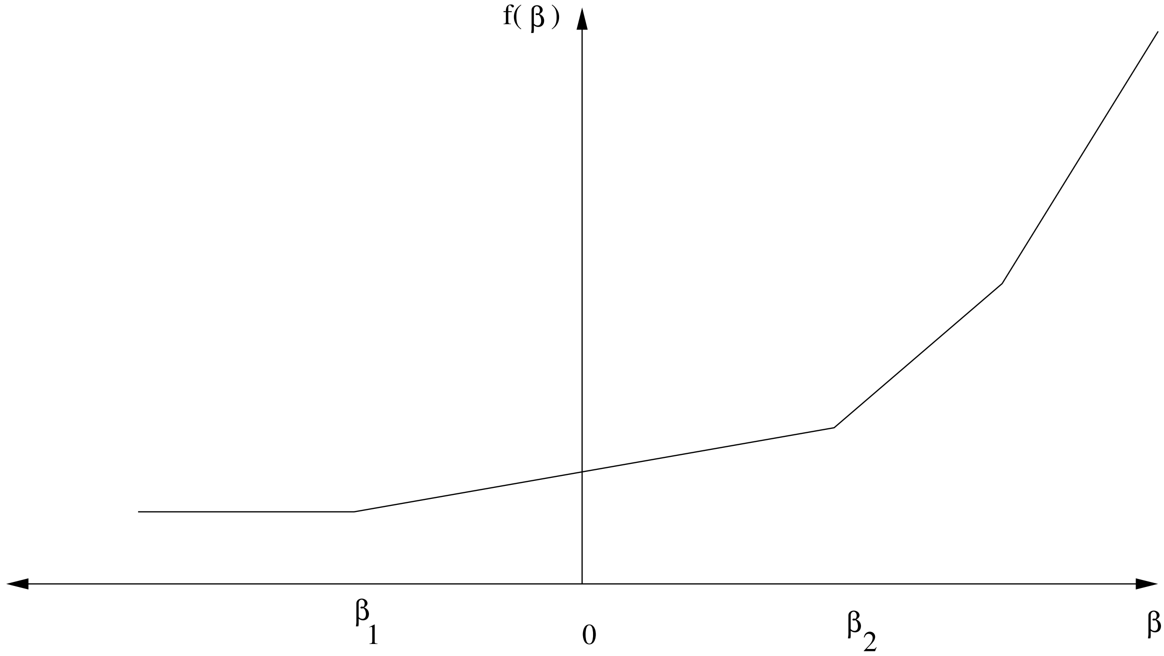

Fig. 15.2: $\beta = 0$ is in the interior of linearity interval.

Clearly, if the function $f_{l_i^c}(\beta)$ does not change much when β is changed, then we can conclude that the optimal objective value is insensitive to changes in l_i^c. Therefore, we are interested in the rate of change

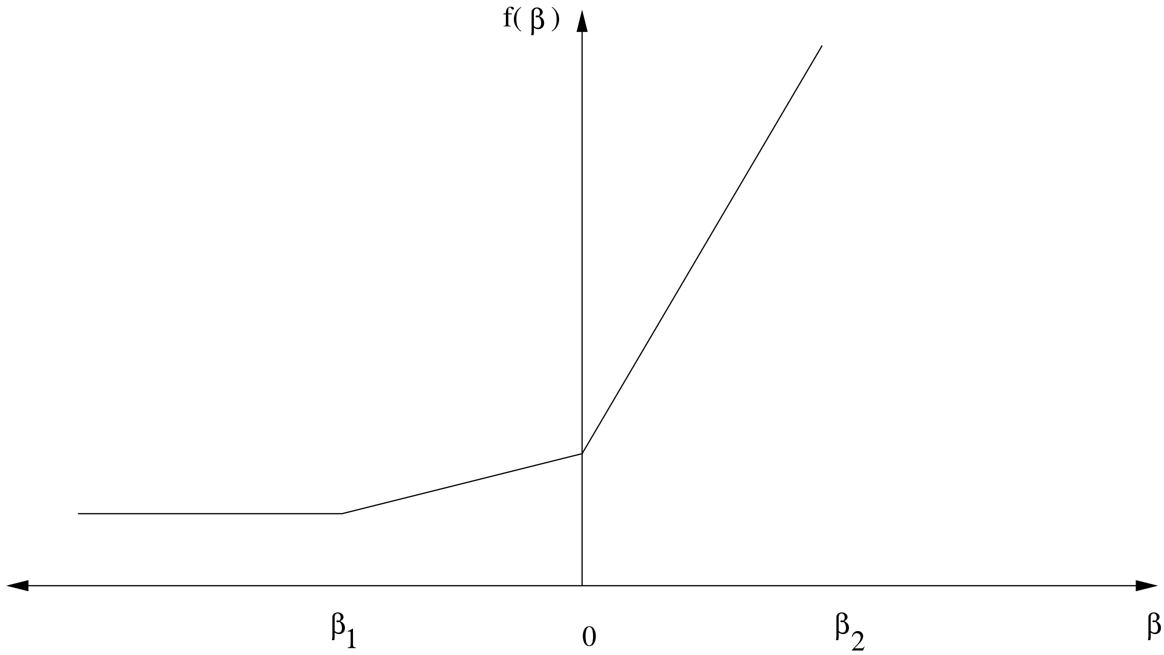

Fig. 15.3: $\beta = 0$ is a breakpoint.

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

$$f'_{l_i^c}(\beta) = f'_{l_i^c}(0).$$

Since $f_{l_i^c}$ is not a smooth function $f'_{l_i^c}$ may not be defined at 0, as illustrated in Fig. 15.3. 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 (15.7) 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. *[Chv83]*, is based on an optimal basic solution or, equivalently, on an optimal basis. This method may produce misleading results *[RTV97]* but is **computationally cheap**. Therefore, and for historical reasons, this method is available 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. This implies that the computed interval is only a subset of the largest interval for which the shadow price is constant. Furthermore, the optimal objective value function might have a breakpoint for $\beta = 0$. In this case 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.

The Optimal Partition Type Sensitivity Analysis

Another method for computing the complete linearity interval is called the *optimal partition type sensitivity analysis*. The main drawback of the optimal partition type sensitivity analysis is that it is computationally expensive compared to the basis type analysis. This type of sensitivity analysis is currently provided as an experimental feature in **MOSEK**.

Given the optimal primal and dual solutions to (15.7), i.e. x^* and $((s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*)$ the optimal objective value is given by

$$z^* := c^T x^*.$$

The left and right shadow prices σ_1 and σ_2 for l_i^c are given by this pair of optimization problems:

$$\begin{array}{lllll}
\sigma_1 & = & \text{minimize} & e_i^T s_l^c & \\
 & & \text{subject to} & A^T(s_l^c - s_u^c) + s_l^x - s_u^x & = & c, \\
 & & & (l^c)^T(s_l^c) - (u^c)^T(s_u^c) + (l^x)^T(s_l^x) - (u^x)^T(s_u^x) & = & z^*, \\
 & & & s_l^c, s_u^c, s_l^c, s_u^x \geq 0 & &
\end{array}$$

and

$$\begin{array}{lllll}
\sigma_2 & = & \text{maximize} & e_i^T s_l^c & \\
 & & \text{subject to} & A^T(s_l^c - s_u^c) + s_l^x - s_u^x & = & c, \\
 & & & (l^c)^T(s_l^c) - (u^c)^T(s_u^c) + (l^x)^T(s_l^x) - (u^x)^T(s_u^x) & = & z^*, \\
 & & & s_l^c, s_u^c, s_l^c, s_u^x \geq 0. & &
\end{array}$$

These two optimization problems make it easy to interpret the shadow price. Indeed, if $((s_l^c)^*, (s_u^c)^*, (s_l^x)^*, (s_u^x)^*)$ is an arbitrary optimal solution then

$$(s_l^c)_i^* \in [\sigma_1, \sigma_2].$$

Next, the linearity interval $[\beta_1, \beta_2]$ for l_i^c is computed by solving the two optimization problems

$$\begin{array}{lllllll}
\beta_1 & = & \text{minimize} & & & \beta & & \\
 & & \text{subject to} & l^c + \beta e_i & \leq & Ax & \leq & u^c, \\
 & & & & & c^T x - \sigma_1 \beta & = & z^*, \\
 & & & l^x & \leq & x & \leq & u^x,
\end{array}$$

and

$$\begin{array}{lllllll}
\beta_2 & = & \text{maximize} & & & \beta & & \\
 & & \text{subject to} & l^c + \beta e_i & \leq & Ax & \leq & u^c, \\
 & & & & & c^T x - \sigma_2 \beta & = & z^*, \\
 & & & l^x & \leq & x & \leq & u^x.
\end{array}$$

The linearity intervals and shadow prices for u_i^c, l_j^x, and u_j^x are computed similarly to l_i^c.

The left and right shadow prices for c_j denoted σ_1 and σ_2 respectively are computed as follows:

$$\begin{array}{rclcccc} \sigma_1 & = & \text{minimize} & & e_j^T x & & \\ & & \text{subject to} & l^c + \beta e_i & \leq & Ax & \leq & u^c, \\ & & & & & c^T x & = & z^*, \\ & & & l^x & \leq & x & \leq & u^x, \end{array}$$

and

$$\begin{array}{rclcccc} \sigma_2 & = & \text{maximize} & & e_j^T x & & \\ & & \text{subject to} & l^c + \beta e_i & \leq & Ax & \leq & u^c, \\ & & & & & c^T x & = & z^*, \\ & & & l^x & \leq & x & \leq & u^x. \end{array}$$

Once again the above two optimization problems make it easy to interpret the shadow prices. Indeed, if x^* is an arbitrary primal optimal solution, then

$$x_j^* \in [\sigma_1, \sigma_2].$$

The linearity interval $[\beta_1, \beta_2]$ for a c_j is computed as follows:

$$\begin{array}{rclccc} \beta_1 & = & \text{minimize} & \beta & & \\ & & \text{subject to} & A^T(s_l^c - s_u^c) + s_l^x - s_u^x & = & c + \beta e_j, \\ & & & (l^c)^T(s_l^c) - (u^c)^T(s_u^c) + (l^x)^T(s_l^x) - (u^x)^T(s_u^x) - \sigma_1 \beta & \leq & z^*, \\ & & & s_l^c, s_u^c, s_l^c, s_u^x \geq 0 & & \end{array}$$

and

$$\begin{array}{rclccc} \beta_2 & = & \text{maximize} & \beta & & \\ & & \text{subject to} & A^T(s_l^c - s_u^c) + s_l^x - s_u^x & = & c + \beta e_j, \\ & & & (l^c)^T(s_l^c) - (u^c)^T(s_u^c) + (l^x)^T(s_l^x) - (u^x)^T(s_u^x) - \sigma_2 \beta & \leq & z^*, \\ & & & s_l^c, s_u^c, s_l^c, s_u^x \geq 0. & & \end{array}$$

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. 15.4.

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

$$\begin{array}{ccccccccccccccc} x_{11} & + & x_{12} & & & & & & & & & & & \leq & 400, \\ & & & & x_{23} & + & x_{24} & & & & & & & \leq & 1200, \\ & & & & & & & & x_{31} & + & x_{33} & + & x_{34} & \leq & 1000, \\ x_{11} & & & & & & & + & x_{31} & & & & & = & 800, \\ & & x_{12} & & & & & & & & & & & = & 100, \\ & & & & x_{23} & + & & & & & x_{33} & & & = & 500, \\ & & & & & & x_{24} & + & & & & & x_{34} & = & 500, \\ x_{11}, & & x_{12}, & & x_{23}, & & x_{24}, & & x_{31}, & & x_{33}, & & x_{34} & \geq & 0. \end{array} \tag{15.9}$$

The sensitivity parameters are shown in Table 15.1 and Table 15.2 for the basis type analysis and in Table 15.3 and Table 15.4 for the optimal partition type analysis.

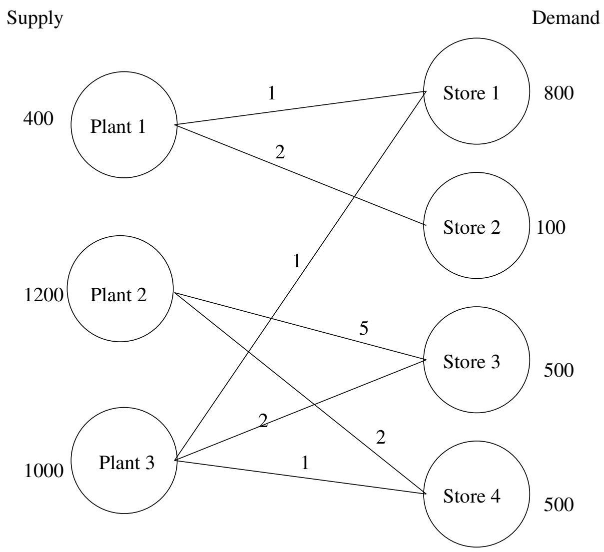

Fig. 15.4: Supply, demand and cost of transportation.

Table 15.1: Ranges and shadow prices related to bounds on constraints and variables: results for the basis type sensitivity analysis.

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 15.2: Ranges and shadow prices related to bounds on constraints and variables: results for the optimal partition type sensitivity analysis.

Con.	β_1	β_2	σ_1	σ_2
1	-300.00	500.00	3.00	1.00
2	-700.00	$+\infty$	-0.00	-0.00
3	-500.00	500.00	3.00	1.00
4	-500.00	500.00	2.00	4.00
5	-100.00	300.00	3.00	5.00
6	-500.00	700.00	3.00	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}	$-\infty$	500.00	0.00	2.00
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}	$-\infty$	500.00	0.00	2.00

Table 15.3: Ranges and shadow prices related to the objective coefficients: results for the basis type sensitivity analysis.

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

Table 15.4: Ranges and shadow prices related to the objective coefficients: results for the optimal partition type sensitivity analysis.

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 optimal partition type sensitivity analysis we see that for constraint number 1 we have $\sigma_1 = 3$, $\sigma_2 = 1$ and $\beta_1 = -300$, $\beta_2 = 500$. Therefore, we have a left linearity interval of $[-300, 0]$ and a right interval of $[0, 500]$. The corresponding left and right shadow prices are 3 and 1 respectively. This implies that if the upper bound on constraint 1 increases by

$$\beta \in [0, \beta_1] = [0, 500]$$

then the optimal objective value will decrease by the value

$$\sigma_2 \beta = 1\beta.$$

Correspondingly, if the upper bound on constraint 1 is decreased by

$$\beta \in [0, 300]$$

then the optimal objective value will increase by the value

$$\sigma_1 \beta = 3\beta.$$

15.3.2 Sensitivity Analysis with MOSEK

MOSEK provides the functions *MSK_primalsensitivity* and *MSK_dualsensitivity* for performing sensitivity analysis. The code in Listing 15.4 gives an example of its use.

Listing 15.4: 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};
  MSKidxt         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;

  /* 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_optimize(task);

    if ( r == MSK_RES_OK )
    {
      /* Analyze upper bound on c1 and the equality constraint on c4 */
      MSKidxt subi[]  = {0, 3};
      MSKmarke marki[] = {MSK_MARK_UP, MSK_MARK_UP};

      /* Analyze lower bound on the variables x12 and x31 */
```

```
    MSKidxt 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,
                            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

SIXTEEN

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*
- *Nonlinear API (SCopt, DGopt, EXPopt)*

16.1 API Conventions

16.1.1 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 16.1 the variable names used to specify the problem parameters are listed.

Table 16.1: 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
numqonz	int		q^o_{ij}
qosubi	int[]	numqonz	q^o_{ij}
qosubj	int[]	numqonz	q^o_{ij}
qoval	double*	numqonz	q^o_{ij}
c	double[]	numvar	c_j
cfix	double		c^f
numqcnz	int		q^k_{ij}
qcsubk	int[]	qcnz	q^k_{ij}
qcsubi	int[]	qcnz	q^k_{ij}
qcsubj	int[]	qcnz	q^k_{ij}
qcval	double*	qcnz	q^k_{ij}
aptrb	int[]	numvar	a_{ij}
aptre	int[]	numvar	a_{ij}
asub	int[]	aptre[numvar-1]	a_{ij}
aval	double[]	aptre[numvar-1]	a_{ij}
bkc	MSKboundkeye*	numcon	l^c_k and u^c_k
blc	double[]	numcon	l^c_k
buc	double[]	numcon	u^c_k
bkx	MSKboundkeye*	numvar	l^x_k and u^x_k
blx	double[]	numvar	l^x_k
bux	double[]	numvar	u^x_k

The relation between the variable names and the problem parameters is as follows:

- The quadratic terms in the objective: $q^o_{\mathtt{qosubi[t]},\mathtt{qosubj[t]}} = \mathtt{qoval[t]}, \quad t = 0, \ldots, \mathtt{numqonz} - 1.$
- The linear terms in the objective : $c_j = \mathtt{c[j]}, \quad j = 0, \ldots, \mathtt{numvar} - 1$
- The fixed term in the objective : $c^f = \mathtt{cfix}$.
- The quadratic terms in the constraints: $q^{\mathtt{qcsubk[t]}}_{\mathtt{qcsubi[t]},\mathtt{qcsubj[t]}} = \mathtt{qcval[t]}, \quad t = 0, \ldots, \mathtt{numqcnz} - 1$
- The linear terms in the constraints: $a_{\mathtt{asub[t]},\mathtt{j}} = \mathtt{aval[t]}, \quad t = \mathtt{ptrb[j]}, \ldots, \mathtt{ptre[j]} - 1, \; j = 0, \ldots, \mathtt{numvar} - 1$

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.

16.1.2 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 16.2

Table 16.2: 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

$$l_k^c = \mathtt{blc}[\mathtt{k}], \quad k = 0, \ldots, \mathtt{numcon} - 1$$

$$u_k^c = \mathtt{buc}[\mathtt{k}], \quad k = 0, \ldots, \mathtt{numcon} - 1.$$

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

$$l_j^x = \mathtt{blx}[\mathtt{j}], \quad j = 0, \ldots, \mathtt{numvar} - 1.$$

$$u_j^x = \mathtt{bux}[\mathtt{j}], \quad j = 0, \ldots, \mathtt{numvar} - 1.$$

16.1.3 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_getboundslice(task,MSK_ACC_CON, 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);
```

16.1.4 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:

$$a_{\mathtt{asub}[k],j} = \mathtt{aval}[k] \quad \text{for} \quad j = 0, \ldots, \mathtt{numcol} - 1,\ k = \mathtt{ptrb}[j], \ldots, \mathtt{ptre}[j] - 1.$$

As an example consider the matrix

$$A = \begin{bmatrix} 1.1 & & 1.3 & 1.4 & \\ & 2.2 & & & 2.5 \\ 3.1 & & & 3.4 & \\ & & 4.4 & & \end{bmatrix} \tag{16.1}$$

which can be represented in the column ordered sparse matrix format as

$$\begin{aligned} \mathtt{ptrb} &= [0, 2, 3, 5, 7], \\ \mathtt{ptre} &= [2, 3, 5, 7, 8], \\ \mathtt{asub} &= [0, 2, 1, 0, 3, 0, 2, 1], \\ \mathtt{aval} &= [1.1, 3.1, 2.2, 1.3, 4.4, 1.4, 3.4, 2.5]. \end{aligned}$$

Fig. 16.1 illustrates how the matrix A in (16.1) is represented in column ordered sparse matrix format.

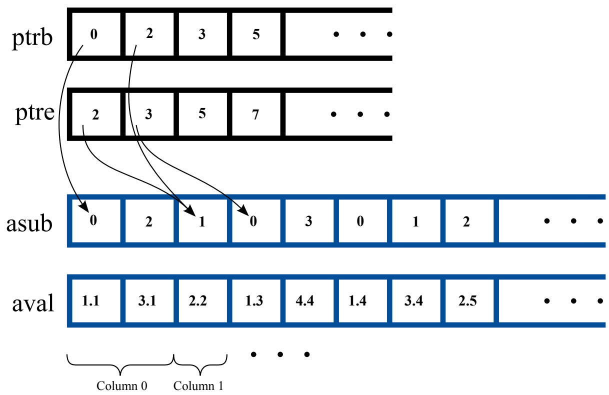

Fig. 16.1: The matrix A (16.1) represented in column ordered packed sparse matrix format.

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 (16.1) would now be represented as:

$$\begin{aligned} \mathtt{ptrb} &= [0, 2, 3, 5, 7], \\ \mathtt{nzc} &= [2, 1, 2, 2, 1], \\ \mathtt{asub} &= [0, 2, 1, 0, 3, 0, 2, 1], \\ \mathtt{aval} &= [1.1, 3.1, 2.2, 1.3, 4.4, 1.4, 3.4, 2.5]. \end{aligned}$$

Row ordered sparse matrix

The matrix A (16.1) can also be represented in the row ordered sparse matrix format as:

$$\begin{array}{lcl} \mathtt{ptrb} & = & [0,3,5,7], \\ \mathtt{ptre} & = & [3,5,7,8], \\ \mathtt{asub} & = & [0,2,3,1,4,0,3,2], \\ \mathtt{aval} & = & [1.1,1.3,1.4,2.2,2.5,3.1,3.4,4.4]. \end{array}$$

16.2 Functions grouped by topic

Basis matrix

- *Infrequent:* *MSK_basiscond*, *MSK_initbasissolve*, *MSK_solvewithbasis*

Bound data

- *MSK_putconbound* – Changes the bound for one constraint.
- *MSK_putconboundlist* – Changes the bounds of a list of constraints.
- *MSK_putconboundslice* – Changes the bounds for a slice of the constraints.
- *MSK_putvarbound* – Changes the bound for one variable.
- *MSK_putvarboundlist* – Changes the bounds of a list of variables.
- *Infrequent:* *MSK_chgconbound*, *MSK_chgvarbound*, *MSK_getconbound*, *MSK_getconboundslice*, *MSK_getvarbound*, *MSK_getvarboundslice*
- *Deprecated:* ~~*MSK_chgbound*~~, ~~*MSK_getbound*~~, ~~*MSK_getboundslice*~~, ~~*MSK_putbound*~~, ~~*MSK_putboundlist*~~, ~~*MSK_putboundslice*~~

Callback

- *MSK_linkfunctotaskstream* – Connects a user-defined function to a task stream.
- *MSK_putcallbackfunc* – Input the progress callback function.
- *MSK_putnlfunc* – Inputs nonlinear function information.
- *MSK_unlinkfuncfromtaskstream* – Disconnects a user-defined function from a task stream.
- *Infrequent:* *MSK_getcallbackfunc*, *MSK_getnlfunc*, *MSK_linkfunctoenvstream*, *MSK_putexitfunc*, *MSK_putresponsefunc*, *MSK_unlinkfuncfromenvstream*

Conic constraint data

- *MSK_appendcone* – Appends a new conic constraint to the problem.
- *MSK_putcone* – Replaces a conic constraint.
- *MSK_removecones* – Removes a number of conic constraints from the problem.
- *Infrequent:* *MSK_appendconeseq*, *MSK_appendconesseq*, *MSK_getcone*, *MSK_getconeinfo*, *MSK_getnumcone*, *MSK_getnumconemem*

Data file

- *MSK_readdata* – Reads problem data from a file.
- *MSK_readsolution* – Reads a solution from a file.
- *MSK_writedata* – Writes problem data to a file.
- *MSK_writesolution* – Write a solution to a file.
- *Infrequent:* *MSK_readdataautoformat*, *MSK_readdataformat*, *MSK_readparamfile*, *MSK_writejsonsol*, *MSK_writeparamfile*

Environment management

- *MSK_deleteenv* – Delete a MOSEK environment.
- *MSK_licensecleanup* – Stops all threads and delete all handles used by the license system.
- *MSK_makeenv* – Creates a new MOSEK environment.
- *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_checkoutlicense*, *MSK_makeenvalloc*, *MSK_putlicensecode*

Infeasibility diagnostics

- *MSK_getinfeasiblesubproblem* – Obtains an infeasible subproblem.
- *MSK_primalrepair* – Repairs a primal infeasible optimization problem by adjusting the bounds on the constraints and variables.

Linear algebra

- *MSK_axpy* – Computes vector addition and multiplication by a scalar.
- *MSK_computesparsecholesky* – Computes a Cholesky factorization of sparse matrix.
- *MSK_dot* – Computes the inner product of two vectors.
- *MSK_gemm* – Performs a dense matrix multiplication.
- *MSK_gemv* – Computes dense matrix times a dense vector product.
- *MSK_potrf* – Computes a Cholesky factorization of a dense matrix.
- *MSK_sparsetriangularsolvedense* – Solves a sparse triangular system of linear equations.
- *MSK_syeig* – Computes all eigenvalues of a symmetric dense matrix.
- *MSK_syevd* – Computes all the eigenvalues and eigenvectors of a symmetric dense matrix, and thus its eigenvalue decomposition.
- *MSK_syrk* – Performs a rank-k update of a symmetric matrix.

Linear constraint data

- *MSK_appendcons* – Appends a number of constraints to the optimization task.
- *MSK_getnumcon* – Obtains the number of constraints.
- *MSK_putconboundslice* – Changes the bounds for a slice of the constraints.
- *MSK_removecons* – Removes a number of constraints.
- *Infrequent:* *MSK_getmaxnumcon*

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_unlinkfuncfromtaskstream* – Disconnects a user-defined function from a task stream.
- *Infrequent:* *MSK_echoenv*, *MSK_echotask*, *MSK_linkfiletoenvstream*, *MSK_linkfunctoenvstream*, *MSK_unlinkfuncfromenvstream*

Memory

- *Infrequent:* *MSK_callocdbgenv*, *MSK_callocdbgtask*, *MSK_callocenv*, *MSK_calloctask*, *MSK_checkmemenv*, *MSK_checkmemtask*, *MSK_freedbgenv*, *MSK_freedbgtask*, *MSK_freeenv*, *MSK_freetask*, *MSK_getmemusagetask*

Naming

- *MSK_putbarvarname* – Sets the name of a semidefinite variable.
- *MSK_putconename* – Sets the name of a cone.
- *MSK_putconname* – Sets the name of a constraint.
- *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_getbarvarname*, *MSK_getbarvarnameindex*, *MSK_getbarvarnamelen*, *MSK_getconename*, *MSK_getconenameindex*, *MSK_getconenamelen*, *MSK_getconname*, *MSK_getconnameindex*, *MSK_getconnamelen*, *MSK_getmaxnamelen*, *MSK_getobjname*, *MSK_getobjnamelen*, *MSK_gettaskname*, *MSK_gettasknamelen*, *MSK_getvarname*, *MSK_getvarnameindex*, *MSK_getvarnamelen*

Objective data

- *MSK_putcfix* – Replaces the fixed term in the objective.
- *MSK_putobjsense* – Sets the objective sense.
- *Infrequent:* *MSK_getobjsense*

Optimization

- *MSK_optimize* – Optimizes the problem.
- *MSK_optimizetrm* – Optimizes the problem.

Optimizer statistics

- *MSK_getdouinf* – Obtains a double information item.
- *MSK_getintinf* – Obtains an integer information item.
- *MSK_getlintinf* – Obtains a long integer information item.
- *Infrequent:* *MSK_getinfindex*, *MSK_getinfmax*, *MSK_getinfname*, *MSK_getnadouinf*, *MSK_getnaintinf*, *MSK_getnaintparam*

Parameter management

- *Infrequent:* *MSK_getnumparam*, *MSK_getparammax*, *MSK_getparamname*, *MSK_getsymbcondim*, *MSK_iparvaltosymnam*, *MSK_isdouparname*, *MSK_isintparname*, *MSK_isstrparname*, *MSK_setdefaults*, *MSK_symnamtovalue*, *MSK_whichparam*

Parameters (get)

- *Infrequent:* *MSK_getdouparam*, *MSK_getintparam*, *MSK_getnadouparam*, *MSK_getnastrparam*, *MSK_getnastrparamal*, *MSK_getstrparam*, *MSK_getstrparamal*, *MSK_getstrparamlen*

Parameters (put)

- *MSK_putdouparam* – Sets a double parameter.
- *MSK_putintparam* – Sets an integer parameter.
- *MSK_putstrparam* – Sets a string parameter.
- *Infrequent:* *MSK_putnadouparam*, *MSK_putnaintparam*, *MSK_putnastrparam*, *MSK_putparam*

Scalar variable data

- *MSK_appendvars* – Appends a number of variables to the optimization task.
- *MSK_getnumvar* – Obtains the number of variables.
- *MSK_putacol* – Replaces all elements in one column of the linear constraint matrix.
- *MSK_putaij* – Changes a single value in the linear coefficient matrix.
- *MSK_putarow* – Replaces all elements in one row of the linear constraint matrix.
- *MSK_putcj* – Modifies one linear coefficient in the objective.
- *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.
- *MSK_putvarboundslice* – Changes the bounds for a slice of the variables.

- *MSK_putvartype* – Sets the variable type of one variable.
- *MSK_removevars* – Removes a number of variables.
- *Infrequent:* *MSK_commitchanges*, *MSK_getacol*, *MSK_getacolnumnz*, *MSK_getacolslicetrip*, *MSK_getaij*, *MSK_getarow*, *MSK_getarownumnz*, *MSK_getarowslicetrip*, *MSK_getc*, *MSK_getcfix*, *MSK_getcj*, *MSK_getcslice*, *MSK_getlenbarvarj*, *MSK_getmaxnumanz*, *MSK_getmaxnumanz64*, *MSK_getmaxnumqnz*, *MSK_getmaxnumqnz64*, *MSK_getmaxnumvar*, *MSK_getnumanz*, *MSK_getnumanz64*, *MSK_getnumintvar*, *MSK_getnumqconknz*, *MSK_getnumqconknz64*, *MSK_getnumqobjnz*, *MSK_getnumqobjnz64*, *MSK_getnumsymmat*, *MSK_getqconk*, *MSK_getqconk64*, *MSK_getqobj*, *MSK_getqobj64*, *MSK_getqobjij*, *MSK_getsparsesymmat*, *MSK_getsymmatinfo*, *MSK_getvartype*, *MSK_getvartypelist*, *MSK_putacollist*, *MSK_putacollist64*, *MSK_putacolslice*, *MSK_putacolslice64*, *MSK_putaijlist*, *MSK_putaijlist64*, *MSK_putarowlist*, *MSK_putarowlist64*, *MSK_putarowslice*, *MSK_putarowslice64*, *MSK_putclist*, *MSK_putcslice*, *MSK_putmaxnumanz*, *MSK_putmaxnumqnz*, *MSK_putmaxnumvar*, *MSK_putvartypelist*
- *Deprecated:* ~~*MSK_getaslice*~~, ~~*MSK_getaslice64*~~

Sensitivity analysis

- *MSK_dualsensitivity* – Performs sensitivity analysis on objective coefficients.
- *MSK_primalsensitivity* – Perform sensitivity analysis on bounds.
- *MSK_sensitivityreport* – Creates a sensitivity report.

Solution (get)

- *MSK_getbarsj* – Obtains the dual solution for a semidefinite variable.
- *MSK_getbarxj* – Obtains the primal solution for a semidefinite variable.
- *MSK_getskcslice* – Obtains the status keys for a slice of the constraints.
- *MSK_getskxslice* – Obtains the status keys for a slice of the scalar variables.
- *MSK_getslcslice* – Obtains a slice of the slc vector for a solution.
- *MSK_getslxslice* – Obtains a slice of the slx vector for a solution.
- *MSK_getsnxslice* – Obtains a slice of the snx vector for a solution.
- *MSK_getsucslice* – Obtains a slice of the suc vector for a solution.
- *MSK_getsuxslice* – Obtains a slice of the sux vector for a solution.
- *MSK_getxcslice* – Obtains a slice of the xc vector for a solution.
- *MSK_getxxslice* – Obtains a slice of the xx vector for a solution.
- *MSK_getyslice* – Obtains a slice of the y vector for a solution.
- *Infrequent:* *MSK_getreducedcosts*, *MSK_getskc*, *MSK_getskx*, *MSK_getslc*, *MSK_getslx*, *MSK_getsnx*, *MSK_getsolution*, *MSK_getsolutionslice*, *MSK_getsuc*, *MSK_getsux*, *MSK_getxc*, *MSK_getxx*, *MSK_gety*
- *Deprecated:* ~~*MSK_getsolutioni*~~

Solution (put)

- *MSK_putbarsj* – Sets the dual solution for a semidefinite variable.
- *MSK_putbarxj* – Sets the primal solution for a semidefinite variable.

- *MSK_putskcslice* – Sets the status keys for a slice of the constraints.
- *MSK_putskxslice* – Sets the status keys for a slice of the variables.
- *MSK_putslcslice* – Sets a slice of the slc vector for a solution.
- *MSK_putslxslice* – Sets a slice of the slx vector for a solution.
- *MSK_putsnxslice* – Sets a slice of the snx vector for a solution.
- *MSK_putsolution* – Inserts a solution.
- *MSK_putsucslice* – Sets a slice of the suc vector for a solution.
- *MSK_putsuxslice* – Sets a slice of the sux vector for a solution.
- *MSK_putxcslice* – Sets a slice of the xc vector for a solution.
- *MSK_putxxslice* – Obtains a slice of the xx vector for a solution.
- *MSK_putyslice* – Sets a slice of the y vector for a solution.
- *Infrequent:* *MSK_putskc*, *MSK_putskx*, *MSK_putslc*, *MSK_putslx*, *MSK_putsnx*, *MSK_putsuc*, *MSK_putsux*, *MSK_putxc*, *MSK_putxx*, *MSK_puty*, *MSK_solstatostr*
- *Deprecated:* ~~*MSK_putsolutioni*~~

Solution information

- *MSK_getdualobj* – Computes the dual objective value associated with the solution.
- *MSK_getdualsolutionnorms* – Compute norms of the dual solution.
- *MSK_getdviolbarvar* – Computes the violation of dual solution for a set of semidefinite variables.
- *MSK_getdviolcon* – Computes the violation of a dual solution associated with a set of constraints.
- *MSK_getdviolcones* – Computes the violation of a solution for set of dual conic constraints.
- *MSK_getdviolvar* – Computes the violation of a dual solution associated with a set of scalar variables.
- *MSK_getprimalobj* – Computes the primal objective value for the desired solution.
- *MSK_getprimalsolutionnorms* – Compute norms of the primal solution.
- *MSK_getprosta* – Obtains the problem status.
- *MSK_getpviolbarvar* – Computes the violation of a primal solution for a list of semidefinite variables.
- *MSK_getpviolcon* – Computes the violation of a primal solution associated to a constraint.
- *MSK_getpviolcones* – Computes the violation of a solution for set of conic constraints.
- *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_solutiondef* – Checks whether a solution is defined.

Symmetric matrix variable data

- *MSK_appendbarvars* – Appends semidefinite variables to the problem.
- *MSK_appendsparsesymmat* – Appends a general sparse symmetric matrix to the storage of symmetric matrices.
- *MSK_putbaraij* – Inputs an element of barA.

- *MSK_putbarcj* – Changes one element in barc.
- *Infrequent:* *MSK_getbarablocktriplet*, *MSK_getbaraidx*, *MSK_getbaraidxij*, *MSK_getbaraidxinfo*, *MSK_getbarasparsity*, *MSK_getbarcblocktriplet*, *MSK_getbarcidx*, *MSK_getbarcidxinfo*, *MSK_getbarcidxj*, *MSK_getbarcsparsity*, *MSK_getdimbarvarj*, *MSK_getmaxnumbarvar*, *MSK_getnumbarablocktriplets*, *MSK_getnumbaranz*, *MSK_getnumbarcblocktriplets*, *MSK_getnumbarcnz*, *MSK_getnumbarvar*, *MSK_putbarablocktriplet*, *MSK_putbarcblocktriplet*, *MSK_putmaxnumbarvar*, *MSK_removebarvars*

Task diagnostics

- *MSK_checkconvexity* – Checks if a quadratic optimization problem is convex.
- *MSK_getprobtype* – Obtains the problem type.
- *MSK_onesolutionsummary* – Prints a short summary of a specified solution.
- *MSK_optimizersummary* – Prints a short summary with optimizer statistics from last optimization.
- *MSK_printdata* – Prints a part of the problem data to a stream.
- *MSK_printparam* – Prints the current parameter settings.
- *MSK_solutionsummary* – Prints a short summary of the current solutions.
- *MSK_updatesolutioninfo* – Update the information items related to the solution.
- *Infrequent:* *MSK_analyzenames*, *MSK_analyzeproblem*, *MSK_analyzesolution*, *MSK_echointro*, *MSK_readsummary*

Task management

- *MSK_deletetask* – Deletes a task.
- *MSK_maketask* – Creates a new task.
- *Infrequent:* *MSK_clonetask*, *MSK_deletesolution*, *MSK_getcodedesc*, *MSK_getmaxnumcone*, *MSK_getresponseclass*, *MSK_inputdata*, *MSK_inputdata64*, *MSK_makeemptytask*, *MSK_putmaxnumcon*, *MSK_putmaxnumcone*

Other

- *MSK_asyncgetresult* – Request a response from a remote job.
- *MSK_asyncoptimize* – Offload the optimization task to a solver server.
- *MSK_asyncpoll* – Requests information about the status of the remote job.
- *MSK_asyncstop* – Request that the job identified by the token is terminated.
- *MSK_checkversion* – Compares a version of the MOSEK DLL with a specified version.
- *MSK_getbuildinfo* – Obtains build information.
- *MSK_getversion* – Obtains MOSEK version information.
- *MSK_optimizermt* – Offload the optimization task to a solver server.
- *MSK_putsolutionyi* – Inputs the dual variable of a solution.
- *MSK_readtask* – Load task data from a file.
- *MSK_resizetask* – Resizes an optimization task.
- *MSK_toconic* – In-place reformulation of a QCQP to a COP

- *MSK_writetask* – Write a complete binary dump of the task data.
- *Infrequent:* *MSK_bktostr*, *MSK_callbackcodetostr*, *MSK_conetypetostr*, *MSK_getapiecenumnz*, *MSK_getenv*, *MSK_getlasterror*, *MSK_getlasterror64*, *MSK_getsymbcon*, *MSK_isinfinity*, *MSK_probtypetostr*, *MSK_prostatostr*, *MSK_sktostr*, *MSK_strdupdbgtask*, *MSK_strduptask*, *MSK_strtoconetype*, *MSK_strtosk*, *MSK_utf8towchar*, *MSK_wchartoutf8*, *MSK_writetasksolverresult_file*
- *Deprecated:* ~~*MSK_getaslicenumnz*~~, ~~*MSK_getaslicenumnz64*~~

16.3 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 *Task diagnostics*

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 *Task diagnostics*

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 *Task diagnostics*

`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 *Symmetric matrix variable data*

`MSK_appendcone`

```
MSKrescodee (MSKAPI MSK_appendcone) (
  MSKtask_t task,
  MSKconetypee ct,
  MSKrealt conepar,
  MSKint32t nummem,
  const MSKint32t * submem)
```

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`.

Depending on the value of `ct` this function appends a normal (*MSK_CT_QUAD*) or rotated quadratic cone (*MSK_CT_RQUAD*).

Define

$$\hat{x} = x_{\texttt{submem}[0]}, \ldots, x_{\texttt{submem}[\texttt{nummem}-1]}.$$

Depending on the value of `ct` this function appends one of the constraints:

- Quadratic cone (*MSK_CT_QUAD*) :

$$\hat{x}_0 \geq \sqrt{\sum_{i=1}^{i<\texttt{nummem}} \hat{x}_i^2}$$

- Rotated quadratic cone (*MSK_CT_RQUAD*) :

$$2\hat{x}_0\hat{x}_1 \geq \sum_{i=2}^{i<\texttt{nummem}} \hat{x}_i^2, \quad \hat{x}_0, \hat{x}_1 \geq 0$$

Please note that the sets of variables appearing in different conic constraints must be disjoint.

For an explained code example see Section *Conic Quadratic Optimization*.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `ct` (*MSKconetypee*) – Specifies the type of the cone. (input)
- `conepar` (*MSKrealt*) – This argument is currently not used. It 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 *Conic constraint data*

`MSK_appendconeseq`

```
MSKrescodee (MSKAPI MSK_appendconeseq) (
  MSKtask_t task,
  MSKconetypee ct,
  MSKrealt conepar,
  MSKint32t nummem,
  MSKint32t j)
```

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*) – This argument is currently not used. It 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 *Conic constraint data*

MSK_appendconesseq

```
MSKrescodee (MSKAPI MSK_appendconesseq) (
  MSKtask_t task,
  MSKint32t num,
  const MSKconetypee * ct,
  const MSKrealt * conepar,
  const MSKint32t * nummem,
  MSKint32t j)
```

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* *) – This argument is currently not used. It 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 *Conic constraint data*

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 *Linear constraint data*

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 *Symmetric matrix variable data*

`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 *Scalar variable data*

`MSK_asyncgetresult`

```
MSKrescodee (MSKAPI MSK_asyncgetresult) (
  MSKtask_t task,
  const char * server,
  const char * port,
  const char * token,
  MSKbooleant * respavailable,
  MSKrescodee * resp,
  MSKrescodee * trm)
```

Request a response from a remote job. If successful, solver response, termination code and solutions are retrieved.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- server (*MSKstring_t*) – Name or IP address of the solver server. (input)
- port (*MSKstring_t*) – Network port of the solver service. (input)
- token (*MSKstring_t*) – 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.

MSK_asyncoptimize

```
MSKrescodee (MSKAPI MSK_asyncoptimize) (
  MSKtask_t task,
  const char * server,
  const char * port,
  char * token)
```

Offload the optimization task to a solver server defined by server:port. The call will return immediately and not wait for the result.

If the string parameter *MSK_SPAR_REMOTE_ACCESS_TOKEN* is not blank, it will be passed to the server as authentication.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- server (*MSKstring_t*) – Name or IP address of the solver server (input)
- port (*MSKstring_t*) – Network port of the solver service (input)
- token (*MSKstring_t*) – Returns the task token (output)

Return (*MSKrescodee*) – The function response code.

MSK_asyncpoll

```
MSKrescodee (MSKAPI MSK_asyncpoll) (
  MSKtask_t task,
  const char * server,
  const char * port,
  const char * token,
  MSKbooleant * respavailable,
  MSKrescodee * resp,
  MSKrescodee * trm)
```

Requests information about the status of the remote job.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- server (*MSKstring_t*) – Name or IP address of the solver server (input)
- port (*MSKstring_t*) – Network port of the solver service (input)
- token (*MSKstring_t*) – 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.

MSK_asyncstop

```
MSKrescodee (MSKAPI MSK_asyncstop) (
  MSKtask_t task,
  const char * server,
  const char * port,
  const char * token)
```

Request that the job identified by the token is terminated.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- server (*MSKstring_t*) – Name or IP address of the solver server (input)
- port (*MSKstring_t*) – Network port of the solver service (input)
- token (*MSKstring_t*) – The task token (input)

Return (*MSKrescodee*) – The function response code.

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.

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 *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` (*MSKstring_t*) – String corresponding to the bound key code `bk`. (output)

Return (*MSKrescodee*) – The function response code.

`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` (*MSKstring_t*) – String corresponding to the callback code. (output)

Return (*MSKrescodee*) – The function response code.

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 (*MSKstring_t*) – 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 *Memory*

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 (*MSKstring_t*) – 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 *Memory*

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 *Memory*

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 *Memory*

MSK_checkconvexity

```
MSKrescodee (MSKAPI MSK_checkconvexity) (
  MSKtask_t task)
```

This function checks if a quadratic optimization problem is convex. The amount of checking is controlled by *MSK_IPAR_CHECK_CONVEXITY*.

The function reports an error if the problem is not convex.

Parameters task (*MSKtask_t*) – An optimization task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Task diagnostics*

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 *Environment management*

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 *Environment management*

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 (*MSKstring_t*) – 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 *Memory*

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 (*MSKstring_t*) – 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 *Memory*

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 *Environment management*

`MSK_checkversion`

```
MSKrescodee (MSKAPI MSK_checkversion) (
  MSKenv_t env,
  MSKint32t major,
  MSKint32t minor,
  MSKint32t build,
  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)
- `build` (*MSKint32t*) – Build number. (input)
- `revision` (*MSKint32t*) – Revision number. (input)

Return (*MSKrescodee*) – The function response code.

~~`MSK_chgbound`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_chgbound) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t i,
  MSKint32t lower,
  MSKint32t finite,
  MSKrealt value)
```

Changes a bound for one constraint or variable. If `accmode` equals *MSK_ACC_CON*, a constraint bound is changed, otherwise a variable bound is changed.

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 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)
- `accmode` (*MSKaccmodee*) – Defines if operations are performed row-wise (constraint-oriented) or column-wise (variable-oriented). (input)
- `i` (*MSKint32t*) – Index of the constraint or 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 *Bound data*

`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 *Bound data*

`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 *Bound data*

`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, so a task containing nonlinear functions cannot be cloned.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `clonedtask` (*MSKtask_t by reference*) – The cloned task. (output)

Return (*MSKrescodee*) – The function response code.

Groups *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 *Scalar variable data*

MSK_computesparsecholesky

```
MSKrescodee (MSKAPI MSK_computesparsecholesky) (
  MSKenv_t env,
  MSKint32t multithread,
  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)
- multithread (*MSKint32t*) – If nonzero then the function may exploit multiple threads. (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

```
MSKrescodee (MSKAPI MSK_conetypetostr) (
  MSKtask_t task,
  MSKconetypee ct,
  char * str)
```

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 (*MSKstring_t*) – String corresponding to the cone type code ct. (output)

Return (*MSKrescodee*) – The function response code.

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 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 *Task management*

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 *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

$$\{\mathtt{subj}[i] \mid i = 0, \ldots, \mathtt{numj} - 1\}$$

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 (*MSKstring_t*) – 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 *Task diagnostics*

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 (*MSKstring_t*) – Is a valid C format string which matches the arguments in .... (input)
- varnumarg (...) – Additional arguments (input)

Return (*MSKrescodee*) – The function response code.

Groups *Logging*

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 (*MSKstring_t*) – File from which the function is called. (input)
- line (unsigned) – Line in the file from which the function is called. (input)

Return (void)

Groups *Memory*

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 (*MSKstring_t*) – File from which the function is called. (input)
- line (unsigned) – Line in the file from which the function is called. (input)

Return (void)

Groups *Memory*

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 *Memory*

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 *Memory*

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.

Parameters

- env (*MSKenv_t*) – The MOSEK environment. (input)
- transa (*MSKtransposee*) – Indicates whether the matrix A must be transposed. (input)
- transb (*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

$$y := \alpha Ax + \beta y,$$

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.

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_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 *Scalar variable data*

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 *Scalar variable data*

MSK_getacolslicetrip

```
MSKrescodee (MSKAPI MSK_getacolslicetrip) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint64t maxnumnz,
  MSKint64t * surp,
  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)
- surp (*MSKint64t by reference*) – Surplus of subscript and coefficient arrays. The required entries are stored sequentially in subi, subj and val starting from position surp away from the end of the arrays. On return surp will be decremented by the total number of non-zeros written. (input/output)
- subi (*MSKint32t* *) – Constraint subscripts. (output)

- subj (*MSKint32t* *) – Column subscripts. (output)
- val (*MSKrealt* *) – Values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Scalar variable data*

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 *Scalar variable data*

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.

`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 *Scalar variable data*

`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 *Scalar variable data*

`MSK_getarowslicetrip`

```
MSKrescodee (MSKAPI MSK_getarowslicetrip) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  MSKint64t maxnumnz,
  MSKint64t * surp,
  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)
- `surp` (*MSKint64t by reference*) – Surplus of subscript and coefficient arrays. The required entries are stored sequentially in `subi`, `subj` and `val` starting from position `surp` away from the end of the arrays. On return `surp` will be decremented by the total number of non-zeros written. (input/output)
- `subi` (*MSKint32t**) – Constraint subscripts. (output)
- `subj` (*MSKint32t**) – Column subscripts. (output)
- `val` (*MSKrealt**) – Values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Scalar variable data*

~~`MSK_getaslice`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getaslice) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t first,
  MSKint32t last,
  MSKint32t maxnumnz,
  MSKint32t * surp,
  MSKint32t * ptrb,
  MSKint32t * ptre,
  MSKint32t * sub,
  MSKrealt * val)
```

Obtains a sequence of rows or columns from A in sparse format.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `accmode` (*MSKaccmodee*) – Defines whether a column slice or a row slice is requested. (input)
- `first` (*MSKint32t*) – Index of the first row or column in the sequence. (input)
- `last` (*MSKint32t*) – Index of the last row or column in the sequence **plus one**. (input)
- `maxnumnz` (*MSKint32t*) – Denotes the length of the arrays `sub` and `val`. (input)
- `surp` (*MSKint32t by reference*) – Surplus of subscript and coefficient arrays. The required entries are stored sequentially in `sub` and `val` starting from position `surp` away from the end of the arrays. Upon return `surp` will be decremented by the total number of non-zeros written. (input/output)
- `ptrb` (*MSKint32t**) – `ptrb[t]` is an index pointing to the first element in the t-th row or column obtained. (output)

- ptre (*MSKint32t* *) – ptre[t] is an index pointing to the last element plus one in the t-th row or column obtained. (output)
- sub (*MSKint32t* *) – Contains the row or column subscripts. (output)
- val (*MSKrealt* *) – Contains the coefficient values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Scalar variable data*

~~MSK_getaslice64~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getaslice64) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t first,
  MSKint32t last,
  MSKint64t maxnumnz,
  MSKint64t * surp,
  MSKint64t * ptrb,
  MSKint64t * ptre,
  MSKint32t * sub,
  MSKrealt * val)
```

Obtains a sequence of rows or columns from A in sparse format.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- accmode (*MSKaccmodee*) – Defines whether a column slice or a row slice is requested. (input)
- first (*MSKint32t*) – Index of the first row or column in the sequence. (input)
- last (*MSKint32t*) – Index of the last row or column in the sequence **plus one**. (input)
- maxnumnz (*MSKint64t*) – Denotes the length of the arrays sub and val. (input)
- surp (*MSKint64t by reference*) – Surplus of subscript and coefficient arrays. The required entries are stored sequentially in sub and val starting from position surp away from the end of the arrays. Upon return surp will be decremented by the total number of non-zeros written. (input/output)
- ptrb (*MSKint64t* *) – ptrb[t] is an index pointing to the first element in the t-th row or column obtained. (output)
- ptre (*MSKint64t* *) – ptre[t] is an index pointing to the last element plus one in the t-th row or column obtained. (output)
- sub (*MSKint32t* *) – Contains the row or column subscripts. (output)
- val (*MSKrealt* *) – Contains the coefficient values. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Scalar variable data*

~~MSK_getaslicenumnz~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getaslicenumnz) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t first,
```

```
  MSKint32t last,
  MSKint32t * numnz)
```

Obtains the number of non-zeros in a row or column slice of A.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- accmode (*MSKaccmodee*) – Defines whether non-zeros are counted in a column-slice or a row-slice. (input)
- first (*MSKint32t*) – Index of the first row or column in the sequence. (input)
- last (*MSKint32t*) – Index of the last row or 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.

~~MSK_getaslicenumnz64~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getaslicenumnz64) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t first,
  MSKint32t last,
  MSKint64t * numnz)
```

Obtains the number of non-zeros in a slice of rows or columns of A.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- accmode (*MSKaccmodee*) – Defines whether non-zeros are counted in a column slice or a row slice. (input)
- first (*MSKint32t*) – Index of the first row or column in the sequence. (input)
- last (*MSKint32t*) – Index of the last row or 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.

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

MSK_getbarcidx

```
MSKrescodee (MSKAPI MSK_getbarcidx) (
  MSKtask_t task,
  MSKint64t idx,
  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 *Symmetric matrix variable data*

`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 *Symmetric matrix variable data*

`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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 (get)*

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 (*MSKstring_t*) – The requested name is copied to this buffer. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

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 (*MSKstring_t*) – 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 *Naming*

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 *Naming*

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 (get)*

~~`MSK_getbound`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getbound) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t i,
  MSKboundkeye * bk,
  MSKrealt * bl,
  MSKrealt * bu)
```

Obtains bound information for one constraint or variable.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `accmode` (*MSKaccmodee*) – Defines if operations are performed row-wise (constraint-oriented) or column-wise (variable-oriented). (input)
- `i` (*MSKint32t*) – Index of the constraint or 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 *Bound data*

~~`MSK_getboundslice`~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getboundslice) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t first,
  MSKint32t last,
  MSKboundkeye * bk,
  MSKrealt * bl,
  MSKrealt * bu)
```

Obtains bounds information for a slice of variables or constraints.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)

- accmode (*MSKaccmodee*) – Defines if operations are performed row-wise (constraint-oriented) or column-wise (variable-oriented). (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 *Bound data*

MSK_getbuildinfo

```
MSKrescodee (MSKAPI MSK_getbuildinfo) (
  char * buildstate,
  char * builddate)
```

Obtains build information.

Parameters

- buildstate (*MSKstring_t*) – State of binaries, i.e. a debug, release candidate or final release. (output)
- builddate (*MSKstring_t*) – Date when the binaries were built. (output)

Return (*MSKrescodee*) – The function response code.

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 *Scalar variable data*

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 *Scalar variable data*

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 *Scalar variable data*

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 (*MSKstring_t*) – Symbolic name corresponding to code. (output)
- str (*MSKstring_t*) – Obtains a short description of a response code. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Task management*

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 *Bound data*

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 *Bound data*

MSK_getcone

```
MSKrescodee (MSKAPI MSK_getcone) (
  MSKtask_t task,
  MSKint32t k,
```

```
  MSKconetypee * ct,
  MSKrealt * conepar,
  MSKint32t * nummem,
  MSKint32t * submem)
```

Obtains a cone.

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*) – This argument is currently not used. It 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 *Conic constraint data*

MSK_getconeinfo

```
MSKrescodee (MSKAPI MSK_getconeinfo) (
  MSKtask_t task,
  MSKint32t k,
  MSKconetypee * ct,
  MSKrealt * conepar,
  MSKint32t * nummem)
```

Obtains information about a cone.

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*) – This argument is currently not used. It 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 *Conic constraint data*

MSK_getconename

```
MSKrescodee (MSKAPI MSK_getconename) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t sizename,
  char * name)
```

Obtains the name of a cone.

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 (*MSKstring_t*) – The required name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

MSK_getconenameindex

```
MSKrescodee (MSKAPI MSK_getconenameindex) (
  MSKtask_t task,
  const char * somename,
  MSKint32t * asgn,
  MSKint32t * index)
```

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 (*MSKstring_t*) – 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 *Naming*

MSK_getconenamelen

```
MSKrescodee (MSKAPI MSK_getconenamelen) (
  MSKtask_t task,
  MSKint32t i,
  MSKint32t * len)
```

Obtains the length of the name of a cone.

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 *Naming*

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` (*MSKstring_t*) – The required name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

`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` (*MSKstring_t*) – 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 *Naming*

`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 *Naming*

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 *Scalar variable data*

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 *Symmetric matrix variable data*

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 *Optimizer 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 (get)*

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*

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_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 $\overline{S}_{\mathtt{sub}[k]} \in \mathcal{S}_+$. (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

$$\rho(x,l) = \left\{ \begin{array}{ll} -x, & l > -\infty, \\ |x|, & \text{otherwise.} \end{array} \right.$$

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`

```
MSKrescodee (MSKAPI MSK_getdviolcones) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t num,
  const MSKint32t * sub,
  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

$$\left\{ \begin{array}{ll} \max(0, (\|s_n^x\|_{2:n}^* - (s_n^x)_1^*)/\sqrt{2}, & (s_n^x)^* \geq -\|(s_n^x)_{2:n}^*\|, \\ \|(s_n^x)^*\|, & \text{otherwise.} \end{array} \right.$$

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

$$\rho(x,l) = \begin{cases} -x, & l > -\infty, \\ |x|, & \text{otherwise} \end{cases}$$

and $\tau = 0$ if the solution is a certificate of primal infeasibility and $\tau = 1$ otherwise. The formula for computing the violation is only shown for the linear case but is generalized appropriately for the more general problems. Both when the solution is a certificate of primal infeasibility or when it is a dual feasible solution the violation should be small.

Parameters

- `task` (*MSKtask_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.

`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 Section *Analyzing Infeasible Problems*.

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 diagnostics*

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 (*MSKstring_t*) – Name of the information item. (input)
- infindex (*MSKint32t by reference*) – The item index. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Optimizer 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 *Optimizer 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 (*MSKstring_t*) – Name of the information item. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Optimizer statistics*

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 *Optimizer 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 (get)*

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` (*MSKstring_t*) – Returns the last error message reported in the task. (output)

Return (*MSKrescodee*) – The function response code.

`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` (*MSKstring_t*) – Returns the last error message reported in the task. (output)

Return (*MSKrescodee*) – The function response code.

`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 *Scalar variable data*

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 *Optimizer 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 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 *Naming*

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 *Scalar variable data*

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 *Scalar variable data*

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 *Symmetric matrix variable data*

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 *Linear constraint data*

MSK_getmaxnumcone

```
MSKrescodee (MSKAPI MSK_getmaxnumcone) (
  MSKtask_t task,
  MSKint32t * maxnumcone)
```

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 *Task management*

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 *Scalar variable data*

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 *Scalar variable data*

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 *Scalar variable data*

`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 *Memory*

`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` (*MSKstring_t*) – 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 *Optimizer 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 (*MSKstring_t*) – Name of a parameter. (input)
- parvalue (*MSKrealt by reference*) – Parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters (get)*

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 (*MSKstring_t*) – 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 *Optimizer 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 (*MSKstring_t*) – Name of a parameter. (input)
- parvalue (*MSKint32t by reference*) – Parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Optimizer statistics*

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 (*MSKstring_t*) – 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 (*MSKstring_t*) – Parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters (get)*

MSK_getnastrparamal

```
MSKrescodee (MSKAPI MSK_getnastrparamal) (
  MSKtask_t task,
  const char * paramname,
  MSKint32t numaddchr,
  MSKstring_t * value)
```

Obtains the value of a named string parameter.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- paramname (*MSKstring_t*) – Name of a parameter. (input)
- numaddchr (*MSKint32t*) – Number of additional characters for which room is left in value. (input)
- value (*MSKstring_t**) – 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 (get)*

MSK_getnlfunc

```
MSKrescodee (MSKAPI MSK_getnlfunc) (
  MSKtask_t task,
  MSKuserhandle_t * nlhandle,
  MSKnlgetspfunc * nlgetsp,
  MSKnlgetvafunc * nlgetva)
```

This function is used to retrieve the nonlinear callback functions. If NULL no nonlinear callback function exists.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- nlhandle (*MSKuserhandle_t by reference*) – Retrieve the pointer to the user-defined data structure. This structure is passed to the functions nlgetsp and nlgetva whenever those two functions are called. (input/output)
- nlgetsp (*MSKnlgetspfunc by reference*) – Retrieve the pointer to the function which provides information about the structure of the nonlinear part of the optimization problem. (output)
- nlgetva (*MSKnlgetvafunc**) – Retrieve the function which is used to evaluate the nonlinear function in the optimization problem at a given point. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Callback*

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 *Scalar variable data*

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 *Scalar variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 *Linear constraint data*

MSK_getnumcone

```
MSKrescodee (MSKAPI MSK_getnumcone) (
  MSKtask_t task,
  MSKint32t * numcone)
```

Obtains the number of cones.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- numcone (*MSKint32t by reference*) – Number of conic constraints. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Conic constraint data*

MSK_getnumconemem

```
MSKrescodee (MSKAPI MSK_getnumconemem) (
  MSKtask_t task,
  MSKint32t k,
  MSKint32t * nummem)
```

Obtains the number of members in a cone.

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 *Conic constraint data*

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 *Scalar variable data*

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 *Parameter management*

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 *Scalar variable data*

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 *Scalar variable data*

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 *Scalar variable data*

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 *Scalar variable data*

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 *Scalar variable data*

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 *Scalar variable data*

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 (*MSKstring_t*) – Assigned the objective name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

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 *Naming*

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 *Objective data*

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 *Parameter management*

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 (*MSKstring_t*) – Parameter name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameter management*

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*

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 *Task diagnostics*

MSK_getprosta

```
MSKrescodee (MSKAPI MSK_getprosta) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKprostae * prosta)
```

Obtains the problem status.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)

- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- prosta (*MSKprostae by reference*) – Problem status. (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

```
MSKrescodee (MSKAPI MSK_getpviolcones) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t num,
  const MSKint32t * sub,
  MSKrealt * viol)
```

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_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 $x_{\mathtt{sub[k]}}$. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution information*

MSK_getqconk

```
MSKrescodee (MSKAPI MSK_getqconk) (
  MSKtask_t task,
  MSKint32t k,
  MSKint32t maxnumqcnz,
  MSKint32t * qcsurp,
  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)
- qcsurp (*MSKint32t by reference*) – Surplus of subscript and coefficient arrays. The required entries are stored sequentially in qssubi, qssubj and qsval starting from position qcsurp away from the end of the arrays. On return qcsurp will be decremented by the total number of non-zeros written. (input/output)
- 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 *Scalar variable data*

MSK_getqconk64

```
MSKrescodee (MSKAPI MSK_getqconk64) (
  MSKtask_t task,
  MSKint32t k,
  MSKint64t maxnumqcnz,
  MSKint64t * qcsurp,
  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)
- qcsurp (*MSKint64t by reference*) – Surplus of subscript and coefficient arrays. The required entries are stored sequentially in qcsubi, qcsubj and qcval starting from position qcsurp away from the end of the arrays. On return qcsurp will be decremented by the total number of non-zeros written. (input/output)
- 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 *Scalar variable data*

MSK_getqobj

```
MSKrescodee (MSKAPI MSK_getqobj) (
  MSKtask_t task,
  MSKint32t maxnumqonz,
  MSKint32t * qosurp,
  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)
- qosurp (*MSKint32t by reference*) – Surplus of subscript and coefficient arrays. The required entries are stored sequentially in qosubi, qosubj and qoval starting from position qosurp away from the end of the arrays. On return qosurp will be decremented by the total number of non-zeros written. (input/output)

- 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 *Scalar variable data*

MSK_getqobj64

```
MSKrescodee (MSKAPI MSK_getqobj64) (
  MSKtask_t task,
  MSKint64t maxnumqonz,
  MSKint64t * qosurp,
  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)
- qosurp (*MSKint64t by reference*) – Surplus of subscript and coefficient arrays. The required entries are stored sequentially in qosubi, qosubj and qoval starting from position qosurp away from the end of the arrays. On return qosurp will be decremented by the total number of non-zeros written. (input/output)
- 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 *Scalar variable data*

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 *Scalar variable data*

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.

$$\texttt{redcosts}[j - \texttt{first}] = (s_l^x)_j - (s_u^x)_j, \ j = \texttt{first}, \ldots, \texttt{last} - 1 \tag{16.2}$$

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 (get)*

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 *Task management*

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 (get)*

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 (get)*

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 (get)*

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 (get)*

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 (get)*

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 (get)*

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 (get)*

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 (get)*

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 (get)*

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 (get)*

MSK_getsolsta

```
MSKrescodee (MSKAPI MSK_getsolsta) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKsolstae * solsta)
```

Obtains the solution status.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- solsta (*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 * prosta,
  MSKsolstae * solsta,
  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

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

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:

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

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)
- prosta (*MSKprostae by reference*) – Problem status. (output)
- solsta (*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 (get)*

~~MSK_getsolutioni~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_getsolutioni) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t i,
  MSKsoltypee whichsol,
  MSKstakeye * sk,
  MSKrealt * x,
  MSKrealt * sl,
  MSKrealt * su,
  MSKrealt * sn)
```

Obtains the primal and dual solution information for a single constraint or variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- accmode (*MSKaccmodee*) – Defines whether solution information for a constraint or for a variable is retrieved. (input)
- i (*MSKint32t*) – Index of the constraint or variable. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)

- sk (*MSKstakeye by reference*) – Status key of the constraint of variable. (output)
- x (*MSKrealt by reference*) – Solution value of the primal variable. (output)
- sl (*MSKrealt by reference*) – Solution value of the dual variable associated with the lower bound. (output)
- su (*MSKrealt by reference*) – Solution value of the dual variable associated with the upper bound. (output)
- sn (*MSKrealt by reference*) – Solution value of the dual variable associated with the cone constraint. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution (get)*

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_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 (get)*

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 *Scalar variable data*

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 (*MSKstring_t*) – Parameter value. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters (get)*

MSK_getstrparamal

```
MSKrescodee (MSKAPI MSK_getstrparamal) (
  MSKtask_t task,
  MSKsparame param,
  MSKint32t numaddchr,
  MSKstring_t * 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 (*MSKstring_t 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 (get)*

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 *Parameters (get)*

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 (get)*

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 (get)*

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 (get)*

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 (get)*

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 (*MSKstring_t*) – Name of the ith symbolic constant. (output)
- value (*MSKint32t by reference*) – The corresponding value. (output)

Return (*MSKrescodee*) – The function response code.

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 *Parameter management*

MSK_getsymmatinfo

```
MSKrescodee (MSKAPI MSK_getsymmatinfo) (
  MSKtask_t task,
  MSKint64t idx,
  MSKint32t * dim,
  MSKint64t * nz,
  MSKsymmattypee * type)
```

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)
- type (*MSKsymmattypee by reference*) – Returns the type of the requested matrix. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Scalar variable data*

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` (*MSKstring_t*) – Returns the task name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

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 *Naming*

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 *Bound data*

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 *Bound data*

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 (*MSKstring_t*) – Returns the required name. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

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 (*MSKstring_t*) – 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 *Naming*

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 *Naming*

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 *Scalar variable data*

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 *Scalar variable data*

MSK_getversion

```
MSKrescodee (MSKAPI MSK_getversion) (
  MSKint32t * major,
  MSKint32t * minor,
  MSKint32t * build,
  MSKint32t * revision)
```

Obtains **MOSEK** version information.

Parameters

- `major` (*MSKint32t by reference*) – Major version number. (output)
- `minor` (*MSKint32t by reference*) – Minor version number. (output)
- `build` (*MSKint32t by reference*) – Build number. (output)
- `revision` (*MSKint32t by reference*) – Revision number. (output)

Return (*MSKrescodee*) – The function response code.

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 (get)*

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 (get)*

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 (get)*

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 (get)*

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 (get)*

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 (get)*

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 $\texttt{basis}[i] \leq numcon - 1$, then $x^c_{\texttt{basis}[i]}$ is in the basis at position i, otherwise $x_{\texttt{basis}[i]-\texttt{numcon}}$ is in the basis at position i. (output)

Return (*MSKrescodee*) – The function response code.

Groups *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 *Task management*

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 *Task management*

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 (*MSKstring_t*) – The symbolic name corresponding to whichvalue. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameter management*

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 (*MSKstring_t*) – 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 *Parameter management*

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 (*MSKstring_t*) – 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 *Parameter management*

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 (*MSKstring_t*) – 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 *Parameter management*

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 *Environment management*

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` (*MSKstring_t*) – 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` (*MSKstring_t*) – 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 *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` (*MSKstring_t*) – A user-defined file debug file. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment management*

MSK_makeenvalloc

```
MSKrescodee (MSKAPI MSK_makeenvalloc) (
  MSKenv_t * env,
  MSKuserhandle_t usrptr,
  MSKmallocfunc usrmalloc,
  MSKcallocfunc usrcalloc,
  MSKreallocfunc usrrealloc,
  MSKfreefunc usrfree,
  const char * dbgfile)
```

Creates a new **MOSEK** environment with user-defined memory management functions. The environment must be shared among all tasks in a program.

Parameters

- `env` (*MSKenv_t by reference*) – The MOSEK environment. (output)
- `usrptr` (*MSKuserhandle_t*) – A pointer to a user-defined data structure. The pointer is fed into `usrmalloc` and `usrfree`. (input)
- `usrmalloc` (*MSKmallocfunc*) – A user-defined `malloc` function or a NULL pointer. (input)
- `usrcalloc` (*MSKcallocfunc*) – A user-defined `calloc` function or a NULL pointer. (input)
- `usrrealloc` (*MSKreallocfunc*) – A user-defined `realloc` function or a NULL pointer. (input)
- `usrfree` (*MSKfreefunc*) – A user-defined `free` function which is used to deallocate space allocated by `usrmalloc`. This function must be defined if `usrmalloc!=null`. (input)
- `dbgfile` (*MSKstring_t*) – A user-defined file debug file. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment 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 *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 *Task diagnostics*

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_optimizermt

```
MSKrescodee (MSKAPI MSK_optimizermt) (
  MSKtask_t task,
  const char * server,
  const char * port,
  MSKrescodee * trmcode)
```

Offload the optimization task to a solver server defined by server:port. The call will block until a result is available or the connection closes.

If the string parameter *MSK_SPAR_REMOTE_ACCESS_TOKEN* is not blank, it will be passed to the server as authentication.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- server (*MSKstring_t*) – Name or IP address of the solver server. (input)

- port (*MSKstring_t*) – Network port of the solver server. (input)
- trmcode (*MSKrescodee by reference*) – Is either *MSK_RES_OK* or a termination response code. (output)

Return (*MSKrescodee*) – The function response code.

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 *Task diagnostics*

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 diagnostics*

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_printdata

```
MSKrescodee (MSKAPI MSK_printdata) (
  MSKtask_t task,
  MSKstreamtype whichstream,
  MSKint32t firsti,
  MSKint32t lasti,
  MSKint32t firstj,
  MSKint32t lastj,
  MSKint32t firstk,
  MSKint32t lastk,
  MSKint32t c,
  MSKint32t qo,
  MSKint32t a,
  MSKint32t qc,
  MSKint32t bc,
  MSKint32t bx,
  MSKint32t vartype,
  MSKint32t cones)
```

Prints a part of the problem data to a stream. This function is normally used for debugging purposes only, e.g. to verify that the correct data has been inputted.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichstream (*MSKstreamtypee*) – Index of the stream. (input)
- firsti (*MSKint32t*) – Index of first constraint for which data should be printed. (input)
- lasti (*MSKint32t*) – Index of last constraint plus 1 for which data should be printed. (input)
- firstj (*MSKint32t*) – Index of first variable for which data should be printed. (input)
- lastj (*MSKint32t*) – Index of last variable plus 1 for which data should be printed. (input)
- firstk (*MSKint32t*) – Index of first cone for which data should be printed. (input)
- lastk (*MSKint32t*) – Index of last cone plus 1 for which data should be printed. (input)
- c (*MSKint32t*) – If non-zero c is printed. (input)
- qo (*MSKint32t*) – If non-zero Q^o is printed. (input)
- a (*MSKint32t*) – If non-zero A is printed. (input)
- qc (*MSKint32t*) – If non-zero Q^k is printed for the relevant constraints. (input)

- bc (*MSKint32t*) – If non-zero the constraint bounds are printed. (input)
- bx (*MSKint32t*) – If non-zero the variable bounds are printed. (input)
- vartype (*MSKint32t*) – If non-zero the variable types are printed. (input)
- cones (*MSKint32t*) – If non-zero the conic data is printed. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Task diagnostics*

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 *Task diagnostics*

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 (*MSKstring_t*) – String corresponding to the problem type key probtype. (output)

Return (*MSKrescodee*) – The function response code.

MSK_prostatostr

```
MSKrescodee (MSKAPI MSK_prostatostr) (
  MSKtask_t task,
  MSKprostae prosta,
  char * str)
```

Obtains a string containing the name of a given problem status.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- prosta (*MSKprostae*) – Problem status. (input)
- str (*MSKstring_t*) – String corresponding to the status key prosta. (output)

Return (*MSKrescodee*) – The function response code.

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 *Scalar variable data*

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 *Scalar variable data*

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 *Scalar variable data*

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:

$$\begin{array}{ll} \texttt{for} & i = \texttt{first}, \ldots, \texttt{last} - 1 \\ & a_{\texttt{asub}[k],i} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i], \ldots, \texttt{ptre}[i] - 1. \end{array}$$

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 *Scalar variable data*

MSK_putacolslice64

```
MSKrescodee (MSKAPI MSK_putacolslice64) (
  MSKtask_t task,
  MSKint32t first,
  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:

$$\begin{array}{ll} \texttt{for} & i = \texttt{first}, \ldots, \texttt{last} - 1 \\ & a_{\texttt{asub}[k],i} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i], \ldots, \texttt{ptre}[i] - 1. \end{array}$$

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 (*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 *Scalar variable data*

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

$$a_{i,j} = \texttt{aij}.$$

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 *Scalar variable data*

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

$$a_{\mathtt{subi[k]},\mathtt{subj[k]}} = \mathtt{valij[k]}, \quad k = 0, \ldots, num - 1.$$

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 *Scalar variable data*

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

$$a_{\mathtt{subi[k]},\mathtt{subj[k]}} = \mathtt{valij[k]}, \quad k = 0, \ldots, num - 1.$$

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 *Scalar variable data*

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

$$a_{\mathtt{i},\mathtt{subi}[k]} = \mathtt{vali}[k], \quad k = 0, \ldots, \mathtt{nzi} - 1.$$

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 *Scalar variable data*

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:

$$\begin{array}{ll} \mathtt{for} & i = 0, \ldots, num - 1 \\ & a_{\mathtt{sub}[i],\mathtt{asub}[k]} = \mathtt{aval}[k], \quad k = \mathtt{ptrb}[i], \ldots, \mathtt{ptre}[i] - 1. \end{array}$$

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- num (*MSKint32t*) – Number of 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 *Scalar variable data*

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 *Scalar variable data*

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 columns are set to zero and then updated with:

$$\begin{array}{ll} \texttt{for} & i = \texttt{first}, \ldots, \texttt{last} - 1 \\ & a_{\texttt{sub}[i],\texttt{asub}[k]} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i], \ldots, \texttt{ptre}[i] - 1. \end{array}$$

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- 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 *Scalar variable data*

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 columns are set to zero and then updated with:

$$\begin{array}{ll} \texttt{for} & i = \texttt{first}, \ldots, \texttt{last} - 1 \\ & a_{\texttt{sub}[i], \texttt{asub}[k]} = \texttt{aval}[k], \quad k = \texttt{ptrb}[i], \ldots, \texttt{ptre}[i] - 1. \end{array}$$

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- 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 *Scalar variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 *Symmetric matrix variable data*

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 (put)*

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` (*MSKstring_t*) – The variable name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

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 (put)*

~~MSK_putbound~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_putbound) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t i,
  MSKboundkeye bk,
  MSKrealt bl,
  MSKrealt bu)
```

Changes the bound for either one constraint or one variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- accmode (*MSKaccmodee*) – Defines whether the bound for a constraint (*MSK_ACC_CON*) or variable (*MSK_ACC_VAR*) is changed. (input)
- i (*MSKint32t*) – Index of the constraint or variable. (input)
- bk (*MSKboundkeye*) – New bound key. (input)
- bl (*MSKrealt*) – New lower bound. (input)
- bu (*MSKrealt*) – New upper bound. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Bound data*

~~MSK_putboundlist~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_putboundlist) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t num,
  const MSKint32t * sub,
  const MSKboundkeye * bk,
  const MSKrealt * bl,
  const MSKrealt * bu)
```

Changes the bounds of constraints or variables.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- accmode (*MSKaccmodee*) – Defines whether bounds for constraints (*MSK_ACC_CON*) or variables (*MSK_ACC_VAR*) are changed. (input)
- num (*MSKint32t*) – Number of bounds that should be changed. (input)
- sub (*MSKint32t* *) – Subscripts of the constraints or variables that should be changed. (input)
- bk (*MSKboundkeye* *) – Bound keys. (input)
- bl (*MSKrealt* *) – Values for lower bounds. (input)
- bu (*MSKrealt* *) – Values for upper bounds. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Bound data*

~~MSK_putboundslice~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_putboundslice) (
  MSKtask_t task,
  MSKaccmodee con,
  MSKint32t first,
  MSKint32t last,
  const MSKboundkeye * bk,
  const MSKrealt * bl,
  const MSKrealt * bu)
```

Changes the bounds for a slice of constraints or variables.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- con (*MSKaccmodee*) – Defines whether bounds for constraints (*MSK_ACC_CON*) or variables (*MSK_ACC_VAR*) are changed. (input)
- first (*MSKint32t*) – First index in the sequence. (input)
- last (*MSKint32t*) – Last index plus 1 in the sequence. (input)
- bk (*MSKboundkeye* *) – Bound keys. (input)
- bl (*MSKrealt* *) – Values for lower bounds. (input)
- bu (*MSKrealt* *) – Values for upper bounds. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Bound data*

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 *Objective data*

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 *Scalar variable data*

`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

$$c_{\mathtt{subj[t]}} = \mathtt{val}[\mathtt{t}], \quad t = 0, \ldots, num - 1.$$

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 *Scalar variable data*

`MSK_putconbound`

```
MSKrescodee (MSKAPI MSK_putconbound) (
  MSKtask_t task,
  MSKint32t i,
  MSKboundkeye bk,
  MSKrealt bl,
  MSKrealt bu)
```

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)
- bk (*MSKboundkeye*) – New bound key. (input)
- bl (*MSKrealt*) – New lower bound. (input)
- bu (*MSKrealt*) – New upper bound. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Bound data*

MSK_putconboundlist

```
MSKrescodee (MSKAPI MSK_putconboundlist) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * sub,
  const MSKboundkeye * bk,
  const MSKrealt * bl,
  const MSKrealt * bu)
```

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)
- bk (*MSKboundkeye* *) – Bound keys. (input)
- bl (*MSKrealt* *) – Values for lower bounds. (input)
- bu (*MSKrealt* *) – Values for upper bounds. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Bound data*

MSK_putconboundslice

```
MSKrescodee (MSKAPI MSK_putconboundslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
  const MSKboundkeye * bk,
  const MSKrealt * bl,
  const MSKrealt * bu)
```

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)
- bk (*MSKboundkeye* *) – Bound keys. (input)
- bl (*MSKrealt* *) – Values for lower bounds. (input)
- bu (*MSKrealt* *) – Values for upper bounds. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Linear constraint data*, *Bound data*

MSK_putcone

```
MSKrescodee (MSKAPI MSK_putcone) (
  MSKtask_t task,
  MSKint32t k,
  MSKconetypee ct,
  MSKrealt conepar,
  MSKint32t nummem,
  const MSKint32t * submem)
```

Replaces a conic constraint.

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*) – This argument is currently not used. It 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 *Conic constraint data*

MSK_putconename

```
MSKrescodee (MSKAPI MSK_putconename) (
  MSKtask_t task,
  MSKint32t j,
  const char * name)
```

Sets the name of a cone.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- j (*MSKint32t*) – Index of the cone. (input)
- name (*MSKstring_t*) – The name of the cone. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

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 (*MSKstring_t*) – The name of the constraint. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

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 *Scalar variable data*

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 (put)*

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 *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 (put)*

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 *Environment management*

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 *Environment management*

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` (*MSKstring_t*) – A path specifying where to search for the license. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Environment management*

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 *Environment management*

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 *Scalar variable data*

`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 *Symmetric matrix variable data*

`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 *Task management*

MSK_putmaxnumcone

```
MSKrescodee (MSKAPI MSK_putmaxnumcone) (
  MSKtask_t task,
  MSKint32t maxnumcone)
```

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 *Task management*

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 *Scalar variable data*

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 *Scalar variable data*

`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` (*MSKstring_t*) – Name of a parameter. (input)
- `parvalue` (*MSKrealt*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters (put)*

`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` (*MSKstring_t*) – Name of a parameter. (input)
- `parvalue` (*MSKint32t*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters (put)*

`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 (*MSKstring_t*) – Name of a parameter. (input)
- parvalue (*MSKstring_t*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters (put)*

MSK_putnlfunc

```
MSKrescodee (MSKAPI MSK_putnlfunc) (
  MSKtask_t task,
  MSKuserhandle_t nlhandle,
  MSKnlgetspfunc nlgetsp,
  MSKnlgetvafunc nlgetva)
```

This function is used to communicate the nonlinear function information in a general convex optimization problem to **MOSEK**.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- nlhandle (*MSKuserhandle_t*) – A pointer to a user-defined data structure. It is passed to the functions nlgetsp and nlgetva whenever those two functions called. (input)
- nlgetsp (*MSKnlgetspfunc*) – Pointer to a user-defined function computing non-linear structural information. (input)
- nlgetva (*MSKnlgetvafunc*) – Pointer to user-defined function which evaluates the nonlinear function in the optimization problem at a given point. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Callback*

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 (*MSKstring_t*) – Name of the objective. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

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 *Objective data*

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 (*MSKstring_t*) – Parameter name. (input)
- parvalue (*MSKstring_t*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters (put)*

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:

$$q_{\texttt{qcsubi[t]},\texttt{qcsubj[t]}}^{\texttt{qcsubk[t]}} = q_{\texttt{qcsubj[t]},\texttt{qcsubi[t]}}^{\texttt{qcsubk[t]}} = q_{\texttt{qcsubj[t]},\texttt{qcsubi[t]}}^{\texttt{qcsubk[t]}} + \texttt{qcval[t]},$$

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 *Scalar variable data*

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 *Scalar variable data*

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:

$$q^o_{\mathtt{qosubi[t]},\mathtt{qosubj[t]}} = q^o_{\mathtt{qosubj[t]},\mathtt{qosubi[t]}} = q^o_{\mathtt{qosubj[t]},\mathtt{qosubi[t]}} + \mathtt{qoval}[\mathtt{t}],$$

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 *Scalar variable data*

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

$$q_{ij}^o = q_{ji}^o = \mathtt{qoij}.$$

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^o_{ij}. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Scalar variable data*

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*

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 (put)*

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 (put)*

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 (put)*

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 (put)*

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 (put)*

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 (put)*

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 (put)*

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 (*MSKsoltype*) – 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 (*MSKrescode*) – The function response code.

Groups *Solution (put)*

MSK_putsnx

```
MSKrescode (MSKAPI MSK_putsnx) (
  MSKtask_t task,
  MSKsoltype whichsol,
  const MSKrealt * sux)
```

Sets the s_n^x vector for a solution.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- whichsol (*MSKsoltype*) – Selects a solution. (input)
- sux (*MSKrealt* *) – Dual variables corresponding to the upper bounds on the variables. (input)

Return (*MSKrescode*) – The function response code.

Groups *Solution (put)*

MSK_putsnxslice

```
MSKrescode (MSKAPI MSK_putsnxslice) (
  MSKtask_t task,
  MSKsoltype 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 (*MSKsoltype*) – 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 (put)*

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 (put)*

~~MSK_putsolutioni~~ *Deprecated*

```
MSKrescodee (MSKAPI MSK_putsolutioni) (
  MSKtask_t task,
  MSKaccmodee accmode,
  MSKint32t i,
```

```
  MSKsoltypee whichsol,
  MSKstakeye sk,
  MSKrealt x,
  MSKrealt sl,
  MSKrealt su,
  MSKrealt sn)
```

Sets the primal and dual solution information for a single constraint or variable.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- accmode (*MSKaccmodee*) – Defines whether solution information for a constraint (*MSK_ACC_CON*) or for a variable (*MSK_ACC_VAR*) is modified. (input)
- i (*MSKint32t*) – Index of the constraint or variable. (input)
- whichsol (*MSKsoltypee*) – Selects a solution. (input)
- sk (*MSKstakeye*) – Status key of the constraint or variable. (input)
- x (*MSKrealt*) – Solution value of the primal constraint or 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 (put)*

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.

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 (*MSKstring_t*) – Parameter value. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Parameters (put)*

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 (put)*

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 (put)*

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 (put)*

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 (put)*

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 (*MSKstring_t*) – Name assigned to the task. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

MSK_putvarbound

```
MSKrescodee (MSKAPI MSK_putvarbound) (
  MSKtask_t task,
  MSKint32t j,
  MSKboundkeye bk,
  MSKrealt bl,
  MSKrealt bu)
```

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)
- bk (*MSKboundkeye*) – New bound key. (input)
- bl (*MSKrealt*) – New lower bound. (input)
- bu (*MSKrealt*) – New upper bound. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Bound data*

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 *Bound data*

MSK_putvarboundslice

```
MSKrescodee (MSKAPI MSK_putvarboundslice) (
  MSKtask_t task,
  MSKint32t first,
  MSKint32t last,
```

```
  const MSKboundkeye * bk,
  const MSKrealt * bl,
  const MSKrealt * bu)
```

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)
- bk (*MSKboundkeye* *) – Bound keys. (input)
- bl (*MSKrealt* *) – Values for lower bounds. (input)
- bu (*MSKrealt* *) – Values for upper bounds. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Scalar variable data*

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 (*MSKstring_t*) – The variable name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Naming*

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 *Scalar variable data*

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 *Scalar variable data*

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 (put)*

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 (put)*

MSK_putxx

```
MSKrescodee (MSKAPI MSK_putxx) (
  MSKtask_t task,
  MSKsoltype 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 (put)*

MSK_putxxslice

```
MSKrescodee (MSKAPI MSK_putxxslice) (
  MSKtask_t task,
  MSKsoltypee whichsol,
  MSKint32t first,
  MSKint32t last,
  const 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. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Solution (put)*

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 (put)*

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 (put)*

MSK_readdata

```
MSKrescodee (MSKAPI MSK_readdata) (
  MSKtask_t task,
  const char * filename)
```

Reads an optimization problem and associated data from a file.

The data file format is determined by the *MSK_IPAR_READ_DATA_FORMAT* parameter. By default the parameter has the value *MSK_DATA_FORMAT_EXTENSION* indicating that the extension of the input file should determine the file type. 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 (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Data file*

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 (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Data file*

MSK_readdataformat

```
MSKrescodee (MSKAPI MSK_readdataformat) (
  MSKtask_t task,
  const char * filename,
  int format,
  int compress)
```

Reads an optimization problem and associated data from a file.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- filename (*MSKstring_t*) – A valid file name. (input)
- format (*MSKdataformate*) – File data format. (input)
- compress (*MSKcompresstypee*) – File compression type. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Data file*

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 (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Data file*

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 (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Data file*

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 *Task diagnostics*

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 (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

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 *Symmetric matrix variable data*

MSK_removecones

```
MSKrescodee (MSKAPI MSK_removecones) (
  MSKtask_t task,
  MSKint32t num,
  const MSKint32t * subset)
```

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 *Conic constraint data*

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 *Linear constraint data*

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 *Scalar variable data*

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.

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 *Parameter management*

MSK_sktostr

```
MSKrescodee (MSKAPI MSK_sktostr) (
  MSKtask_t task,
  MSKstakeye sk,
  char * str)
```

Obtains an explanatory string corresponding to a status key.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- sk (*MSKstakeye*) – A valid status key. (input)
- str (*MSKstring_t*) – String corresponding to the status key sk. (output)

Return (*MSKrescodee*) – The function response code.

MSK_solstatostr

```
MSKrescodee (MSKAPI MSK_solstatostr) (
  MSKtask_t task,
  MSKsolstae solsta,
  char * str)
```

Obtains an explanatory string corresponding to a solution status.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- solsta (*MSKsolstae*) – Solution status. (input)
- str (*MSKstring_t*) – String corresponding to the solution status solsta. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Solution (put)*

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 *Task diagnostics*

MSK_solvewithbasis

```
MSKrescodee (MSKAPI MSK_solvewithbasis) (
  MSKtask_t task,
  MSKint32t transp,
  MSKint32t * numnz,
  MSKint32t * sub,
  MSKrealt * val)
```

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

$$B\overline{X} = b \tag{16.3}$$

or the system

$$B^T\overline{X} = b \tag{16.4}$$

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 (*MSKint32t*) – If this argument is zero, then (16.3) is solved, if non-zero then (16.4) is solved. (input)
- numnz (*MSKint32t by reference*) – As input it is the number of non-zeros in b. As output it is the number of non-zeros in $\overline{X}$. (input/output)

- 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)

Return (*MSKrescodee*) – The function response code.

Groups *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_strdupdbgtask

```
char * (MSKAPI MSK_strdupdbgtask) (
  MSKtask_t task,
  const char * str,
  const char * file,
  const unsigned line)
```

Make a copy of a string. The string created by this procedure must be freed by *MSK_freetask*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- str (*MSKstring_t*) – String that should be copied. (input)
- file (*MSKstring_t*) – File from which the function is called. (input)
- line (unsigned) – Line in the file from which the function is called. (input)

Return (*MSKstring_t*) – A copy of the given string.

MSK_strduptask

```
char * (MSKAPI MSK_strduptask) (
  MSKtask_t task,
  const char * str)
```

Make a copy of a string. The string created by this procedure must be freed by *MSK_freetask*.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- str (*MSKstring_t*) – String that should be copied. (input)

Return (*MSKstring_t*) – A copy of the given string.

MSK_strtoconetype

```
MSKrescodee (MSKAPI MSK_strtoconetype) (
  MSKtask_t task,
  const char * str,
  MSKconetypee * conetype)
```

Obtains cone type code corresponding to a cone type string.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- str (*MSKstring_t*) – 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.

MSK_strtosk

```
MSKrescodee (MSKAPI MSK_strtosk) (
  MSKtask_t task,
  const char * str,
  MSKint32t * sk)
```

Obtains the status key corresponding to an explanatory string.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- str (*MSKstring_t*) – Status key string. (input)
- sk (*MSKint32t by reference*) – Status key corresponding to the string. (output)

Return (*MSKrescodee*) – The function response code.

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 (*MSKstring_t*) – Symbolic name. (input)
- value (*MSKstring_t*) – The corresponding value. (output)

Return (*MSKbooleant*) – Indicates if the symbolic name has been converted.

Groups *Parameter management*

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 $k \le n$, 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.

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

```
MSKrescodee (MSKAPI MSK_toconic) (
  MSKtask_t task)
```

This function tries to reformulate a given Quadratically Constrained Quadratic Optimization problem (QCQP) 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.

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 *Task diagnostics*

`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` (*MSKstring_t*) – The UTF8 input string. (input)

Return (*MSKrescodee*) – The function response code.

`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` (*MSKstring_t*) – The input string converted to a UTF8 string. (output)
- `input` (*MSKwchart* `*`) – The *MSKwchart* input string. (input)

Return (*MSKrescodee*) – The function response code.

`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` (*MSKstring_t*) – Parameter name. (input)
- `partype` (*MSKparametertypee by reference*) – Parameter type. (output)
- `param` (*MSKint32t by reference*) – Which parameter. (output)

Return (*MSKrescodee*) – The function response code.

Groups *Parameter management*

`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.

By default the data file format is determined by the file name extension. This behaviour can be overridden by setting the *MSK_IPAR_WRITE_DATA_FORMAT* parameter. 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*.

Please note that if a general nonlinear function appears in the problem then such function *cannot* be written to file and **MOSEK** will issue a warning.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `filename` (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Data file*

`MSK_writejsonsol`

```
MSKrescodee (MSKAPI MSK_writejsonsol) (
  MSKtask_t task,
  const char * filename)
```

Saves the current solutions and solver information items in a JSON file.

Parameters

- `task` (*MSKtask_t*) – An optimization task. (input)
- `filename` (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Data file*

`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` (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Data file*

`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 (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

Groups *Data file*

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 but does not save callback functions and general non-linear terms.

See section *The Task Format* for a description of the Task format.

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- filename (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

MSK_writetasksolverresult_file

```
MSKrescodee (MSKAPI MSK_writetasksolverresult_file) (
  MSKtask_t task,
  const char * filename)
```

Internal

Parameters

- task (*MSKtask_t*) – An optimization task. (input)
- filename (*MSKstring_t*) – A valid file name. (input)

Return (*MSKrescodee*) – The function response code.

16.4 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*
- *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_MAX_TERMS_PER_LINE*
- *MSK_IPAR_OPF_WRITE_HEADER*
- *MSK_IPAR_OPF_WRITE_HINTS*
- *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_READ_DATA_COMPRESSED*
- *MSK_IPAR_READ_DATA_FORMAT*
- *MSK_IPAR_READ_DEBUG*
- *MSK_IPAR_READ_KEEP_FREE_CON*
- *MSK_IPAR_READ_LP_DROP_NEW_VARS_IN_BOU*
- *MSK_IPAR_READ_LP_QUOTED_NAMES*
- *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_DATA_COMPRESSED*
- *MSK_IPAR_WRITE_DATA_FORMAT*
- *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_LP_FULL_OBJ*
- *MSK_IPAR_WRITE_LP_LINE_WIDTH*
- *MSK_IPAR_WRITE_LP_QUOTED_NAMES*
- *MSK_IPAR_WRITE_LP_STRICT_FORMAT*
- *MSK_IPAR_WRITE_LP_TERMS_PER_LINE*
- *MSK_IPAR_WRITE_MPS_FORMAT*
- *MSK_IPAR_WRITE_MPS_INT*
- *MSK_IPAR_WRITE_PRECISION*
- *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_FILE_NAME*
- *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_NL_MERIT_BAL*
- *MSK_DPAR_INTPNT_NL_TOL_DFEAS*
- *MSK_DPAR_INTPNT_NL_TOL_MU_RED*
- *MSK_DPAR_INTPNT_NL_TOL_NEAR_REL*
- *MSK_DPAR_INTPNT_NL_TOL_PFEAS*
- *MSK_DPAR_INTPNT_NL_TOL_REL_GAP*
- *MSK_DPAR_INTPNT_NL_TOL_REL_STEP*
- *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_METHOD*
- *MSK_IPAR_INTPNT_REGULARIZATION_USE*
- *MSK_IPAR_INTPNT_SCALING*
- *MSK_IPAR_INTPNT_SOLVE_FORM*
- *MSK_IPAR_INTPNT_STARTING_POINT*
- *MSK_IPAR_LOG_INTPNT*

License manager

- *MSK_IPAR_CACHE_LICENSE*
- *MSK_IPAR_LICENSE_DEBUG*
- *MSK_IPAR_LICENSE_PAUSE_TIME*
- *MSK_IPAR_LICENSE_SUPPRESS_EXPIRE_WRNS*
- *MSK_IPAR_LICENSE_TRH_EXPIRY_WRN*
- *MSK_IPAR_LICENSE_WAIT*

Logging

- *MSK_IPAR_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_INFEAS_ANA*
- *MSK_IPAR_LOG_INTPNT*
- *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_DISABLE_TERM_TIME*
- *MSK_DPAR_MIO_MAX_TIME*
- *MSK_DPAR_MIO_NEAR_TOL_ABS_GAP*
- *MSK_DPAR_MIO_NEAR_TOL_REL_GAP*
- *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_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_SELECTION_LEVEL*
- *MSK_IPAR_MIO_HEURISTIC_LEVEL*
- *MSK_IPAR_MIO_MAX_NUM_BRANCHES*
- *MSK_IPAR_MIO_MAX_NUM_RELAXS*
- *MSK_IPAR_MIO_MAX_NUM_SOLUTIONS*
- *MSK_IPAR_MIO_NODE_OPTIMIZER*
- *MSK_IPAR_MIO_NODE_SELECTION*
- *MSK_IPAR_MIO_PERSPECTIVE_REFORMULATE*
- *MSK_IPAR_MIO_PROBING_LEVEL*
- *MSK_IPAR_MIO_RINS_MAX_NODES*
- *MSK_IPAR_MIO_ROOT_OPTIMIZER*
- *MSK_IPAR_MIO_ROOT_REPEAT_PRESOLVE_LEVEL*

- *MSK_IPAR_MIO_VB_DETECTION_LEVEL*

Nonlinear convex method

- *MSK_DPAR_INTPNT_NL_MERIT_BAL*
- *MSK_DPAR_INTPNT_NL_TOL_DFEAS*
- *MSK_DPAR_INTPNT_NL_TOL_MU_RED*
- *MSK_DPAR_INTPNT_NL_TOL_NEAR_REL*
- *MSK_DPAR_INTPNT_NL_TOL_PFEAS*
- *MSK_DPAR_INTPNT_NL_TOL_REL_GAP*
- *MSK_DPAR_INTPNT_NL_TOL_REL_STEP*
- *MSK_DPAR_INTPNT_TOL_INFEAS*
- *MSK_IPAR_CHECK_CONVEXITY*
- *MSK_IPAR_LOG_CHECK_CONVEXITY*

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_INFEAS_ANA*
- *MSK_IPAR_LOG_INTPNT*
- *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_INTPNT_MULTI_THREAD*
- *MSK_IPAR_LICENSE_WAIT*
- *MSK_IPAR_LOG_STORAGE*
- *MSK_IPAR_MIO_MT_USER_CB*
- *MSK_IPAR_MT_SPINCOUNT*
- *MSK_IPAR_NUM_THREADS*
- *MSK_IPAR_REMOVE_UNUSED_SOLUTIONS*
- *MSK_IPAR_TIMING_LEVEL*
- *MSK_SPAR_REMOTE_ACCESS_TOKEN*

Presolve

- *MSK_DPAR_PRESOLVE_TOL_ABS_LINDEP*
- *MSK_DPAR_PRESOLVE_TOL_AIJ*
- *MSK_DPAR_PRESOLVE_TOL_REL_LINDEP*
- *MSK_DPAR_PRESOLVE_TOL_S*
- *MSK_DPAR_PRESOLVE_TOL_X*
- *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_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_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_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_NL_TOL_DFEAS*
- *MSK_DPAR_INTPNT_NL_TOL_MU_RED*
- *MSK_DPAR_INTPNT_NL_TOL_NEAR_REL*
- *MSK_DPAR_INTPNT_NL_TOL_PFEAS*
- *MSK_DPAR_INTPNT_NL_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_DISABLE_TERM_TIME*
- *MSK_DPAR_MIO_MAX_TIME*
- *MSK_DPAR_MIO_NEAR_TOL_REL_GAP*
- *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_SOLUTIONS*
- *MSK_IPAR_SIM_MAX_ITERATIONS*

Other

- *MSK_IPAR_COMPRESS_STATFILE*

16.5 Parameters (alphabetical list sorted by type)

- *Double parameters*
- *Integer parameters*
- *String parameters*

16.5.1 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]

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]

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]

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]

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]

Groups *Interior-point method*

MSK_DPAR_DATA_SYM_MAT_TOL

Absolute zero tolerance for elements in in suymmetric matrixes. 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]

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]

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]

Groups *Data check*

MSK_DPAR_DATA_TOL_AIJ

Absolute zero tolerance for elements in A. If any value A_{ij} 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]

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]

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]

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]

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]

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]

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]

Groups *Data check*

MSK_DPAR_DATA_TOL_QIJ

Absolute zero tolerance for elements in Q matrices.

Default 1.0e-16

Accepted [0.0; +inf]

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]

Groups *Data check*

MSK_DPAR_INTPNT_CO_TOL_DFEAS

Dual feasibility tolerance used by the conic interior-point optimizer.

Default 1.0e-8

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

See also *MSK_DPAR_INTPNT_CO_TOL_NEAR_REL*

MSK_DPAR_INTPNT_CO_TOL_INFEAS

Controls when the conic 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]

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

MSK_DPAR_INTPNT_CO_TOL_MU_RED

Relative complementarity gap feasibility tolerance used by the conic interior-point optimizer.

Default 1.0e-8

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

MSK_DPAR_INTPNT_CO_TOL_NEAR_REL

If **MOSEK** cannot compute a solution that has the prescribed accuracy, then it will multiply the termination tolerances with value of this parameter. If the solution then satisfies the termination criteria, then the solution is denoted near optimal, near feasible and so forth.

Default 1000

Accepted [1.0; +inf]

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

MSK_DPAR_INTPNT_CO_TOL_PFEAS

Primal feasibility tolerance used by the conic interior-point optimizer.

Default 1.0e-8

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

See also *MSK_DPAR_INTPNT_CO_TOL_NEAR_REL*

MSK_DPAR_INTPNT_CO_TOL_REL_GAP

Relative gap termination tolerance used by the conic interior-point optimizer.

Default 1.0e-7

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*, *Conic interior-point method*

See also *MSK_DPAR_INTPNT_CO_TOL_NEAR_REL*

MSK_DPAR_INTPNT_NL_MERIT_BAL

Controls if the complementarity and infeasibility is converging to zero at about equal rates.

Default 1.0e-4

Accepted [0.0; 0.99]

Groups *Interior-point method*, *Nonlinear convex method*

MSK_DPAR_INTPNT_NL_TOL_DFEAS

Dual feasibility tolerance used when a nonlinear model is solved.

Default 1.0e-8

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*, *Nonlinear convex method*

MSK_DPAR_INTPNT_NL_TOL_MU_RED

Relative complementarity gap tolerance for the nonlinear solver.

Default 1.0e-12

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*, *Nonlinear convex method*

MSK_DPAR_INTPNT_NL_TOL_NEAR_REL

If the **MOSEK** nonlinear interior-point optimizer cannot compute a solution that has the prescribed accuracy, then it will multiply the termination tolerances with value of this parameter. If the solution then satisfies the termination criteria, then the solution is denoted near optimal, near feasible and so forth.

Default 1000.0

Accepted [1.0; +inf]

Groups *Interior-point method*, *Termination criteria*, *Nonlinear convex method*

MSK_DPAR_INTPNT_NL_TOL_PFEAS
Primal feasibility tolerance used when a nonlinear model is solved.

Default 1.0e-8

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*, *Nonlinear convex method*

MSK_DPAR_INTPNT_NL_TOL_REL_GAP
Relative gap termination tolerance for nonlinear problems.

Default 1.0e-6

Accepted [1.0e-14; +inf]

Groups *Termination criteria*, *Interior-point method*, *Nonlinear convex method*

MSK_DPAR_INTPNT_NL_TOL_REL_STEP
Relative step size to the boundary for general nonlinear optimization problems.

Default 0.995

Accepted [1.0e-4; 0.9999999]

Groups *Interior-point method*, *Nonlinear convex method*

MSK_DPAR_INTPNT_QO_TOL_DFEAS
Dual feasibility tolerance used when the interior-point optimizer is applied to a quadratic optimization problem..

Default 1.0e-8

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*

See also *MSK_DPAR_INTPNT_QO_TOL_NEAR_REL*

MSK_DPAR_INTPNT_QO_TOL_INFEAS
Controls when the conic 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]

Groups *Interior-point method*, *Termination criteria*

MSK_DPAR_INTPNT_QO_TOL_MU_RED
Relative complementarity gap feasibility tolerance used when interior-point optimizer is applied to a quadratic optimization problem.

Default 1.0e-8

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*

MSK_DPAR_INTPNT_QO_TOL_NEAR_REL
If **MOSEK** cannot compute a solution that has the prescribed accuracy, then it will multiply the termination tolerances with value of this parameter. If the solution then satisfies the termination criteria, then the solution is denoted near optimal, near feasible and so forth.

Default 1000

Accepted [1.0; +inf]

Groups *Interior-point method*, *Termination criteria*

MSK_DPAR_INTPNT_QO_TOL_PFEAS
Primal feasibility tolerance used when the interior-point optimizer is applied to a quadratic optimization problem.

Default 1.0e-8

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*

See also *MSK_DPAR_INTPNT_QO_TOL_NEAR_REL*

MSK_DPAR_INTPNT_QO_TOL_REL_GAP
Relative gap termination tolerance used when the interior-point optimizer is applied to a quadratic optimization problem.

Default 1.0e-8

Accepted [0.0; 1.0]

Groups *Interior-point method*, *Termination criteria*

See also *MSK_DPAR_INTPNT_QO_TOL_NEAR_REL*

MSK_DPAR_INTPNT_TOL_DFEAS
Dual feasibility tolerance used for linear optimization problems.

Default 1.0e-8

Accepted [0.0; 1.0]

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]

Groups *Interior-point method*

MSK_DPAR_INTPNT_TOL_INFEAS
Controls when the 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]

Groups *Interior-point method*, *Termination criteria*, *Nonlinear convex method*

MSK_DPAR_INTPNT_TOL_MU_RED
Relative complementarity gap tolerance for linear problems.

Default 1.0e-16

Accepted [0.0; 1.0]

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 is followed very closely. On numerical unstable problems it may be worthwhile to increase this parameter.

Default 1.0e-8

Accepted [0.0; 0.9999]

Groups *Interior-point method*

MSK_DPAR_INTPNT_TOL_PFEAS
Primal feasibility tolerance used for linear optimization problems.

Default 1.0e-8

Accepted [0.0; 1.0]

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]

Groups *Interior-point method*

MSK_DPAR_INTPNT_TOL_REL_GAP

Relative gap termination tolerance for linear problems.

Default 1.0e-8

Accepted [1.0e-14; +inf]

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]

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 stop.

Default 1.0e-6

Accepted [0.0; 1.0]

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]

Groups *Termination criteria*

See also *MSK_DPAR_LOWER_OBJ_CUT_FINITE_TRH*

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]

Groups *Termination criteria*

MSK_DPAR_MIO_DISABLE_TERM_TIME

This parameter specifies the number of seconds n during which the termination criteria governed by

- *MSK_IPAR_MIO_MAX_NUM_RELAXS*
- *MSK_IPAR_MIO_MAX_NUM_BRANCHES*
- *MSK_DPAR_MIO_NEAR_TOL_ABS_GAP*
- *MSK_DPAR_MIO_NEAR_TOL_REL_GAP*

is disabled since the beginning of the optimization.

A negative value is identical to infinity i.e. the termination criteria are never checked.

Default -1.0

Accepted [-inf; +inf]

Groups *Mixed-integer optimization*, *Termination criteria*

See also *MSK_IPAR_MIO_MAX_NUM_RELAXS*, *MSK_IPAR_MIO_MAX_NUM_BRANCHES*, *MSK_DPAR_MIO_NEAR_TOL_ABS_GAP*, *MSK_DPAR_MIO_NEAR_TOL_REL_GAP*

`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]

Groups *Mixed-integer optimization*, *Termination criteria*

`MSK_DPAR_MIO_NEAR_TOL_ABS_GAP`

Relaxed absolute optimality tolerance employed by the mixed-integer optimizer. This termination criteria is delayed. See *MSK_DPAR_MIO_DISABLE_TERM_TIME* for details.

Default 0.0

Accepted [0.0; +inf]

Groups *Mixed-integer optimization*

See also *MSK_DPAR_MIO_DISABLE_TERM_TIME*

`MSK_DPAR_MIO_NEAR_TOL_REL_GAP`

The mixed-integer optimizer is terminated when this tolerance is satisfied. This termination criteria is delayed. See *MSK_DPAR_MIO_DISABLE_TERM_TIME* for details.

Default 1.0e-3

Accepted [0.0; +inf]

Groups *Mixed-integer optimization*, *Termination criteria*

See also *MSK_DPAR_MIO_DISABLE_TERM_TIME*

`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]

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]

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]

    Groups *Mixed-integer optimization*

MSK_DPAR_MIO_TOL_FEAS
: Feasibility tolerance for mixed integer solver.

    Default 1.0e-6

    Accepted [1e-9; 1e-3]

    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]

    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]

    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]

    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]

    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]

    Groups *Presolve*

MSK_DPAR_PRESOLVE_TOL_REL_LINDEP
: Relative tolerance employed by the linear dependency checker.

    Default 1.0e-10

    Accepted [0.0; +inf]

    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]

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]

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]

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]

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]

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]

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]

Groups *Termination criteria*

See also *MSK_DPAR_UPPER_OBJ_CUT_FINITE_TRH*

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]

Groups *Termination criteria*

16.5.2 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*)

Groups *Analysis*

MSK_IPAR_ANA_SOL_PRINT_VIOLATED
Controls whether a list of violated constraints is printed when calling *MSK_analyzesolution*.

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*)

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*)

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*)

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*)

Groups *Simplex optimizer*

MSK_IPAR_BI_CLEAN_OPTIMIZER
Controls which simplex optimizer is used in the clean-up phase.

Default *FREE*

Accepted *FREE*, *INTPNT*, *CONIC*, *PRIMAL_SIMPLEX*, *DUAL_SIMPLEX*, *FREE_SIMPLEX*, *MIXED_INT* (see *MSKoptimizertypee*)

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*)

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*)

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]

Groups *Basis identification*, *Termination criteria*

MSK_IPAR_CACHE_LICENSE

Specifies if the license is kept checked out for the lifetime of the mosek environment (*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*)

Groups *License manager*

MSK_IPAR_CHECK_CONVEXITY

Specify the level of convexity check on quadratic problems.

Default *FULL*

Accepted *NONE*, *SIMPLE*, *FULL* (see *MSKcheckconvexitytypee*)

Groups *Data check*, *Nonlinear convex method*

MSK_IPAR_COMPRESS_STATFILE

Control compression of stat files.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

MSK_IPAR_INFEAS_GENERIC_NAMES
Controls whether generic names are used when an infeasible subproblem is created.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

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*)

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*)

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]

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*)

Groups *Interior-point method*, *Basis identification*

See also *MSK_IPAR_BI_IGNORE_MAX_ITER*, *MSK_IPAR_BI_IGNORE_NUM_ERROR*, *MSK_IPAR_BI_MAX_ITERATIONS*, *MSK_IPAR_BI_CLEAN_OPTIMIZER*

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.

Groups *Interior-point method*

MSK_IPAR_INTPNT_HOTSTART
Currently not in use.

Default *NONE*

Accepted *NONE*, *PRIMAL*, *DUAL*, *PRIMAL_DUAL* (see *MSKintpnthotstarte*)

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]

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]

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]

Groups *Interior-point method*

MSK_IPAR_INTPNT_MULTI_THREAD
Controls whether the interior-point optimizers are allowed to employ multiple threads if more threads is available.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Groups *Overall system*

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]

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*)

Groups *Interior-point method*

MSK_IPAR_INTPNT_REGULARIZATION_USE
Controls whether regularization is allowed.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

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*, *MODERATE*, *AGGRESSIVE* (see *MSKscalingtypee*)

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*)

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*)

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*)

Groups *License manager*

MSK_IPAR_LICENSE_PAUSE_TIME

If *MSK_IPAR_LICENSE_WAIT*= *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]

Groups *License manager*

MSK_IPAR_LICENSE_SUPPRESS_EXPIRE_WRNS

Controls whether license features expire warnings are suppressed.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Groups *License manager*, *Output information*

MSK_IPAR_LICENSE_TRH_EXPIRY_WRN

If a license feature expires in a numbers days less than the value of this parameter then a warning will be issued.

Default 7

Accepted [0; +inf]

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*)

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]

Groups *Output information*, *Logging*

See also *MSK_IPAR_LOG_CUT_SECOND_OPT*

MSK_IPAR_LOG_ANA_PRO

Controls amount of output from the problem analyzer.

Default 1

Accepted [0; +inf]

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]

Groups *Basis identification*, *Output information*, *Logging*

MSK_IPAR_LOG_BI_FREQ

Controls how frequent the optimizer outputs information about the basis identification and how frequent the user-defined callback function is called.

Default 2500

Accepted [0; +inf]

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]

Groups *Data check*, *Nonlinear convex method*

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]

Groups *Output information*, *Logging*

See also *MSK_IPAR_LOG*, *MSK_IPAR_LOG_INTPNT*, *MSK_IPAR_LOG_MIO*, *MSK_IPAR_LOG_SIM*

`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]

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]

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]

Groups *Data input/output*, *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]

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]

Groups *Interior-point method*, *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]

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]

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]

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]

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]

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]

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]

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]

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]

Groups *Simplex optimizer*, *Output information*, *Logging*

`MSK_IPAR_LOG_SIM_MINOR`

Currently not in use.

Default 1

Accepted [0; +inf]

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]

Groups *Output information*, *Overall system*, *Logging*

MSK_IPAR_MAX_NUM_WARNINGS

Each warning is shown a limit number times controlled by this parameter. A negative value is identical to infinite number of times.

Default 10

Accepted [-inf; +inf]

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*)

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*)

Groups *Mixed-integer optimization*

MSK_IPAR_MIO_CUT_CLIQUE

Controls whether clique cuts should be generated.

Default *ON*

Accepted

- *ON*: Turns generation of this cut class on.
- *OFF*: Turns generation of this cut class off.

Groups *Mixed-integer optimization*

MSK_IPAR_MIO_CUT_CMIR

Controls whether mixed integer rounding cuts should be generated.

Default *ON*

Accepted

- *ON*: Turns generation of this cut class on.
- *OFF*: Turns generation of this cut class off.

Groups *Mixed-integer optimization*

MSK_IPAR_MIO_CUT_GMI

Controls whether GMI cuts should be generated.

Default *ON*

Accepted

- *ON*: Turns generation of this cut class on.
- *OFF*: Turns generation of this cut class off.

Groups *Mixed-integer optimization*

MSK_IPAR_MIO_CUT_IMPLIED_BOUND
Controls whether implied bound cuts should be generated.

Default *OFF*

Accepted

- *ON*: Turns generation of this cut class on.
- *OFF*: Turns generation of this cut class off.

Groups *Mixed-integer optimization*

MSK_IPAR_MIO_CUT_KNAPSACK_COVER
Controls whether knapsack cover cuts should be generated.

Default *OFF*

Accepted

- *ON*: Turns generation of this cut class on.
- *OFF*: Turns generation of this cut class 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]

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]

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]

Groups *Mixed-integer optimization*, *Termination criteria*

See also *MSK_DPAR_MIO_DISABLE_TERM_TIME*

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]

Groups *Mixed-integer optimization*

See also *MSK_DPAR_MIO_DISABLE_TERM_TIME*

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]

Groups *Mixed-integer optimization*, *Termination criteria*

See also *MSK_DPAR_MIO_DISABLE_TERM_TIME*

MSK_IPAR_MIO_MODE
Controls whether the optimizer includes the integer restrictions when solving a (mixed) integer optimization problem.

Default *SATISFIED*

Accepted *IGNORED*, *SATISFIED* (see *MSKmiomodee*)

Groups *Overall solver*

MSK_IPAR_MIO_MT_USER_CB
If true user callbacks are called from each thread used by mixed-integer optimizer. Otherwise it is only called from a single thread.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Groups *Overall system*

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*)

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*, *WORST*, *HYBRID*, *PSEUDO* (see *MSKmionodeseltypee*)

Groups *Mixed-integer optimization*

MSK_IPAR_MIO_PERSPECTIVE_REFORMULATE
Enables or disables perspective reformulation in presolve.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Groups *Mixed-integer optimization*

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]

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]

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*)

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]

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]

Groups *Mixed-integer optimization*

MSK_IPAR_MT_SPINCOUNT
Set the number of iterations to spin before sleeping.

Default 0

Accepted [0; 1000000000]

Groups *Overall system*

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]

Groups *Overall system*

MSK_IPAR_OPF_MAX_TERMS_PER_LINE
The maximum number of terms (linear and quadratic) per line when an OPF file is written.

Default 5

Accepted [0; +inf]

Groups *Data input/output*

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*)

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*)

Groups *Data input/output*

MSK_IPAR_OPF_WRITE_PARAMETERS
Write a parameter section in an OPF file.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

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*)

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*)

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*)

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*)

Groups *Data input/output*

MSK_IPAR_OPF_WRITE_SOLUTIONS
Enable inclusion of solutions in the OPF files.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

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*)

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*)

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*)

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]

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]

Groups *Presolve*

`MSK_IPAR_PRESOLVE_LEVEL`

Currently not used.

Default -1

Accepted [-inf; +inf]

Groups *Overall solver*, *Presolve*

`MSK_IPAR_PRESOLVE_LINDEP_ABS_WORK_TRH`

The linear dependency check is potentially computationally expensive.

Default 100

Accepted [-inf; +inf]

Groups *Presolve*

`MSK_IPAR_PRESOLVE_LINDEP_REL_WORK_TRH`

The linear dependency check is potentially computationally expensive.

Default 100

Accepted [-inf; +inf]

Groups *Presolve*

`MSK_IPAR_PRESOLVE_LINDEP_USE`

Controls whether the linear constraints are checked for linear dependencies.

Default *ON*

Accepted

- *ON*: Turns the linear dependency check on.
- *OFF*: Turns the linear dependency check off.

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]

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*)

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*)

Groups *Overall solver*

MSK_IPAR_READ_DATA_COMPRESSED
If this option is turned on,it is assumed that the data file is compressed.

Default *FREE*

Accepted *NONE*, *FREE*, *GZIP* (see *MSKcompresstypee*)

Groups *Data input/output*

MSK_IPAR_READ_DATA_FORMAT
Format of the data file to be read.

Default *EXTENSION*

Accepted *EXTENSION*, *MPS*, *LP*, *OP*, *XML*, *FREE_MPS*, *TASK*, *CB*, *JSON_TASK* (see *MSKdataformate*)

Groups *Data input/output*

MSK_IPAR_READ_DEBUG
Turns on additional debugging information when reading files.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

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.

Groups *Data input/output*

MSK_IPAR_READ_LP_DROP_NEW_VARS_IN_BOU
If this option is turned on, **MOSEK** will drop variables that are defined for the first time in the bounds section.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

Groups *Data input/output*

MSK_IPAR_READ_LP_QUOTED_NAMES
If a name is in quotes when reading an LP file, the quotes will be removed.

Default *ON*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

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*)

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]

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*)

Groups *Data input/output*

MSK_IPAR_REMOVE_UNUSED_SOLUTIONS

Removes unsued solutions before the optimization is performed.

Default *OFF*

Accepted *ON*, *OFF* (see *MSKonoffkeye*)

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*)

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*)

Groups *Overall solver*, *Simplex optimizer*

MSK_IPAR_SENSITIVITY_TYPE

Controls which type of sensitivity analysis is to be performed.

Default *BASIS*

Accepted *BASIS*, *OPTIMAL_PARTITION* (see *MSKsensitivitytypee*)

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*)

Groups *Simplex optimizer*

MSK_IPAR_SIM_DEGEN

Controls how aggressively degeneration is handled.

Default *FREE*

Accepted *NONE*, *FREE*, *AGGRESSIVE*, *MODERATE*, *MINIMUM* (see *MSKsimdegene*)

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]

Groups *Dual simplex*

MSK_IPAR_SIM_DUAL_PHASEONE_METHOD

An experimental feature.

Default 0

Accepted [0; 10]

Groups *Simplex optimizer*

MSK_IPAR_SIM_DUAL_RESTRICT_SELECTION

The dual simplex optimizer can use a so-called restricted selection/pricing strategy to chooses 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]

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*)

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*)

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*)

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.

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]

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]

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*)

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]

Groups *Primal simplex*

MSK_IPAR_SIM_PRIMAL_PHASEONE_METHOD

An experimental feature.

Default 0

Accepted [0; 10]

Groups *Simplex optimizer*

MSK_IPAR_SIM_PRIMAL_RESTRICT_SELECTION

The primal simplex optimizer can use a so-called restricted selection/pricing strategy to chooses 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]

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*)

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]

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*)

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*)

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*, *MODERATE*, *AGGRESSIVE* (see *MSKscalingtypee*)

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*)

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*)

Groups *Simplex optimizer*

MSK_IPAR_SIM_STABILITY_PRIORITY

Controls how high priority the numerical stability should be given.

Default 50

Accepted [0; 100]

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*)

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*)

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*)

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]

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]

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*)

Groups *Progress callback*, *Overall solver*

`MSK_IPAR_TIMING_LEVEL`

Controls the amount of timing performed inside **MOSEK**.

Default 1

Accepted [0; +inf]

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*)

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*)

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*)

Groups *Data input/output*, *Solution input/output*

MSK_IPAR_WRITE_DATA_COMPRESSED
Controls whether the data file is compressed while it is written. 0 means no compression while higher values mean more compression.

Default 0

Accepted [0; +inf]

Groups *Data input/output*

MSK_IPAR_WRITE_DATA_FORMAT
Controls the data format when a task is written using *MSK_writedata*.

Default *EXTENSION*

Accepted *EXTENSION*, *MPS*, *LP*, *OP*, *XML*, *FREE_MPS*, *TASK*, *CB*, *JSON_TASK* (see *MSKdataformate*)

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*)

Groups *Data input/output*

MSK_IPAR_WRITE_FREE_CON
Controls whether the free constraints are written to the data file.

Default *ON*

Accepted

- *ON*: The free constraints are written.
- *OFF*: The free constraints are discarded.

Groups *Data input/output*

MSK_IPAR_WRITE_GENERIC_NAMES
Controls whether the generic names or user-defined names are used in the data file.

Default *OFF*

Accepted

- *ON*: Generic names are used.
- *OFF*: Generic names are not used.

Groups *Data input/output*

MSK_IPAR_WRITE_GENERIC_NAMES_IO

Index origin used in generic names.

Default 1

Accepted [0; +inf]

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.

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*)

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*)

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*)

Groups *Data input/output*, *Solution 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*)

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]

Groups *Data input/output*

MSK_IPAR_WRITE_LP_QUOTED_NAMES
: If this option is turned on, then **MOSEK** will quote invalid LP names when writing an LP file.

  Default *ON*

  Accepted *ON*, *OFF* (see *MSKonoffkeye*)

  Groups *Data input/output*

MSK_IPAR_WRITE_LP_STRICT_FORMAT
: Controls whether LP output files satisfy the LP format strictly.

  Default *OFF*

  Accepted *ON*, *OFF* (see *MSKonoffkeye*)

  Groups *Data input/output*

MSK_IPAR_WRITE_LP_TERMS_PER_LINE
: Maximum number of terms on a single line in an LP file written by **MOSEK**. 0 means unlimited.

  Default 10

  Accepted [0; +inf]

  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*)

  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*: Marker records are written.
  - *OFF*: Marker records are not written.

  Groups *Data input/output*

MSK_IPAR_WRITE_PRECISION
: Controls the precision with which `double` numbers are printed in the MPS data file. In general it is not worthwhile to use a value higher than 15.

  Default 15

  Accepted [0; +inf]

  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*)

  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*)

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*)

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*)

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*)

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*)

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*)

Groups *Data input/output*

16.5.3 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.

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.

Groups *Data input/output*

MSK_SPAR_DEBUG_FILE_NAME

MOSEK debug file.

Accepted Any valid file name.

Groups *Data input/output*

MSK_SPAR_INT_SOL_FILE_NAME

Name of the `int` solution file.

Accepted Any valid file name.

Groups *Data input/output*, *Solution input/output*

MSK_SPAR_ITR_SOL_FILE_NAME

Name of the `itr` solution file.

Accepted Any valid file name.

Groups *Data input/output*, *Solution input/output*

MSK_SPAR_MIO_DEBUG_STRING

For internal debugging purposes.

Accepted Any valid string.

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.

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.

Groups *Data input/output*

MSK_SPAR_PARAM_WRITE_FILE_NAME

The parameter database is written to this file.

Accepted Any valid file name.

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.

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.

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.

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.

Groups *Data input/output*

MSK_SPAR_REMOTE_ACCESS_TOKEN
An access token used to submit tasks to a remote **MOSEK** server. An access token is a random 32-byte string encoded in base64, i.e. it is a 44 character ASCII string.

Accepted Any valid string.

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.

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.

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.

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.

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.

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.

Groups *Data input/output*, *Solution input/output*

MSK_SPAR_STAT_FILE_NAME
Statistics file name.

Accepted Any valid file name.

Groups *Data input/output*

`MSK_SPAR_STAT_KEY`
Key used when writing the summary file.

Accepted Any valid string.

Groups *Data input/output*

`MSK_SPAR_STAT_NAME`
Name used when writing the statistics file.

Accepted Any valid XML string.

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.

Groups *Data input/output*

16.6 Response codes

- *Termination*
- *Warnings*
- *Errors*

`MSKrescodee`
The enumeration type containing all response codes.

16.6.1 Termination

`MSK_RES_OK (0)`
No error occurred.

`MSK_RES_TRM_MAX_ITERATIONS (10000)`
The optimizer terminated at the maximum number of iterations.

`MSK_RES_TRM_MAX_TIME (10001)`
The optimizer terminated at the maximum amount of time.

`MSK_RES_TRM_OBJECTIVE_RANGE (10002)`
The optimizer terminated with an objective value outside the objective range.

`MSK_RES_TRM_MIO_NEAR_REL_GAP (10003)`
The mixed-integer optimizer terminated as the delayed near optimal relative gap tolerance was satisfied.

`MSK_RES_TRM_MIO_NEAR_ABS_GAP (10004)`
The mixed-integer optimizer terminated as the delayed near optimal absolute gap tolerance was satisfied.

`MSK_RES_TRM_MIO_NUM_RELAXS (10008)`
The mixed-integer optimizer terminated as the maximum number of relaxations was reached.

`MSK_RES_TRM_MIO_NUM_BRANCHES (10009)`
The mixed-integer optimizer terminated as the maximum number of branches was reached.

MSK_RES_TRM_NUM_MAX_NUM_INT_SOLUTIONS (10015)
: The mixed-integer optimizer terminated as the maximum number of feasible solutions was reached.

MSK_RES_TRM_STALL (10006)
: The optimizer is terminated due to slow progress.

Stalling means that numerical problems prevent the optimizer from making reasonable progress and that it make 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 (near) feasible or near 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 then solution. If the solution near 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 and c) a non-convex problems. Case c) is only relevant for general non-linear problems. It is not possible in general for **MOSEK** to check if a specific problems is convex since such a check would be NP hard in itself. This implies that care should be taken when solving problems involving general user defined functions.

MSK_RES_TRM_USER_CALLBACK (10007)
: The optimizer terminated due to the return of the user-defined callback function.

MSK_RES_TRM_MAX_NUM_SETBACKS (10020)
: 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 (10025)
: The optimizer terminated due to numerical problems.

MSK_RES_TRM_INTERNAL (10030)
: The optimizer terminated due to some internal reason. Please contact **MOSEK** support.

MSK_RES_TRM_INTERNAL_STOP (10031)
: The optimizer terminated for internal reasons. Please contact **MOSEK** support.

16.6.2 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_NO_NONLINEAR_FUNCTION_WRITE (450)
: The problem contains a general nonlinear function in either the objective or the constraints. Such a nonlinear function cannot be written to a disk file. Note that quadratic terms when inputted explicitly can be written to disk.

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_LICENSE_FEATURE_EXPIRE (505)
: 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_WRITE_CHANGED_NAMES (803)
: Some names were changed because they were invalid for the output file format.

MSK_RES_WRN_WRITE_DISCARDED_CFIX (804)
: The fixed objective term could not be converted to a variable and was discarded in the output file.

MSK_RES_WRN_CONSTRUCT_SOLUTION_INFEAS (805)
: After fixing the integer variables at the suggested values then the problem is infeasible.

MSK_RES_WRN_CONSTRUCT_INVALID_SOL_ITG (807)
: The initial value for one or more of the integer variables is not feasible.

MSK_RES_WRN_CONSTRUCT_NO_SOL_ITG (810)
: The construct solution requires an integer solution.

MSK_RES_WRN_DUPLICATE_CONSTRAINT_NAMES (850)
: Two constraint names are identical.

MSK_RES_WRN_DUPLICATE_VARIABLE_NAMES (851)
: Two variable names are identical.

MSK_RES_WRN_DUPLICATE_BARVARIABLE_NAMES (852)
: Two barvariable names are identical.

MSK_RES_WRN_DUPLICATE_CONE_NAMES (853)
: Two cone names are identical.

MSK_RES_WRN_ANA_LARGE_BOUNDS (900)
This warning is issued by the problem analyzer, if one or more constraint or variable bounds are very large. One should consider omitting these bounds entirely by setting them to +inf or -inf.

MSK_RES_WRN_ANA_C_ZERO (901)
This warning is issued by the problem analyzer, if the coefficients in the linear part of the objective are all zero.

MSK_RES_WRN_ANA_EMPTY_COLS (902)
This warning is issued by the problem analyzer, if columns, in which all coefficients are zero, are found.

MSK_RES_WRN_ANA_CLOSE_BOUNDS (903)
This warning is issued by problem analyzer, if ranged constraints or variables with very close upper and lower bounds are detected. One should consider treating such constraints as equalities and such variables as constants.

MSK_RES_WRN_ANA_ALMOST_INT_BOUNDS (904)
This warning is issued by the problem analyzer if a constraint is bound nearly integral.

MSK_RES_WRN_QUAD_CONES_WITH_ROOT_FIXED_AT_ZERO (930)
For at least one quadratic cone the root is fixed at (nearly) zero. This may cause problems such as a very large dual solution. Therefore, it is recommended to remove such cones before optimizing the problems, or to fix all the variables in the cone to 0.

MSK_RES_WRN_RQUAD_CONES_WITH_ROOT_FIXED_AT_ZERO (931)
For at least one rotated quadratic cone at least one of the root variables are fixed at (nearly) zero. This may cause problems such as a very large dual solution. Therefore, it is recommended to remove such cones before optimizing the problems, or to fix all the variables in the cone to 0.

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.

16.6.3 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_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 license file or a token server. See the **MOSEK** installation 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_OPEN_DL (1030)`
A dynamic link library could not be opened.

`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_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_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_INV_CON_KEY (1107)`
: An invalid constraint key occurred in an MPS file.

`MSK_RES_ERR_MPS_INV_BOUND_KEY (1108)`
: An invalid bound key occurred in an MPS file.

`MSK_RES_ERR_MPS_INV_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 CSECTIONs are given the same name.

MSK_RES_ERR_MPS_CONE_TYPE (1117)
: Invalid cone type specified in a CSECTION.

MSK_RES_ERR_MPS_CONE_OVERLAP (1118)
: A variable is specified to be a member of several cones.

MSK_RES_ERR_MPS_CONE_REPEAT (1119)
: A variable is repeated within the CSECTION.

MSK_RES_ERR_MPS_NON_SYMMETRIC_Q (1120)
: A non symmetric matrice 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_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_FORMAT (1160)
: Syntax error in an LP file.

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_OPF_FORMAT (1168)
Syntax error in an OPF file

MSK_RES_ERR_OPF_NEW_VARIABLE (1169)
Introducing new variables is now allowed. When a `[variables]` section is present, it is not allowed to introduce new variables later in the problem.

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_OPF_PREMATURE_EOF (1172)
Premature end of file in an OPF 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_ARGUMENT_LENNEQ (1197)
Incorrect length of arguments.

MSK_RES_ERR_ARGUMENT_TYPE (1198)
Incorrect argument type.

MSK_RES_ERR_NR_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_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_PARAM_NAME (1205)
The parameter name is not correct.

MSK_RES_ERR_PARAM_NAME_DOU (1206)
The parameter name is not correct for a double parameter.

MSK_RES_ERR_PARAM_NAME_INT (1207)
The parameter name is not correct for an integer parameter.

MSK_RES_ERR_PARAM_NAME_STR (1208)
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 small.

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_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_FIRST (1261)
Invalid first.

MSK_RES_ERR_LAST (1262)
Invalid index last. A given index was out of expected range.

MSK_RES_ERR_NEGATIVE_SURPLUS (1263)
Negative surplus.

MSK_RES_ERR_NEGATIVE_APPEND (1264)
Cannot append a negative number.

MSK_RES_ERR_UNDEF_SOLUTION (1265)
MOSEK has the following solution types:

- an interior-point solution,
- an 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_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)
    An 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 too few 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_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 (1425)
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_NONLINEAR_FUNCTIONS_NOT_ALLOWED (1428)
An operation that is invalid for problems with nonlinear functions defined has been attempted.

MSK_RES_ERR_USER_FUNC_RET (1430)
An user function reported an error.

MSK_RES_ERR_USER_FUNC_RET_DATA (1431)
An user function returned invalid data.

`MSK_RES_ERR_USER_NLO_FUNC (1432)`
The user-defined nonlinear function reported an error.

`MSK_RES_ERR_USER_NLO_EVAL (1433)`
The user-defined nonlinear function reported an error.

`MSK_RES_ERR_USER_NLO_EVAL_HESSUBI (1440)`
The user-defined nonlinear function reported an invalid subscript in the Hessian.

`MSK_RES_ERR_USER_NLO_EVAL_HESSUBJ (1441)`
The user-defined nonlinear function reported an invalid subscript in the Hessian.

`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_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_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_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. This means that the simplex or the conic optimizer is chosen to optimize a nonlinear 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_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. This is the case e.g. for general nonlinear optimization problems.

`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_CANNOT_CLONE_NL (2505)
A task with a nonlinear function callback cannot be cloned.

MSK_RES_ERR_CANNOT_HANDLE_NL (2506)
A function cannot handle a task with nonlinear function callbacks.

MSK_RES_ERR_INVALID_ACCMODE (2520)
An invalid access mode is specified.

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)
: An 32 bit integer could not cast to a 64 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 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 value floating 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_CONES (4005)
The file format does not support a problem with conic constraints.

MSK_RES_ERR_INVALID_FILE_FORMAT_FOR_GENERAL_NL (4010)
The file format does not support a problem with general nonlinear terms.

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_NON_UNIQUE_ARRAY (5000)
An array does not contain unique elements.

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 (7109)
Duplicate VAR keyword.

MSK_RES_ERR_CBF_DUPLICATE_INT (7110)
Duplicate INT keyword.

MSK_RES_ERR_CBF_INVALID_VAR_TYPE (7111)
Invalid variable type.

MSK_RES_ERR_CBF_INVALID_CON_TYPE (7112)
Invalid constraint type.

MSK_RES_ERR_CBF_INVALID_DOMAIN_DIMENSION (7113)
Invalid domain dimension.

MSK_RES_ERR_CBF_DUPLICATE_OBJACOORD (7114)
Duplicate index in OBJCOORD.

MSK_RES_ERR_CBF_DUPLICATE_BCOORD (7115)
Duplicate index in BCOORD.

MSK_RES_ERR_CBF_DUPLICATE_ACOORD (7116)
Duplicate index in ACOORD.

MSK_RES_ERR_CBF_TOO_FEW_VARIABLES (7117)
Too few variables defined.

MSK_RES_ERR_CBF_TOO_FEW_CONSTRAINTS (7118)
Too few constraints defined.

MSK_RES_ERR_CBF_TOO_FEW_INTS (7119)
Too few ints are specified.

MSK_RES_ERR_CBF_TOO_MANY_INTS (7120)
Too many ints are specified.

MSK_RES_ERR_CBF_INVALID_INT_INDEX (7121)
Invalid INT index.

MSK_RES_ERR_CBF_UNSUPPORTED (7122)
Unsupported feature is present.

MSK_RES_ERR_CBF_DUPLICATE_PSDVAR (7123)
Duplicate PSDVAR keyword.

MSK_RES_ERR_CBF_INVALID_PSDVAR_DIMENSION (7124)
Invalid PSDVAR dimmension.

MSK_RES_ERR_CBF_TOO_FEW_PSDVAR (7125)
Too few variables defined.

MSK_RES_ERR_MIO_INVALID_ROOT_OPTIMIZER (7130)
An invalid root optimizer was selected for the problem type.

MSK_RES_ERR_MIO_INVALID_NODE_OPTIMIZER (7131)
An invalid node optimizer was selected for the problem type.

MSK_RES_ERR_TOCONIC_CONSTR_Q_NOT_PSD (7150)
The matrix defining the quadratric part of constraint is not positive semidefinite.

MSK_RES_ERR_TOCONIC_CONSTRAINT_FX (7151)
The quadratic constraint is an equality, thus not convex.

MSK_RES_ERR_TOCONIC_CONSTRAINT_RA (7152)
The quadratic constraint has finite lower and upper bound, and therefore it is not convex.

MSK_RES_ERR_TOCONIC_CONSTR_NOT_CONIC (7153)
The constraint is not conic representable.

MSK_RES_ERR_TOCONIC_OBJECTIVE_NOT_PSD (7155)
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

16.7 Enumerations

MSKlanguagee
Language selection constants

MSK_LANG_ENG (0)
English language selection

MSK_LANG_DAN (1)
Danish language selection

MSKaccmodee
Constraint or variable access modes. All functions using this enum are deprecated. Use separate functions for rows/columns instead.

MSK_ACC_VAR (0)
Access data by columns (variable oriented)

MSK_ACC_CON (1)
Access data by rows (constraint oriented)

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.

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_FULL_CONVEXITY_CHECK (7)
Begin full convexity check.

MSK_CALLBACK_BEGIN_INFEAS_ANA (8)
The callback function is called when the infeasibility analyzer is started.

MSK_CALLBACK_BEGIN_INTPNT (9)
The callback function is called when the interior-point optimizer is started.

MSK_CALLBACK_BEGIN_LICENSE_WAIT (10)
Begin waiting for license.

MSK_CALLBACK_BEGIN_MIO (11)
The callback function is called when the mixed-integer optimizer is started.

MSK_CALLBACK_BEGIN_OPTIMIZER (12)
The callback function is called when the optimizer is started.

MSK_CALLBACK_BEGIN_PRESOLVE (13)
The callback function is called when the presolve is started.

MSK_CALLBACK_BEGIN_PRIMAL_BI (14)
The callback function is called from within the basis identification procedure when the primal phase is started.

MSK_CALLBACK_BEGIN_PRIMAL_REPAIR (15)
Begin primal feasibility repair.

MSK_CALLBACK_BEGIN_PRIMAL_SENSITIVITY (16)
Primal sensitivity analysis is started.

MSK_CALLBACK_BEGIN_PRIMAL_SETUP_BI (17)
The callback function is called when the primal BI setup is started.

MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX (18)
The callback function is called when the primal simplex optimizer is started.

MSK_CALLBACK_BEGIN_PRIMAL_SIMPLEX_BI (19)
: 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 (20)
: Begin QCQO reformulation.

MSK_CALLBACK_BEGIN_READ (21)
: **MOSEK** has started reading a problem file.

MSK_CALLBACK_BEGIN_ROOT_CUTGEN (22)
: The callback function is called when root cut generation is started.

MSK_CALLBACK_BEGIN_SIMPLEX (23)
: The callback function is called when the simplex optimizer is started.

MSK_CALLBACK_BEGIN_SIMPLEX_BI (24)
: The callback function is called from within the basis identification procedure when the simplex clean-up phase 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_FULL_CONVEXITY_CHECK (36)
: End full convexity check.

MSK_CALLBACK_END_INFEAS_ANA (37)
: The callback function is called when the infeasibility analyzer is terminated.

MSK_CALLBACK_END_INTPNT (38)
: The callback function is called when the interior-point optimizer is terminated.

MSK_CALLBACK_END_LICENSE_WAIT (39)
: End waiting for license.

`MSK_CALLBACK_END_MIO (40)`
The callback function is called when the mixed-integer optimizer is terminated.

`MSK_CALLBACK_END_OPTIMIZER (41)`
The callback function is called when the optimizer is terminated.

`MSK_CALLBACK_END_PRESOLVE (42)`
The callback function is called when the presolve is completed.

`MSK_CALLBACK_END_PRIMAL_BI (43)`
The callback function is called from within the basis identification procedure when the primal phase is terminated.

`MSK_CALLBACK_END_PRIMAL_REPAIR (44)`
End primal feasibility repair.

`MSK_CALLBACK_END_PRIMAL_SENSITIVITY (45)`
Primal sensitivity analysis is terminated.

`MSK_CALLBACK_END_PRIMAL_SETUP_BI (46)`
The callback function is called when the primal BI setup is terminated.

`MSK_CALLBACK_END_PRIMAL_SIMPLEX (47)`
The callback function is called when the primal simplex optimizer is terminated.

`MSK_CALLBACK_END_PRIMAL_SIMPLEX_BI (48)`
The callback function is called from within the basis identification procedure when the primal clean-up phase is terminated.

`MSK_CALLBACK_END_QCQO_REFORMULATE (49)`
End QCQO reformulation.

`MSK_CALLBACK_END_READ (50)`
MOSEK has finished reading a problem file.

`MSK_CALLBACK_END_ROOT_CUTGEN (51)`
The callback function is called when root cut generation is is terminated.

`MSK_CALLBACK_END_SIMPLEX (52)`
The callback function is called when the simplex optimizer is terminated.

`MSK_CALLBACK_END_SIMPLEX_BI (53)`
The callback function is called from within the basis identification procedure when the simplex clean-up phase 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_FULL_CONVEXITY_CHECK (61)`
: The callback function is called at an intermediate stage of the full convexity check.

`MSK_CALLBACK_IM_INTPNT (62)`
: 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 (63)`
: **MOSEK** is waiting for a license.

`MSK_CALLBACK_IM_LU (64)`
: The callback function is called from within the LU factorization procedure at an intermediate point.

`MSK_CALLBACK_IM_MIO (65)`
: The callback function is called at an intermediate point in the mixed-integer optimizer.

`MSK_CALLBACK_IM_MIO_DUAL_SIMPLEX (66)`
: 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 (67)`
: 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 (68)`
: The callback function is called at an intermediate point in the mixed-integer optimizer while running the primal simplex optimizer.

`MSK_CALLBACK_IM_ORDER (69)`
: The callback function is called from within the matrix ordering procedure at an intermediate point.

`MSK_CALLBACK_IM_PRESOLVE (70)`
: The callback function is called from within the presolve procedure at an intermediate stage.

`MSK_CALLBACK_IM_PRIMAL_BI (71)`
: The callback function is called from within the basis identification procedure at an intermediate point in the primal phase.

`MSK_CALLBACK_IM_PRIMAL_SENSIVITY (72)`
: The callback function is called at an intermediate stage of the primal sensitivity analysis.

`MSK_CALLBACK_IM_PRIMAL_SIMPLEX (73)`
: The callback function is called at an intermediate point in the primal simplex optimizer.

`MSK_CALLBACK_IM_QO_REFORMULATE (74)`
: The callback function is called at an intermediate stage of the conic quadratic reformulation.

`MSK_CALLBACK_IM_READ (75)`
: Intermediate stage in reading.

`MSK_CALLBACK_IM_ROOT_CUTGEN (76)`
: The callback is called from within root cut generation at an intermediate stage.

`MSK_CALLBACK_IM_SIMPLEX (77)`
: The callback function is called from within the simplex optimizer at an intermediate point.

`MSK_CALLBACK_IM_SIMPLEX_BI (78)`
: 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 (79)`
: The callback function is called from within the interior-point optimizer after the information database has been updated.

`MSK_CALLBACK_NEW_INT_MIO (80)`
The callback function is called after a new integer solution has been located by the mixed-integer optimizer.

`MSK_CALLBACK_PRIMAL_SIMPLEX (81)`
The callback function is called from within the primal simplex optimizer.

`MSK_CALLBACK_READ_OPF (82)`
The callback function is called from the OPF reader.

`MSK_CALLBACK_READ_OPF_SECTION (83)`
A chunk of Q non-zeros has been read from a problem file.

`MSK_CALLBACK_SOLVING_REMOTE (84)`
The callback function is called while the task is being solved on a remote server.

`MSK_CALLBACK_UPDATE_DUAL_BI (85)`
The callback function is called from within the basis identification procedure at an intermediate point in the dual phase.

`MSK_CALLBACK_UPDATE_DUAL_SIMPLEX (86)`
The callback function is called in the dual simplex optimizer.

`MSK_CALLBACK_UPDATE_DUAL_SIMPLEX_BI (87)`
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 (88)`
The callback function is called from within the presolve procedure.

`MSK_CALLBACK_UPDATE_PRIMAL_BI (89)`
The callback function is called from within the basis identification procedure at an intermediate point in the primal phase.

`MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX (90)`
The callback function is called in the primal simplex optimizer.

`MSK_CALLBACK_UPDATE_PRIMAL_SIMPLEX_BI (91)`
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_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.

`MSKconetypee`
: Cone types

  `MSK_CT_QUAD (0)`
  : The cone is a quadratic cone.

  `MSK_CT_RQUAD (1)`
  : The cone is a rotated quadratic 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_XML (4)`
  : The data file is an XML formatted file.

  `MSK_DATA_FORMAT_FREE_MPS (5)`
  : The data a free MPS formatted file.

  `MSK_DATA_FORMAT_TASK (6)`
  : Generic task dump file.

  `MSK_DATA_FORMAT_CB (7)`
  : Conic benchmark format,

  `MSK_DATA_FORMAT_JSON_TASK (8)`
  : JSON based task format.

`MSKdinfiteme`
: Double information items

  `MSK_DINF_BI_CLEAN_DUAL_TIME (0)`
  : Time spent within the dual clean-up optimizer of the basis identification procedure since its invocation.

MSK_DINF_BI_CLEAN_PRIMAL_TIME (1)
Time spent within the primal clean-up optimizer of the basis identification procedure since its invocation.

MSK_DINF_BI_CLEAN_TIME (2)
Time spent within the clean-up phase of the basis identification procedure since its invocation.

MSK_DINF_BI_DUAL_TIME (3)
Time spent within the dual phase basis identification procedure since its invocation.

MSK_DINF_BI_PRIMAL_TIME (4)
Time spent within the primal phase of the basis identification procedure since its invocation.

MSK_DINF_BI_TIME (5)
Time spent within the basis identification procedure since its invocation.

MSK_DINF_INTPNT_DUAL_FEAS (6)
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 (7)
Dual objective value reported by the interior-point optimizer.

MSK_DINF_INTPNT_FACTOR_NUM_FLOPS (8)
An estimate of the number of flops used in the factorization.

MSK_DINF_INTPNT_OPT_STATUS (9)
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 (10)
Order time (in seconds).

MSK_DINF_INTPNT_PRIMAL_FEAS (11)
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 (12)
Primal objective value reported by the interior-point optimizer.

MSK_DINF_INTPNT_TIME (13)
Time spent within the interior-point optimizer since its invocation.

MSK_DINF_MIO_CLIQUE_SEPARATION_TIME (14)
Seperation time for clique cuts.

MSK_DINF_MIO_CMIR_SEPARATION_TIME (15)
Seperation time for CMIR cuts.

MSK_DINF_MIO_CONSTRUCT_SOLUTION_OBJ (16)
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 (17)
Value of the dual bound after presolve but before cut generation.

MSK_DINF_MIO_GMI_SEPARATION_TIME (18)
Seperation time for GMI cuts.

MSK_DINF_MIO_HEURISTIC_TIME (19)
Total time spent in the optimizer.

MSK_DINF_MIO_IMPLIED_BOUND_TIME (20)
Seperation time for implied bound cuts.

MSK_DINF_MIO_KNAPSACK_COVER_SEPARATION_TIME (21)
Seperation time for knapsack cover.

MSK_DINF_MIO_OBJ_ABS_GAP (22)
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 (23)
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 (24)
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 (25)
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_OPTIMIZER_TIME (26)
Total time spent in the optimizer.

MSK_DINF_MIO_PROBING_TIME (27)
Total time for probing.

MSK_DINF_MIO_ROOT_CUTGEN_TIME (28)
Total time for cut generation.

MSK_DINF_MIO_ROOT_OPTIMIZER_TIME (29)
Time spent in the optimizer while solving the root relaxation.

MSK_DINF_MIO_ROOT_PRESOLVE_TIME (30)
Time spent in while presolving the root relaxation.

MSK_DINF_MIO_TIME (31)
Time spent in the mixed-integer optimizer.

MSK_DINF_MIO_USER_OBJ_CUT (32)
If the objective cut is used, then this information item has the value of the cut.

MSK_DINF_OPTIMIZER_TIME (33)
Total time spent in the optimizer since it was invoked.

MSK_DINF_PRESOLVE_ELI_TIME (34)
Total time spent in the eliminator since the presolve was invoked.

MSK_DINF_PRESOLVE_LINDEP_TIME (35)
Total time spent in the linear dependency checker since the presolve was invoked.

MSK_DINF_PRESOLVE_TIME (36)
Total time (in seconds) spent in the presolve since it was invoked.

MSK_DINF_PRIMAL_REPAIR_PENALTY_OBJ (37)
The optimal objective value of the penalty function.

MSK_DINF_QCQO_REFORMULATE_MAX_PERTURBATION (38)
: Maximum absolute diagonal perturbation occuring during the QCQO reformulation.

MSK_DINF_QCQO_REFORMULATE_TIME (39)
: Time spent with conic quadratic reformulation.

MSK_DINF_QCQO_REFORMULATE_WORST_CHOLESKY_COLUMN_SCALING (40)
: Worst Cholesky column scaling.

MSK_DINF_QCQO_REFORMULATE_WORST_CHOLESKY_DIAG_SCALING (41)
: Worst Cholesky diagonal scaling.

MSK_DINF_RD_TIME (42)
: Time spent reading the data file.

MSK_DINF_SIM_DUAL_TIME (43)
: Time spent in the dual simplex optimizer since invoking it.

MSK_DINF_SIM_FEAS (44)
: Feasibility measure reported by the simplex optimizer.

MSK_DINF_SIM_OBJ (45)
: Objective value reported by the simplex optimizer.

MSK_DINF_SIM_PRIMAL_TIME (46)
: Time spent in the primal simplex optimizer since invoking it.

MSK_DINF_SIM_TIME (47)
: Time spent in the simplex optimizer since invoking it.

MSK_DINF_SOL_BAS_DUAL_OBJ (48)
: Dual objective value of the basic solution.

MSK_DINF_SOL_BAS_DVIOLCON (49)
: Maximal dual bound violation for x^c in the basic solution.

MSK_DINF_SOL_BAS_DVIOLVAR (50)
: Maximal dual bound violation for x^x in the basic solution.

MSK_DINF_SOL_BAS_NRM_BARX (51)
: Infinity norm of $\overline{X}$ in the basic solution.

MSK_DINF_SOL_BAS_NRM_SLC (52)
: Infinity norm of s_l^c in the basic solution.

MSK_DINF_SOL_BAS_NRM_SLX (53)
: Infinity norm of s_l^x in the basic solution.

MSK_DINF_SOL_BAS_NRM_SUC (54)
: Infinity norm of s_u^c in the basic solution.

MSK_DINF_SOL_BAS_NRM_SUX (55)
: Infinity norm of s_u^X in the basic solution.

MSK_DINF_SOL_BAS_NRM_XC (56)
: Infinity norm of x^c in the basic solution.

MSK_DINF_SOL_BAS_NRM_XX (57)
: Infinity norm of x^x in the basic solution.

MSK_DINF_SOL_BAS_NRM_Y (58)
: Infinity norm of y in the basic solution.

MSK_DINF_SOL_BAS_PRIMAL_OBJ (59)
: Primal objective value of the basic solution.

MSK_DINF_SOL_BAS_PVIOLCON (60)
: Maximal primal bound violation for x^c in the basic solution.

MSK_DINF_SOL_BAS_PVIOLVAR (61)
: Maximal primal bound violation for x^x in the basic solution.

MSK_DINF_SOL_ITG_NRM_BARX (62)
: Infinity norm of $\overline{X}$ in the integer solution.

MSK_DINF_SOL_ITG_NRM_XC (63)
: Infinity norm of x^c in the integer solution.

MSK_DINF_SOL_ITG_NRM_XX (64)
: Infinity norm of x^x in the integer solution.

MSK_DINF_SOL_ITG_PRIMAL_OBJ (65)
: Primal objective value of the integer solution.

MSK_DINF_SOL_ITG_PVIOLBARVAR (66)
: Maximal primal bound violation for $\overline{X}$ in the integer solution.

MSK_DINF_SOL_ITG_PVIOLCON (67)
: Maximal primal bound violation for x^c in the integer solution.

MSK_DINF_SOL_ITG_PVIOLCONES (68)
: Maximal primal violation for primal conic constraints in the integer solution.

MSK_DINF_SOL_ITG_PVIOLITG (69)
: Maximal violation for the integer constraints in the integer solution.

MSK_DINF_SOL_ITG_PVIOLVAR (70)
: Maximal primal bound violation for x^x in the integer solution.

MSK_DINF_SOL_ITR_DUAL_OBJ (71)
: Dual objective value of the interior-point solution.

MSK_DINF_SOL_ITR_DVIOLBARVAR (72)
: Maximal dual bound violation for $\overline{X}$ in the interior-point solution.

MSK_DINF_SOL_ITR_DVIOLCON (73)
: Maximal dual bound violation for x^c in the interior-point solution.

MSK_DINF_SOL_ITR_DVIOLCONES (74)
: Maximal dual violation for dual conic constraints in the interior-point solution.

MSK_DINF_SOL_ITR_DVIOLVAR (75)
: Maximal dual bound violation for x^x in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_BARS (76)
: Infinity norm of $\overline{S}$ in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_BARX (77)
: Infinity norm of $\overline{X}$ in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SLC (78)
: Infinity norm of s_l^c in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SLX (79)
: Infinity norm of s_l^x in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SNX (80)
: Infinity norm of s_n^x in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SUC (81)
: Infinity norm of s_u^c in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_SUX (82)
: Infinity norm of s_u^X in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_XC (83)
: Infinity norm of x^c in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_XX (84)
Infinity norm of x^x in the interior-point solution.

MSK_DINF_SOL_ITR_NRM_Y (85)
Infinity norm of y in the interior-point solution.

MSK_DINF_SOL_ITR_PRIMAL_OBJ (86)
Primal objective value of the interior-point solution.

MSK_DINF_SOL_ITR_PVIOLBARVAR (87)
Maximal primal bound violation for $\overline{X}$ in the interior-point solution.

MSK_DINF_SOL_ITR_PVIOLCON (88)
Maximal primal bound violation for x^c in the interior-point solution.

MSK_DINF_SOL_ITR_PVIOLCONES (89)
Maximal primal violation for primal conic constraints in the interior-point solution.

MSK_DINF_SOL_ITR_PVIOLVAR (90)
Maximal primal bound violation for x^x in the interior-point solution.

MSK_DINF_TO_CONIC_TIME (91)
Time spent in the last to conic reformulation.

MSKfeaturee
License feature

MSK_FEATURE_PTS (0)
Base system.

MSK_FEATURE_PTON (1)
Nonlinear extension.

MSKliinfiteme
Long integer information items.

MSK_LIINF_BI_CLEAN_DUAL_DEG_ITER (0)
Number of dual degenerate clean iterations performed in the basis identification.

MSK_LIINF_BI_CLEAN_DUAL_ITER (1)
Number of dual clean iterations performed in the basis identification.

MSK_LIINF_BI_CLEAN_PRIMAL_DEG_ITER (2)
Number of primal degenerate clean iterations performed in the basis identification.

MSK_LIINF_BI_CLEAN_PRIMAL_ITER (3)
Number of primal clean iterations performed in the basis identification.

MSK_LIINF_BI_DUAL_ITER (4)
Number of dual pivots performed in the basis identification.

MSK_LIINF_BI_PRIMAL_ITER (5)
Number of primal pivots performed in the basis identification.

MSK_LIINF_INTPNT_FACTOR_NUM_NZ (6)
Number of non-zeros in factorization.

MSK_LIINF_MIO_INTPNT_ITER (7)
Number of interior-point iterations performed by the mixed-integer optimizer.

MSK_LIINF_MIO_PRESOLVED_ANZ (8)
Number of non-zero entries in the constraint matrix of presolved problem.

MSK_LIINF_MIO_SIM_MAXITER_SETBACKS (9)
Number of times the the simplex optimizer has hit the maximum iteration limit when re-optimizing.

MSK_LIINF_MIO_SIMPLEX_ITER (10)
Number of simplex iterations performed by the mixed-integer optimizer.

MSK_LIINF_RD_NUMANZ (11)
Number of non-zeros in A that is read.

MSK_LIINF_RD_NUMQNZ (12)
Number of Q non-zeros.

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

  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_NUM_ROUNDINGS (21)
: Number of values in the integer solution that is rounded to an integer value.

MSK_IINF_MIO_CONSTRUCT_SOLUTION (22)
: If this item has the value 0, then **MOSEK** did not try to construct an initial integer feasible solution. If the item has a positive value, then **MOSEK** successfully constructed an initial integer feasible solution.

MSK_IINF_MIO_INITIAL_SOLUTION (23)
: Is non-zero if an initial integer solution is specified.

MSK_IINF_MIO_NEAR_ABSGAP_SATISFIED (24)
: Non-zero if absolute gap is within relaxed tolerances.

MSK_IINF_MIO_NEAR_RELGAP_SATISFIED (25)
: Non-zero if relative gap is within relaxed tolerances.

MSK_IINF_MIO_NODE_DEPTH (26)
: Depth of the last node solved.

MSK_IINF_MIO_NUM_ACTIVE_NODES (27)
: Number of active branch bound nodes.

MSK_IINF_MIO_NUM_BRANCH (28)
: Number of branches performed during the optimization.

MSK_IINF_MIO_NUM_CLIQUE_CUTS (29)
: Number of clique cuts.

MSK_IINF_MIO_NUM_CMIR_CUTS (30)
: Number of Complemented Mixed Integer Rounding (CMIR) cuts.

MSK_IINF_MIO_NUM_GOMORY_CUTS (31)
: Number of Gomory cuts.

MSK_IINF_MIO_NUM_IMPLIED_BOUND_CUTS (32)
: Number of implied bound cuts.

MSK_IINF_MIO_NUM_INT_SOLUTIONS (33)
: Number of integer feasible solutions that has been found.

MSK_IINF_MIO_NUM_KNAPSACK_COVER_CUTS (34)
: Number of clique cuts.

MSK_IINF_MIO_NUM_RELAX (35)
: Number of relaxations solved during the optimization.

MSK_IINF_MIO_NUM_REPEATED_PRESOLVE (36)
: Number of times presolve was repeated at root.

MSK_IINF_MIO_NUMCON (37)
: Number of constraints in the problem solved by the mixed-integer optimizer.

MSK_IINF_MIO_NUMINT (38)
: Number of integer variables in the problem solved be the mixed-integer optimizer.

MSK_IINF_MIO_NUMVAR (39)
: Number of variables in the problem solved by the mixed-integer optimizer.

MSK_IINF_MIO_OBJ_BOUND_DEFINED (40)
: Non-zero if a valid objective bound has been found, otherwise zero.

MSK_IINF_MIO_PRESOLVED_NUMBIN (41)
: Number of binary variables in the problem solved be the mixed-integer optimizer.

MSK_IINF_MIO_PRESOLVED_NUMCON (42)
: Number of constraints in the presolved problem.

MSK_IINF_MIO_PRESOLVED_NUMCONT (43)
: Number of continuous variables in the problem solved be the mixed-integer optimizer.

MSK_IINF_MIO_PRESOLVED_NUMINT (44)
: Number of integer variables in the presolved problem.

MSK_IINF_MIO_PRESOLVED_NUMVAR (45)
: Number of variables in the presolved problem.

MSK_IINF_MIO_RELGAP_SATISFIED (46)
: Non-zero if relative gap is within tolerances.

MSK_IINF_MIO_TOTAL_NUM_CUTS (47)
: Total number of cuts generated by the mixed-integer optimizer.

MSK_IINF_MIO_USER_OBJ_CUT (48)
: If it is non-zero, then the objective cut is used.

MSK_IINF_OPT_NUMCON (49)
: Number of constraints in the problem solved when the optimizer is called.

MSK_IINF_OPT_NUMVAR (50)
: Number of variables in the problem solved when the optimizer is called

MSK_IINF_OPTIMIZE_RESPONSE (51)
: The response code returned by optimize.

MSK_IINF_RD_NUMBARVAR (52)
: Number of variables read.

MSK_IINF_RD_NUMCON (53)
: Number of constraints read.

MSK_IINF_RD_NUMCONE (54)
: Number of conic constraints read.

MSK_IINF_RD_NUMINTVAR (55)
: Number of integer-constrained variables read.

MSK_IINF_RD_NUMQ (56)
Number of nonempty Q matrices read.

MSK_IINF_RD_NUMVAR (57)
Number of variables read.

MSK_IINF_RD_PROTYPE (58)
Problem type.

MSK_IINF_SIM_DUAL_DEG_ITER (59)
The number of dual degenerate iterations.

MSK_IINF_SIM_DUAL_HOTSTART (60)
If 1 then the dual simplex algorithm is solving from an advanced basis.

MSK_IINF_SIM_DUAL_HOTSTART_LU (61)
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 (62)
The number of iterations taken with dual infeasibility.

MSK_IINF_SIM_DUAL_ITER (63)
Number of dual simplex iterations during the last optimization.

MSK_IINF_SIM_NUMCON (64)
Number of constraints in the problem solved by the simplex optimizer.

MSK_IINF_SIM_NUMVAR (65)
Number of variables in the problem solved by the simplex optimizer.

MSK_IINF_SIM_PRIMAL_DEG_ITER (66)
The number of primal degenerate iterations.

MSK_IINF_SIM_PRIMAL_HOTSTART (67)
If 1 then the primal simplex algorithm is solving from an advanced basis.

MSK_IINF_SIM_PRIMAL_HOTSTART_LU (68)
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 (69)
The number of iterations taken with primal infeasibility.

MSK_IINF_SIM_PRIMAL_ITER (70)
Number of primal simplex iterations during the last optimization.

MSK_IINF_SIM_SOLVE_DUAL (71)
Is non-zero if dual problem is solved.

MSK_IINF_SOL_BAS_PROSTA (72)
Problem status of the basic solution. Updated after each optimization.

MSK_IINF_SOL_BAS_SOLSTA (73)
Solution status of the basic solution. Updated after each optimization.

MSK_IINF_SOL_ITG_PROSTA (74)
Problem status of the integer solution. Updated after each optimization.

MSK_IINF_SOL_ITG_SOLSTA (75)
Solution status of the integer solution. Updated after each optimization.

MSK_IINF_SOL_ITR_PROSTA (76)
Problem status of the interior-point solution. Updated after each optimization.

MSK_IINF_SOL_ITR_SOLSTA (77)
Solution status of the interior-point solution. Updated after each optimization.

MSK_IINF_STO_NUM_A_REALLOC (78)
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 written.

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.

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_WORST (3)`
The optimizer employs a worst bound node selection strategy.

`MSK_MIO_NODE_SELECTION_HYBRID (4)`
The optimizer employs a hybrid strategy.

`MSK_MIO_NODE_SELECTION_PSEUDO (5)`
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_GECO (3)
General convex optimization.

MSK_PROBTYPE_CONIC (4)
A conic optimization.

MSK_PROBTYPE_MIXED (5)
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_NEAR_PRIM_AND_DUAL_FEAS (8)
The problem is at least nearly primal and dual feasible.

MSK_PRO_STA_NEAR_PRIM_FEAS (9)
The problem is at least nearly primal feasible.

MSK_PRO_STA_NEAR_DUAL_FEAS (10)
The problem is at least nearly 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 (11)`
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.

`MSK_SCALING_MODERATE (2)`
A conservative scaling is performed.

`MSK_SCALING_AGGRESSIVE (3)`
A very aggressive 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.

`MSK_SENSITIVITY_TYPE_OPTIMAL_PARTITION (1)`
Optimal partition 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_NEAR_OPTIMAL (7)
The solution is nearly optimal.

MSK_SOL_STA_NEAR_PRIM_FEAS (8)
The solution is nearly primal feasible.

MSK_SOL_STA_NEAR_DUAL_FEAS (9)
The solution is nearly dual feasible.

MSK_SOL_STA_NEAR_PRIM_AND_DUAL_FEAS (10)
The solution is nearly 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_NEAR_PRIM_INFEAS_CER (11)
The solution is almost a certificate of primal infeasibility.

MSK_SOL_STA_NEAR_DUAL_INFEAS_CER (12)
The solution is almost a certificate of dual infeasibility.

MSK_SOL_STA_PRIM_ILLPOSED_CER (13)
The solution is a certificate that the primal problem is illposed.

MSK_SOL_STA_DUAL_ILLPOSED_CER (14)
The solution is a certificate that the dual problem is illposed.

MSK_SOL_STA_INTEGER_OPTIMAL (15)
The primal solution is integer optimal.

MSK_SOL_STA_NEAR_INTEGER_OPTIMAL (16)
The primal solution is near 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.

16.8 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.

16.9 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.

MSKcallocfunc

```
void *  (MSKAPI * MSKcallocfunc) (
  MSKuserhandle_t usrptr,
  const size_t num,
  const size_t size)
```

A user-defined memory allocation function. The function must be compatible with the C calloc function.

Parameters

- usrptr (*MSKuserhandle_t*) – A pointer to a user-defined structure. (input)
- num (size_t) – The number of elements. (input)
- size (size_t) – The size of an element. (input)

Return (void*) – A pointer to the allocated memory.

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 (*MSKstring_t*) – 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 (*MSKstring_t*) – A message about the error. (input)

Return (void)

MSKfreefunc

```
void  (MSKAPI * MSKfreefunc) (
  MSKuserhandle_t usrptr,
  void * buffer)
```

A user-defined memory freeing function.

Parameters

- `usrptr` (*MSKuserhandle_t*) – A pointer to a user-defined structure. (input)
- `buffer` (`void*`) – A pointer to the buffer which should be freed. (input/output)

Return (`void`)

MSKmallocfunc

```
void *  (MSKAPI * MSKmallocfunc) (
  MSKuserhandle_t usrptr,
  const size_t size)
```

A user-defined memory allocation function. The function must be compatible with the C `malloc` function.

Parameters

- `usrptr` (*MSKuserhandle_t*) – A pointer to a user-defined structure. (input)
- `size` (`size_t`) – The number of characters to allocate. (input)

Return (`void*`) – A pointer to the allocated memory.

MSKnlgetspfunc

```
MSKint32t  (MSKAPI * MSKnlgetspfunc) (
  MSKuserhandle_t nlhandle,
  MSKint32t * numgrdobjnz,
  MSKint32t * grdobjsub,
  MSKint32t i,
  MSKbooleant * convali,
  MSKint32t * grdconinz,
  MSKint32t * grdconisub,
  MSKint32t yo,
  MSKint32t numycnz,
  const MSKint32t * ycsub,
  MSKint32t maxnumhesnz,
  MSKint32t * numhesnz,
  MSKint32t * hessubi,
  MSKint32t * hessubj)
```

Type definition of the callback function which is used to provide structural information about the nonlinear functions f and g in the optimization problem.

Hence, it is the user's responsibility to provide a function satisfying the definition.

The user *must not* call any **MOSEK** function directly or indirectly from the callback function.

Parameters

- `nlhandle` (*MSKuserhandle_t*) – A pointer to a user-defined data structure specified when the function is attached to a task using the function *MSK_putnlfunc*. (input/output)

- numgrdobjnz (*MSKint32t* *) – If requested, numgrdobjnz should be assigned the number of non-zero elements in the gradient of f. (output)
- grdobjsub (*MSKint32t* *) – If requested then it contains the positions of the non-zero elements in the gradient of f. The elements are stored in

$$\texttt{grdobjsub}[0, .., numgrdobjnz - 1].$$

  (output)
- i (*MSKint32t*) – Index of a constraint. If $i < 0$ or $i \geq$ numcon, no information about a constraint is requested. (input)
- convali (*MSKbooleant* *) – If requested, assign a true/false value indicating if constraint i contains general nonlinear terms. (output)
- grdconinz (*MSKint32t* *) – If requested, it should be assigned the number of non-zero elements in $\nabla g_i(x)$. (output)
- grdconisub (*MSKint32t* *) – If requested, this array shall contain the indexes of the non-zeros in $\nabla g_i(x)$. The length of the array must be the same as given in grdconinz. (output)
- yo (*MSKint32t*) – If non-zero, then f shall be included when the gradient and the Hessian of the Lagrangian are computed. (input)
- numycnz (*MSKint32t*) – Number of constraint functions which are included in the definition of the Lagrangian. See (16.5). (input)
- ycsub (*MSKint32t* *) – Indexes of constraint functions which are included in the definition of the Lagrangian. See (16.5). (input)
- maxnumhesnz (*MSKint32t*) – Length of the arguments hessubi and hessubj. (input)
- numhesnz (*MSKint32t* *) – If requested, numhesnz should be assigned the number of non-zero elements in the lower triangular part of the Hessian of the Lagrangian:

$$L := \texttt{yo} f(x) - \sum_{k=0}^{numycnz-1} g_{\texttt{ycsub[k]}}(x). \tag{16.5}$$

  (output)
- hessubi (*MSKint32t* *) – If requested, hessubi and hessubj are used to convey the position of the non-zeros in the Hessian of the Lagrangian L (see (16.5)) as follows

$$\nabla^2 L_{\texttt{hessubi}[k],\texttt{hessubj}[k]}(x) \neq 0.0$$

  for $k = 0, \ldots, \texttt{numhesnz} - 1$.

  All other positions in L are assumed to be zero. Please note that *only* the lower *or* the upper triangular part of the Hessian should be return. (output)
- hessubj (*MSKint32t* *) – See the argument *hessubi*. (output)

Return (*MSKint32t*) – If the return is non-zero, **MOSEK** assumes that an error occurred during the structure computation, and optimization will be terminated.

MSKnlgetvafunc

```
MSKint32t  (MSKAPI * MSKnlgetvafunc) (
  MSKuserhandle_t nlhandle,
  const MSKrealt * xx,
```

```
MSKrealt yo,
const MSKrealt * yc,
MSKrealt * objval,
MSKint32t * numgrdobjnz,
MSKint32t * grdobjsub,
MSKrealt * grdobjval,
MSKint32t numi,
const MSKint32t * subi,
MSKrealt * conval,
const MSKint32t * grdconptrb,
const MSKint32t * grdconptre,
const MSKint32t * grdconsub,
MSKrealt * grdconval,
MSKrealt * grdlag,
MSKint32t maxnumhesnz,
MSKint32t * numhesnz,
MSKint32t * hessubi,
MSKint32t * hessubj,
MSKrealt * hesval)
```

Type definition of the callback function which is used to provide structural and numerical information about the nonlinear functions f and g in an optimization problem.

For later use we need the definition of the Lagrangian L which is given by

$$L := \mathtt{yo} * f(\mathtt{xx}) - \sum_{k=0}^{numi-1} \mathtt{yc}_{\mathtt{subi[k]}} g_{\mathtt{subi[k]}}(\mathtt{xx}). \tag{16.6}$$

The user *must not* call any **MOSEK** function directly or indirectly from the callback function.

Parameters

- `nlhandle` (*MSKuserhandle_t*) – A pointer to a user-defined data structure. The pointer is passed to **MOSEK** when the function *MSK_putnlfunc* is called. (input/output)
- `xx` (*MSKrealt* *) – The point at which the nonlinear function must be evaluated. The length equals the number of variables in the task. (input)
- `yo` (*MSKrealt*) – Multiplier on the objective function f. (input)
- `yc` (*MSKrealt* *) – Multipliers for the constraint functions g_i. The length is `numcon`. (input)
- `objval` (*MSKrealt* *) – If requested, `objval` shall be assigned the value of f evaluated at xx. (output)
- `numgrdobjnz` (*MSKint32t* *) – If requested, `numgrdobjnz` shall be assigned the number of non-zero elements in the gradient of f. (output)
- `grdobjsub` (*MSKint32t* *) – If requested, it shall contain the position of the non-zero elements in the gradient of f. The elements are stored in

  $$\mathtt{grdobjsub}[0, \ldots, numgrdobjnz - 1].$$

  (output)
- `grdobjval` (*MSKrealt* *) – If requested, it shall contain the gradient of f evaluated at `xx`. The following data structure

  $$\mathtt{grdobjval}[\mathrm{k}] = \frac{\partial f}{\partial x_{\mathtt{grdobjsub[k]}}}(\mathtt{xx})$$

  for $k = 0, \ldots, numgrdobjnz - 1$ is used. (output)
- `numi` (*MSKint32t*) – Number of elements in `subi`. (input)

- subi (*MSKint32t* *) – subi$[0, ..., numi-1]$ contain the indexes of the constraints that has to be evaluated. The length is numi. (input)
- conval (*MSKrealt* *) – $g(\texttt{xx})$ for the required constraint functions i.e.

  $$\texttt{conval}[\texttt{k}] = g_{\texttt{subi}[\texttt{k}]}(\texttt{xx})$$

  for $k = 0, \ldots, numi - 1$. (output)
- grdconptrb (*MSKint32t* *) – If given, it specifies the structure of the gradients of the constraint functions. See the argument grdconval for details. (input)
- grdconptre (*MSKint32t* *) – If given, it specifies the structure of the gradients of the constraint functions. See the argument grdconval for details. (input)
- grdconsub (*MSKint32t* *) – It shall specifies the positions of the non-zeros in the gradients of the constraints. See the argument grdconval for details. (input)
- grdconval (*MSKrealt* *) – If requested, it shall specify the values of the gradient of the nonlinear constraints.

  Together grdconptrb, grdconptre, grdconsub and grdconval are used to specify the gradients of the nonlinear constraint functions.

  The gradient data is stored as follows

  $$\begin{array}{l} \texttt{grdconval}[\texttt{k}] = \frac{\partial g_{\texttt{subi}[\texttt{i}]}(xx)}{\partial xx_{\texttt{grdconsub}[\texttt{k}]}}, \quad \text{for} \\ \quad k = \texttt{grdconptrb}[i], \ldots, \texttt{grdconptre}[i] - 1, \\ \quad i = 0, \ldots, numi - 1. \end{array}$$

  (output)
- grdlag (*MSKrealt* *) – If requested, grdlag shall contain the gradient of the Lagrangian function, i.e.

  $$\texttt{grdlag} = \nabla L.$$

  (output)
- maxnumhesnz (*MSKint32t*) – Maximum number of non-zeros in the Hessian of the Lagrangian, i.e. maxnumhesnz is the length of the arrays hessubi, hessubj, and hesval. (input)
- numhesnz (*MSKint32t* *) – If requested, numhesnz shall be assigned the number of non-zeros elements in the Hessian of the Lagrangian L. See (16.6). (output)
- hessubi (*MSKint32t* *) – See the argument hesval. (output)
- hessubj (*MSKint32t* *) – See the argument hesval. (output)
- hesval (*MSKrealt* *) – Together hessubi, hessubj, and hesval specify the Hessian of the Lagrangian function L defined in (16.6).

  The Hessian is stored in the following format:

  $$\texttt{hesval}[k] = \nabla^2 L_{\min(\texttt{hessubi}[k],\texttt{hessubj}[k]),\max(\texttt{hessubi}[k],\texttt{hessubj}[k])}$$

  for $k = 0, \ldots, numhesnz[0] - 1$. Please note that if an element is specified multiple times, then the elements are added together. Hence, *only* the lower *or* the upper triangular part of the Hessian should be returned. (output)

Return (*MSKint32t*) – If the return value is non-zero, **MOSEK** will assume an error happened during the function evaluation.

MSKreallocfunc

```
void *  (MSKAPI * MSKreallocfunc) (
  MSKuserhandle_t usrptr,
  void * ptr,
  const size_t size)
```

A user-defined memory reallocation function. The function must be compatible with the C `realloc` function.

Parameters

- `usrptr` (*MSKuserhandle_t*) – A pointer to a user-defined structure. (input)
- `ptr` (`void*`) – The pointer to the allocated memory. (input/output)
- `size` (`size_t`) – Size of the new block. (input)

Return (`void*`) – A pointer to the allocated memory.

`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` (*MSKstring_t*) – 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` (*MSKstring_t*) – A string containing a message to a stream. (input)

Return (`void`)

16.10 Nonlinear extensions

16.10.1 Separable Convex Optimization (SCopt)

SCopt is an easy-to-use interface to the nonlinear optimizer when solving separable convex problems. See Sec. 8.1 for a tutorial and example code. As currently implemented, SCopt can handle only the nonlinear expressions $x\ln(x)$, e^x, $\ln(x)$, and x^g. However, it should be fairly easy to extend the interface to other nonlinear function of a single variable if needed.

The code using the SCopt interface must include `scopt-ext.h` and must be linked with `scopt-ext.c`. Both extension files can be found in the `examples/c` directory.

All the linear data of the problem, such as c and A, is inputted to **MOSEK** as usual, i.e. using the relevant functions in the **MOSEK** API. Every nonlinear expression added to the objective should be specified by a 5-tuple of parameters:

opro[k]	oprjo[k]	oprfo[k]	oprgo[k]	oprho[k]	Expression added in objective
MSK_OPR_ENT	j	f	g	h	$fx_j\ln(x_j)$
MSK_OPR_EXP	j	f	g	h	fe^{gx_j+h}
MSK_OPR_LOG	j	f	g	h	$f\ln(gx_j+h)$
MSK_OPR_POW	j	f	g	h	$f(x_j+h)^g$

Every nonlinear expression added to the constraints should be specified by a 6-tuple of parameters:

oprc[k]	opric[k]	oprjc[k]	oprfc[k]	oprgc[k]	oprhc[k]	Expression added to constraint i
MSK_OPR_ENT	i	j	f	g	h	$fx_j\ln(x_j)$
MSK_OPR_EXP	i	j	f	g	h	fe^{gx_j+h}
MSK_OPR_LOG	i	j	f	g	h	$f\ln(gx_j+h)$
MSK_OPR_POW	i	j	f	g	h	$f(x_j+h)^g$

In each case `opr` specifies the kind of expression to be added, `oprf`, `oprg` and `oprh` are the parameters and `opri`, `oprj` determine the variable and/or constraint to be considered. The concrete API specification follows.

`MSKscopre`

Type of nonlinear term in the SCopt interface.

`MSK_OPR_ENT (0)`

Entropy function $fx\ln(x)$

`MSK_OPR_EXP (1)`

Exponential function fe^{gx+h}

`MSK_OPR_LOG (2)`

Logarithm $f\ln(gx+h)$

`MSK_OPR_POW (3)`

Power function $f(x+h)^g$

`MSK_scbegin`

```
MSKrescodee MSK_scbegin(
   MSKtask_t task,
   int       numopro,
   int       *opro,
   int       *oprjo,
   double    *oprfo,
   double    *oprgo,
```

```
   double    *oprho,
   int       numoprc,
   int       *oprc,
   int       *opric,
   int       *oprjc,
   double    *oprfc,
   double    *oprgc,
   double    *oprhc,
   schand_t  *sch)
```

Define the nonlinear part of the problem in the format specified by the SCopt interface. The `o` arguments describe the nonlinear terms added to the objective and the `c` arguments describe the nonlinear terms added to the constraints. Multiple terms involving the same variable and constraint are possible, they will be added up.

Parameters

- `task` (*MSKtask_t*) – The optimization task. (input)
- `numopro` (`int`) – Number of nonlinear terms in the objective. (input)
- `opro` (*MSKscopre**) – List of function indicators defining the objective terms. (input)
- `oprjo` (`int*`) – List of variable indexes for the objective terms. (input)
- `oprfo` (`double*`) – List of f values for the objective terms. (input)
- `oprgo` (`double*`) – List of g values for the objective terms. (input)
- `oprho` (`double*`) – List of h values for the objective terms. (input)
- `numopro` – Number of nonlinear terms in the constraints.
- `oprc` (*MSKscopre**) – List of function indicators defining the constraint terms. (input)
- `opric` (`int*`) – List of constraint indexes for the constraint terms. (input)
- `oprjc` (`int*`) – List of variable indexes for the constraint terms. (input)
- `oprfc` (`double*`) – List of f values for the constraint terms. (input)
- `oprgc` (`double*`) – List of g values for the constraint terms. (input)
- `oprhc` (`double*`) – List of h values for the constraint terms. (input)
- `sch` (`void*` *by reference*) – A handle to the nonlinear data part. (output)

`MSK_scend`

```
MSKrescodee MSK_scend(
   MSKtask_t task,
   schand_t  *sch)
```

Remove all non-linear separable terms from the task.

Parameters

- `task` (*MSKtask_t*) – The optimization task. (input)
- `sch` (`void*` *by reference*) – A handle to the nonlinear data part. (input/output)

Return (*MSKrescodee*) – The function response code.

`MSK_scwrite`

```
MSKrescodee MSK_scwrite(
   MSKtask_t task,
   schand_t  sch,
   char      *filename)
```

Write the problem to an SCopt file and a normal problem file.

Parameters

- `task` (*MSKtask_t*) – The optimization task. (input)
- `sch` (`void*`) – A handle to the nonlinear data part. (input)
- `filename` (`char*`) – Any string. The nonlinear part of the problem is written to `filename.sco` and the linear part to `filename.mps`. (input)

Return (*MSKrescodee*) – The function response code.

`MSK_scread`

```
MSKrescodee MSK_scread(
   MSKtask_t task,
   schand_t  *sch,
   char      *filename)
```

Read a problem from files written by *MSK_scwrite*.

Parameters

- `task` (*MSKtask_t*) – The optimization task. (input)
- `sch` (`void*` *by reference*) – A handle to the nonlinear data part. (output)
- `filename` (`char*`) – Any valid file name. (input)

Return (*MSKrescodee*) – The function response code.

16.10.2 Exponential optimization

MOSEK has an extension for exponential optimization problems (EXPopt). See Sec. 8.2 for a tutorial and code examples. The code using this interface must include `expopt.h` and must be linked with `expopt.c`, `dgopt.c` and `scopt-ext.c`. These files can be found in the `examples/c` directory.

`MSK_expoptsetup`

```
MSKrescodee MSK_expoptsetup(
    MSKtask_t     expopttask,
    MSKint32t     solveform,
    MSKint32t     numcon,
    MSKint32t     numvar,
    MSKint32t     numter,
    MSKidxt       *subi,
    double        *c,
    MSKidxt       *subk,
    MSKidxt       *subj,
    double        *akj,
    MSKint32t     numanz,
    expopthand_t  *expopthnd)
```

Sets up an exponential optimization problem.

Parameters

- `expopttask` (*MSKtask_t*) – The optimization task.

- solveform (*MSKint32t*) – If 0 solver is chosen freely, 1: solve dual, -1: solve primal
- numcon (*MSKint32t*) – Number of constraints
- numvar (*MSKint32t*) – Number of variables
- numter (*MSKint32t*) – Number of exponential terms
- subi (*MSKint32t* *) – The constraint where the term belongs. Zero denotes the objective.
- c (double*) – The c coefficients of nonlinear terms.
- subk (*MSKint32t* *) – Term indices.
- subj (*MSKint32t* *) – Variable indices.
- akj (double*) – akj[i] is coefficient of variable subj[i] in term subk[i], i.e. $a_{\mathtt{subk}[i],\mathtt{subj}[i]} = \mathtt{akj}[i]$.
- numanz (*MSKint32t*) – Number of linear terms in the exponents, i.e. the length of subk, subj and akj.
- expopthnd (void**) – Data structure containing the nonlinear information.

Return (*MSKrescodee*) – The function response code.

MSK_expoptimize

```
MSKrescodee MSK_expoptimize(
    MSKtask_t    expopttask,
    MSKprostae   *prosta,
    MSKsolstae   *solsta,
    double       *objval,
    double       *xx,
    double       *y,
    expopthand_t *expopthnd)
```

Solves an exponential optimization problem.

Parameters

- expopttask (*MSKtask_t*) – The optimization task.
- prosta (**MSKprostae*) – Problem status.
- solsta (**MSKsolstae*) – Solution status.
- objval (double*) – Objective value.
- xx (double*) – Primal solution values.
- y (double*) – Dual solution values (only when dual form is used).
- expopthnd (void**) – Data structure containing the nonlinear information.

Return (*MSKrescodee*) – The function response code.

MSK_expoptread

```
MSKrescodee MSK_expoptread(
    MSKenv_t      env,
    const char    *filename,
    MSKint32t     *numcon,
    MSKint32t     *numvar,
    MSKint32t     *numter,
    MSKidxt       **subi,
    double        **c,
```

```
    MSKidxt       **subk,
    MSKidxt       **subj,
    double        **akj,
    MSKint32t     *numanz)
```

Reads an exponential optimization problem from a file and saves it into arrays suitable for *MSK_expoptsetup*. The user must manually free the memory allocated by this function.

Parameters

- `env` (*MSKenv_t*) – The environment.
- `filename` (`char*`) – Any valid file name.
- `numcon` (*MSKint32t* `*`) – Number of constraints
- `numvar` (*MSKint32t* `*`) – Number of variables
- `numter` (*MSKint32t* `*`) – Number of exponential terms
- `subi` (*MSKint32t* `**`) – The constraint where the term belongs. Zero denotes the objective.
- `c` (`double**`) – The c coefficients of nonlinear terms.
- `subk` (*MSKint32t* `**`) – Term indices.
- `subj` (*MSKint32t* `**`) – Variable indices.
- `akj` (`double**`) – `akj[i]` is coefficient of variable `subj[i]` in term `subk[i]`, i.e. $a_{\mathtt{subk}[i],\mathtt{subj}[i]} = \mathtt{akj}[i]$.
- `numanz` (*MSKint32t* `*`) – Number of linear terms in the exponents, i.e. the length of `subk`, `subj` and `akj`.

Return (*MSKrescodee*) – The function response code.

`MSK_expoptwrite`

```
MSKrescodee MSK_expoptwrite(
    MSKenv_t      env,
    MSKint32t     numcon,
    MSKint32t     numvar,
    MSKint32t     numter,
    MSKidxt       *subi,
    double        *c,
    MSKidxt       *subk,
    MSKidxt       *subj,
    double        *akj,
    MSKint32t     numanz)
```

Write an exponential optimization problem to a file.

Parameters

- `env` (*MSKenv_t*) – The environment.
- `filename` (`char*`) – Any valid file name.
- `solveform` (*MSKint32t*) – If 0 solver is chosen freely, 1: solve dual, -1: solve primal
- `numcon` (*MSKint32t*) – Number of constraints
- `numvar` (*MSKint32t*) – Number of variables
- `numter` (*MSKint32t*) – Number of exponential terms

- subi (*MSKint32t* *) – The constraint where the term belongs. Zero denotes the objective.
- c (double*) – The c coefficients of nonlinear terms.
- subk (*MSKint32t* *) – Term indices.
- subj (*MSKint32t* *) – Variable indices.
- akj (double*) – akj[i] is coefficient of variable subj[i] in term subk[i], i.e. $a_{\mathtt{subk}[i],\mathtt{subj}[i]} = \mathtt{akj}[i]$.
- numanz (*MSKint32t*) – Number of linear terms in the exponents, i.e. the length of subk, subj and akj.

Return (*MSKrescodee*) – The function response code.

MSK_expoptfree

```
MSKrescodee MSK_expoptfree(
   MSKtask_t    expopttask,
   expopthand_t *expopthnd)
```

Free the data structures allocated by a call to *MSK_expoptsetup*.

Parameters

- expopttask (*MSKtask_t*) – The optimization task.
- expopthnd (void**) – Data structure containing the nonlinear information.

Return (*MSKrescodee*) – The function response code.

16.10.3 Dual geometric optimization

MOSEK has an extension for dual geometric optimization (DGopt). See Sec. 8.3 for a tutorial and code examples. The code using this interface must include dgopt.h and must be linked with dgopt.c and scopt-ext.c. These files can be found in the examples/c directory.

MSK_dgosetup

```
MSKrescodee  MSK_dgosetup(
    MSKtask_t task,
    MSKintt   numvar,
    MSKintt   numcon,
    MSKintt   t,
    double    *v,
    MSKintt   *p,
    dgohand_t *nlh)
```

Sets up the objective in a dual geometric optimization problem

$$f(x) = \sum_{j=0}^{n-1} x_j \ln\left(\frac{v_j}{x_j}\right) + \sum_{i=1}^{t}\left(\sum_{j=p_i}^{p_{i+1}-1} x_j\right) \ln\left(\sum_{j=p_i}^{p_{i+1}-1} x_j\right)$$

Parameters

- task (*MSKtask_t*) – The optimization task.
- numvar (*MSKint32t*) – The number of variables n.
- numcon (*MSKint32t*) – The number of variables in the primal formulation of the problem (p_1).

- t (*MSKint32t*) – The number of constraints in the primal formulation of the problem (t).
- v (double*) – Coefficients v_i.
- p (*MSKint32t* *) – Term indices p_i.
- nlh (void**) – Data structure containing the nonlinear information.

Return (*MSKrescodee*) – The function response code.

MSK_dgoread

```
MSKrescodee  MSK_dgoread(
    MSKtask_t task,
    char      *filename,
    MSKintt   *numvar,
    MSKintt   *numcon,
    MSKintt   *t,
    double    **v,
    MSKintt   **p)
```

Reads the data of a dual geometric problem in the format suitable for *MSK_dgosetup*.

Parameters

- task (*MSKtask_t*) – The optimization task.
- filename (char*) – Any valid file name.
- numvar (*MSKint32t* *) – The number of variables n.
- numcon (*MSKint32t* *) – The number of variables in the primal formulation of the problem (p_1).
- t (*MSKint32t* *) – The number of constraints in the primal formulation of the problem (t).
- v (double**) – Coefficients v_i.
- p (*MSKint32t* **) – Term indices p_i.

Return (*MSKrescodee*) – The function response code.

MSK_freedgo

```
MSKrescodee MSK_freedgo(
   MSKtask_t task,
   dgohand_t  *nlh)
```

Free the data structures allocated by a call to *MSK_dgosetup*.

Parameters

- task (*MSKtask_t*) – The optimization task.
- nlh (void**) – Data structure containing the nonlinear information.

Return (*MSKrescodee*) – The function response code.

CHAPTER

SEVENTEEN

SUPPORTED FILE FORMATS

MOSEK supports a range of problem and solution formats listed in Table 17.1 and Table 17.2. The **Task format** is **MOSEK**'s native binary format and it supports all features that **MOSEK** supports. The **OPF format** is **MOSEK**'s human-readable alternative that supports nearly all features (everything except semidefinite problems). In general, text formats are significantly slower to read, but can be examined and edited directly in any text editor.

Problem formats

See Table 17.1.

Table 17.1: List of supported file formats for optimization problems.

Format Type	Ext.	Binary/Text	LP	QO	CQO	SDP
LP	lp	plain text	X	X		
MPS	mps	plain text	X	X		
OPF	opf	plain text	X	X	X	
CBF	cbf	plain text	X		X	X
OSiL	xml	xml text	X	X		
Task format	task	binary	X	X	X	X
Jtask format	jtask	text	X	X	X	X

Solution formats

See Table 17.2.

Table 17.2: 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	Solution

Compression

MOSEK supports GZIP compression of files. Problem files with an additional `.gz` extension are assumed to be compressed when read, and are automatically compressed when written. For example, a file called

```
problem.mps.gz
```

will be considered as a GZIP compressed MPS file.

17.1 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 on 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.

17.1.1 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 (`[ ]`) 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

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

If the same variable occurs more than once in the linear part, the coefficients are added, so that `4 x1 + 2 x1` is equivalent to `6 x1`. In the quadratic expressions `x1 * x2` is equivalent to `x2 * x1` and, as in the linear part, if the same variables multiplied or squared occur several times their coefficients are added.

Constraints

The second section beginning with one of the keywords

```
subj to
subject to
s.t.
st
```

defines the linear constraint matrix A and the quadratic matrices Q^i.

A constraint contains a name (optional), expressions adhering to the same rules as in the objective and a bound:

```
subject to
con1: x1 + x2 + [ x3^2 ]/2 <= 5.1
```

The bound type (here <=) 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, but **MOSEK** supports defining ranged constraints by using double-colon (`::`) instead of a single-colon (`:`) after the constraint name, i.e.

$$-5 \leq x_1 + x_2 \leq 5 \tag{17.1}$$

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 (17.1) 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
```

17.1.2 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
```

17.1.3 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.

MOSEK Extensions to the LP Format

Some optimization software packages employ a more strict definition of the LP format than the one used by **MOSEK**. The limitations imposed by the strict LP format are the following:

- Quadratic terms in the constraints are not allowed.
- Names can be only 16 characters long.
- Lines must not exceed 255 characters in length.

If an LP formatted file created by **MOSEK** should satisfy the strict definition, then the parameter

- *MSK_IPAR_WRITE_LP_STRICT_FORMAT*

should be set; note, however, that some problems cannot be written correctly as a strict LP formatted file. For instance, all names are truncated to 16 characters and hence they may loose their uniqueness and change the problem.

To get around some of the inconveniences converting from other problem formats, **MOSEK** allows lines to contain 1024 characters and names may have any length (shorter than the 1024 characters).

Internally in **MOSEK** names may contain any (printable) character, many of which cannot be used in LP names. Setting the parameters

- *MSK_IPAR_READ_LP_QUOTED_NAMES* and
- *MSK_IPAR_WRITE_LP_QUOTED_NAMES*

allows **MOSEK** to use quoted names. The first parameter tells **MOSEK** to remove quotes from quoted names e.g, `"x1"`, when reading LP formatted files. The second parameter tells **MOSEK** to put quotes around any semi-illegal name (names beginning with a number or a period) and fully illegal name (containing illegal characters). As double quote is a legal character in the LP format, quoting semi-illegal names makes them legal in the pure LP format as long as they are still shorter than 16 characters. Fully illegal names are still illegal in a pure LP file.

17.1.4 The strict LP format

The LP format is not a formal standard and different vendors have slightly different interpretations of the LP format. To make **MOSEK**'s definition of the LP format more compatible with the definitions of other vendors, use the parameter setting

- *MSK_IPAR_WRITE_LP_STRICT_FORMAT* = *MSK_ON*

This setting may lead to truncation of some names and hence to an invalid LP file. The simple solution to this problem is to use the parameter setting

- *MSK_IPAR_WRITE_GENERIC_NAMES* = *MSK_ON*

which will cause all names to be renamed systematically in the output file.

17.1.5 Formatting of an LP File

A few parameters control the visual formatting of LP files written by **MOSEK** in order to make it easier to read the files. These parameters are

- *MSK_IPAR_WRITE_LP_LINE_WIDTH*
- *MSK_IPAR_WRITE_LP_TERMS_PER_LINE*

The first parameter sets the maximum number of characters on a single line. The default value is 80 corresponding roughly to the width of a standard text document.

The second parameter sets the maximum number of terms per line; a term means a sign, a coefficient, and a name (for example `+ 42 elephants`). The default value is 0, meaning that there is no maximum.

Unnamed Constraints

Reading and writing an LP file with **MOSEK** may change it superficially. If an LP file contains unnamed constraints or objective these are given their generic names when the file is read (however unnamed constraints in **MOSEK** are written without names).

17.2 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]*.

17.2.1 MPS File Structure

The version of the MPS format supported by **MOSEK** allows specification of an optimization problem of the form

$$
\begin{array}{rcccl}
l^c & \leq & Ax + q(x) & \leq & u^c, \\
l^x & \leq & x & \leq & u^x, \\
 & & x \in \mathcal{K}, & & \\
 & & x_{\mathcal{J}} \ \ \text{integer}, & &
\end{array}
\tag{17.2}
$$

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.
- $q : \mathbb{R}^n \to \mathbb{R}$ is a vector of quadratic functions. Hence,

$$q_i(x) = \frac{1}{2} x^T Q^i x$$

where it is assumed that

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

Please note the explicit $\frac{1}{2}$ in the quadratic term and that Q^i is required to be symmetric.

- $\mathcal{K}$ is a convex cone.
- $\mathcal{J} \subseteq \{1, 2, \ldots, n\}$ is an index set of the integer-constrained variables.

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]     [vname3]  [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
```

```
.. code-block:: text

   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. 17.2.9 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.

Section `NAME`

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. The OBJNAME section contains one line at most which has the form

```
objname
```

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 an 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 the values E, G, L, or N with the following interpretation:

Constraint type	l_i^c	u_i^c
E	finite	l_i^c
G	finite	∞
L	$-\infty$	finite
N	$-\infty$	∞

In the MPS format an 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.

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 th constraint and v_1 denotes the value specified by `[value1]`, then the interpretation of v_1 is:

Constraint	l_i^c	u_i^c
type		
`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			

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 term belongs. 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

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

has the following MPS file representation

```
* File: qo1.mps
NAME          qo1
ROWS
N  obj
G  c1
COLUMNS
```

```
x1        c1        1.0
x2        obj       -1.0
x2        c1        1.0
x3        c1        1.0
RHS
rhs       c1        1.0
QSECTION      obj
x1        x1        2.0
x1        x3        -1.0
x2        x2        0.2
x3        x3        2.0
ENDATA
```

Regarding the QSECTIONs please note that:

- Only one QSECTION is allowed for each constraint.
- The QSECTIONs can appear in an arbitrary order after the COLUMNS section.
- All variable names occurring in the QSECTION must already be specified in the COLUMNS section.
- All entries specified in a QSECTION are assumed to belong to the lower triangular part of the quadratic term of Q .

QMATRIX/QUADOBJ (optional)

The QMATRIX and QUADOBJ sections allow to define the quadratic term of the objective function. They differ in how the quadratic term of the objective function is stored:

- QMATRIX It stores all the nonzeros coefficients, withouot taking advantage of the symmetry of the Q matrix.
- QUADOBJ It only store 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 apper 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 $1/2x^T Qx$.

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.

17.2.2 QCMATRIX (optional)

A QCMATRIX section allows to specify the quadratic part of a given constraints. 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 $1/2x^T Qx$.

The example

$$\begin{array}{lccl} \text{minimize} & x_2 & & \\ \text{subject to} & x_1 + x_2 + x_3 & \geq & 1, \\ & \frac{1}{2}(-2x_1x_3 + 0.2x_2^2 + 2x_3^2) & \leq & 10, \\ & x \geq 0 & & \end{array}$$

has the following MPS file representation

```
* File: qo1.mps
NAME          qo1
ROWS
 N  obj
 G  c1
 L  q1
COLUMNS
    x1        c1        1.0
    x2        obj       -1.0
    x2        c1        1.0
    x3        c1        1.0
RHS
    rhs       c1        1.0
    rhs       q1        10.0
QCMATRIX      q1
    x1        x1        2.0
    x1        x3        -1.0
    x3        x1        -1.0
    x2        x2        0.2
    x3        x3        2.0
ENDATA
```

Regarding the `QCMATRIX`s please note that:

- Only one `QCMATRIX` is allowed for each constraint.
- The `QCMATRIX`s 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.
- A `QCMATRIX` does not exploit the symmetry of Q: an off-diagonal entry (i, j) should apperas twice.

17.2.3 `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 which bounds are modified by the record. `??` and `[value1]` are used to modify the bound vectors according to the following table:

??	l_j^x	u_j^x	Made integer (added to $\mathcal{J}$)
FR	$-\infty$	∞	No
FX	v_1	v_1	No
LO	v_1	unchanged	No
MI	$-\infty$	unchanged	No
PL	unchanged	∞	No
UP	unchanged	v_1	No
BV	0	1	Yes
LI	$\lceil v_1 \rceil$	unchanged	Yes
UI	unchanged	$\lfloor v_1 \rfloor$	Yes

v_1 is the value specified by `[value1]`.

17.2.4 CSECTION (optional)

The purpose of the `CSECTION` is to specify the constraint

$$x \in \mathcal{K}.$$

in (17.2). 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:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} \right\}.$$

- Quadratic cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : x_1 \geq \sqrt{\sum_{j=2}^{n^t} x_j^2} \right\}. \tag{17.3}$$

- Rotated quadratic cone:

$$\mathcal{K}_t = \left\{ x \in \mathbb{R}^{n^t} : 2x_1 x_2 \geq \sum_{j=3}^{n^t} x_j^2, \quad x_1, x_2 \geq 0 \right\}. \tag{17.4}$$

In general, only quadratic and rotated quadratic cones are specified in the MPS file whereas membership of the $\mathbb{R}$ set is not. If a variable is not a member of any other cone then it is assumed to be a member of an $\mathbb{R}$ cone.

Next, let us study an example. Assume that the quadratic cone

$$x_4 \geq \sqrt{x_5^2 + x_8^2}$$

and the rotated quadratic cone

$$x_3 x_7 \geq x_1^2 + x_0^2, \quad x_3, x_7 \geq 0,$$

should be specified in the MPS file. One CSECTION is required for each cone and they are specified as follows:

```
*      1         2         3         4         5         6
*23456789012345678901234567890123456789012345678901234567890
CSECTION      konea       0.0           QUAD
x4
x5
x8
CSECTION      koneb       0.0           RQUAD
x7
x3
x1
x0
```

This first CSECTION specifies the cone (17.3) which is given the name konea. This is a quadratic cone which is specified by the keyword QUAD in the CSECTION header. The 0.0 value in the CSECTION header is not used by the QUAD cone.

The second CSECTION specifies the rotated quadratic cone (17.4). Please note the keyword RQUAD in the CSECTION which is used to specify that the cone is a rotated quadratic cone instead of a quadratic cone. The 0.0 value in the CSECTION header is not used by the RQUAD cone.

In general, a CSECTION header has the format

```
CSECTION      [kname1]    [value1]     [ktype]
```

where the requirement for each field are as follows:

Field	Starting Position	Max Width	Required	Description
[kname1]	5	8	Yes	Name of the cone
[value1]	15	12	No	Cone parameter
[ktype]	25		Yes	Type of the cone.

The possible cone type keys are:

Cone type key	Members	Interpretation.
QUAD	≤ 1	Quadratic cone i.e. (17.3).
RQUAD	≤ 2	Rotated quadratic cone i.e. (17.4).

Please note that a quadratic cone must have at least one member whereas a rotated quadratic cone must have at least two members. A record in the CSECTION has the format

```
[vname1]
```

where the requirements for each field are

Field	Starting Position	Max Width	required	Description
[vname1]	2	8	Yes	A valid variable name

The most important restriction with respect to the CSECTION is that a variable must occur in only one CSECTION.

17.2.5 ENDATA

This keyword denotes the end of the MPS file.

17.2.6 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

- IMPORTANT: 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.

17.2.7 General Limitations

- An MPS file should be an ASCII file.

17.2.8 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.

17.2.9 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, it also presents two main limitations:

- A name must not contain any blanks.
- By default a line in the MPS file must not contain more than 1024 characters. However, by modifying the parameter *MSK_IPAR_READ_MPS_WIDTH* an arbitrary large line width will be accepted.

To use the free MPS format instead of the default MPS format the **MOSEK** parameter *MSK_IPAR_READ_MPS_FORMAT* should be changed.

17.3 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.

17.3.1 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 in quotes [/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]
```

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 the subsection `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 the subsections `b` (linear bounds on variables) and `cone` (quadratic cone).

`[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]`. currently supports the *quadratic cone* and the *rotated quadratic cone*.

A conic constraint is defined as a set of variables which belong to a single unique cone.

- A quadratic cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$x_1^2 \geq \sum_{i=2}^{n} x_i^2, \quad x_1 \geq 0.$$

- A rotated quadratic cone of n variables $x_1, \ldots, x_n$ defines a constraint of the form

$$2x_1 x_2 \geq \sum_{i=3}^{n} x_i^2, \quad x_1, x_2 \geq 0.$$

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 quad]  x,y,z,w  [/cone] # quadratic cone
[cone rquad] x,y,z,w  [/cone] # rotated quadratic cone
[/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. Optionally, an attribute can be added `[variables disallow_new_variables]` indicating that if any variable not listed here occurs later in the file it is an error.

[integer]

This contains a space-separated list of variables and defines the constraint that the listed variables must be integer values.

[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 in a subsection is defined as follows:

```
[hint ITEM] value [/hint]
```

where ITEM may be replaced by numvar (number of variables), numcon (number of linear/quadratic constraints), numanz (number of linear non-zeros in constraints) and 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]
```

Note that a [solution]-section must be always specified inside a [solutions]-section. 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.

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]+)?
```

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"
```

17.3.2 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]
```

17.3.3 Writing OPF Files from MOSEK

To write an OPF file set the parameter *MSK_IPAR_WRITE_DATA_FORMAT* to *MSK_DATA_FORMAT_OP* as this ensures that OPF format is used.

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.

17.3.4 Examples

This section contains a set of small examples written in OPF and describing how to formulate linear, quadratic and conic problems.

Linear Example lo1.opf

Consider the example:

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

having the bounds

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

In the OPF format the example is displayed as shown in Listing 17.1.

Listing 17.1: 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

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

This can be formulated in `opf` as shown below.

Listing 17.2: Example of an OPF file for a quadratic problem.

```
[comment]
  The qo1 example in OPF format
[/comment]

[hints]
  [hint NUMVAR] 3 [/hint]
  [hint NUMCON] 1 [/hint]
  [hint NUMANZ] 3 [/hint]
  [hint NUMQNZ] 4 [/hint]
[/hints]

[variables disallow_new_variables]
  x1 x2 x3
[/variables]

[objective minimize 'obj']
  # The quadratic terms are often written with a factor of 1/2 as here,
  # but this is not required.

   - x2 + 0.5 ( 2.0 x1 ^ 2 - 2.0 x3 * x1 + 0.2 x2 ^ 2 + 2.0 x3 ^ 2 )
[/objective]

[constraints]
  [con 'c1'] 1.0 <= x1 + x2 + x3 [/con]
[/constraints]

[bounds]
  [b] 0 <= * [/b]
[/bounds]
```

Conic Quadratic Example `cqo1.opf`

Consider the example:

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

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 17.3.

Listing 17.3: Example of an OPF file for a conic quadratic problem.

```
[comment]
  The cqo1 example in OPF format.
[/comment]

[hints]
  [hint NUMVAR] 6 [/hint]
  [hint NUMCON] 1 [/hint]
  [hint NUMANZ] 3 [/hint]
[/hints]

[variables disallow_new_variables]
  x1 x2 x3 x4 x5 x6
[/variables]

[objective minimize 'obj']
   x4 + x5 + x6
[/objective]

[constraints]
  [con 'c1']  x1 + x2 + 2e+00 x3 = 1e+00 [/con]
[/constraints]

[bounds]
  # We let all variables default to the positive orthant
  [b] 0 <= * [/b]

  # ...and change those that differ from the default
  [b] x4,x5,x6 free [/b]

  # Define quadratic cone: x4 >= sqrt( x1^2 + x2^2 )
  [cone quad 'k1'] x4, x1, x2 [/cone]

  # Define rotated quadratic cone: 2 x5 x6 >= x3^2
  [cone rquad 'k2'] x5, x6, x3 [/cone]
[/bounds]
```

Mixed Integer Example `milo1.opf`

Consider the mixed integer problem:

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

This can be implemented in `OPF` with the file in Listing 17.4.

Listing 17.4: 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]
```

17.4 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) 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 problem structure is separated from the problem data, and the format moreover facilitates benchmarking of hotstart capability through sequences of changes.

17.4.1 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:

$$\begin{array}{lll} \min / \max & g^{obj} & \\ & g_i \in \mathcal{K}_i, & i \in \mathcal{I}, \\ \text{s.t.} & G_i \in \mathcal{K}_i, & i \in \mathcal{I}^{PSD}, \\ & x_j \in \mathcal{K}_j, & j \in \mathcal{J}, \\ & \overline{X}_j \in \mathcal{K}_j, & j \in \mathcal{J}^{PSD}. \end{array} \tag{17.5}$$

- **Variables** are either scalar variables, x_j for $j \in \mathcal{J}$, or 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}

$$g^{obj} = \sum_{j \in \mathcal{J}^{PSD}} \langle F_j^{obj}, X_j \rangle + \sum_{j \in \mathcal{J}} a_j^{obj} x_j + b^{obj}.$$

CBF format can represent the following 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

$$\left\{ \begin{pmatrix} p \\ q \\ x \end{pmatrix} \in \mathbb{R} \times \mathbb{R} \times \mathbb{R}^{n-2},\ 2pq \geq x^T x,\ p \geq 0,\ q \geq 0 \right\}, \text{ for } n \geq 3.$$

17.4.2 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.

Embedded hotstart-sequences

A sequence of problem instances, based on the same problem structure, is within a single file. This is facilitated via the CHANGE within the problem data information group, as a separator between the information items of each instance. The information items following a CHANGE keyword are appending to, or changing (e.g., setting coefficients back to their default value of zero), the problem data of the preceding instance.

The sequence is intended for benchmarking of hotstart capability, where the solvers can reuse their internal state and solution (subject to the achieved accuracy) as warmpoint for the succeeding instance. Whenever this feature is unsupported or undesired, the keyword CHANGE should be interpreted as the end of file.

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.

17.4.3 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 17.3. 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 17.3.

- **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 minimum sizes are given as follows.

Table 17.3: Cones available in the CBF format

Name	CBF keyword	Cone family
Free domain	`F`	linear
Positive orthant	`L+`	linear
Negative orthant	`L-`	linear
Fixpoint zero	`L=`	linear
Quadratic cone	`Q`	second-order
Rotated quadratic cone	`QR`	second-order

17.4.4 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.

OBJSENSE

Description: Define the objective sense.

`HEADER`: None

`BODY`: One line formatted as:

```
STR
```

having `MIN` indicates minimize, and `MAX` indicates maximize. Capital 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 17.3), 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 17.3), 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.

CHANGE

Start of a new instance specification based on changes to the previous. Can be interpreted as the end of file when the hotstart-sequence is unsupported or undesired.

`BODY`: None

`Header`: None

17.4.5 CBF Format Examples

Minimal Working Example

The conic optimization problem (17.6) , has three variables in a quadratic cone - first one is integer - and an affine expression in domain 0 (equality constraint).

$$\begin{array}{ll} \text{minimize} & 5.1\, x_0 \\ \text{subject to} & 6.2\, x_1 + 7.3\, x_2 - 8.4 \in \{0\} \\ & x \in \mathcal{Q}^3, \;\; x_0 \in \mathbb{Z}. \end{array} \tag{17.6}$$

Its formulation in the Conic Benchmark Format begins with the version of the CBF format used, to safeguard against later revisions.

```
VER
1
```

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 (17.7), has a semidefinite cone, a quadratic cone over unordered subindices, and two equality constraints.

$$\begin{array}{lllll}
\text{minimize} & \left\langle \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}, X_1 \right\rangle + x_1 & & \\
\text{subject to} & \left\langle \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, X_1 \right\rangle + x_1 & = & 1.0\,, \\
 & \left\langle \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, X_1 \right\rangle + x_0 + x_2 & = & 0.5\,, \\
 & x_1 \geq \sqrt{x_0^2 + x_2^2}\,, & & \\
 & X_1 \succeq \mathbf{0}\,. & &
\end{array} \tag{17.7}$$

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 1.
VER
1

# 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
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 in.

$$
\begin{array}{lllll}
\text{minimize} & \left\langle \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, X_1 \right\rangle + x_1 + x_2 + 1 & & \\
\text{subject to} & \left\langle \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, X_1 \right\rangle - x_1 - x_2 & \geq & 0.0\,, \\
 & x_1 \begin{bmatrix} 0 & 1 \\ 1 & 3 \end{bmatrix} + x_2 \begin{bmatrix} 3 & 1 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & \succeq & \mathbf{0}\,, \\
 & X_1 \succeq \mathbf{0}\,. & &
\end{array}
\tag{17.8}
$$

Its formulation in the CBF format is written in what follows

```
# File written using this version of the Conic Benchmark Format:
#     | Version 1.
VER
1

# 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:
#     | 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
```

Optimization Over a Sequence of Objectives

The linear optimization problem (17.9), is defined for a sequence of objectives such that hotstarting from one to the next might be advantages.

$$\begin{array}{lccl} \text{maximize}_k & g_k^{obj} & & \\ \text{subject to} & 50\,x_0 + 31 & \leq & 250\,, \\ & 3\,x_0 - 2x_1 & \geq & -4\,, \\ & x \in \mathbb{R}_+^2, & & \end{array} \tag{17.9}$$

given,

1. $g_0^{obj} = x_0 + 0.64x_1$.
2. $g_1^{obj} = 1.11x_0 + 0.76x_1$.
3. $g_2^{obj} = 1.11x_0 + 0.85x_1$.

Its formulation in the CBF format is reported in Listing 17.5.

Listing 17.5: Problem (17.9) in CBF format.

```
# File written using this version of the Conic Benchmark Format:
#     | Version 1.
VER
1

# The sense of the objective is:
#     | Maximize.
OBJSENSE
MAX

# Two scalar variables in this one conic domain:
#     | Two are nonnegative.
VAR
2 1
L+ 2

# Two scalar constraints with affine expressions in these two conic domains:
#     | One is in the nonpositive domain.
#     | One is in the nonnegative domain.
CON
2 2
L- 1
L+ 1

# Two coordinates in a^{obj}_j coefficients:
#     | a^{obj}[0] = 1.0
#     | a^{obj}[1] = 0.64
OBJACOORD
2
0 1.0
1 0.64

# Four coordinates in a_ij coefficients:
#     | a[0,0] = 50.0
#     | a[1,0] = 3.0
```

```
#     | and more...
ACOORD
4
0 0 50.0
1 0 3.0
0 1 31.0
1 1 -2.0

# Two coordinates in b_i coefficients:
#     | b[0] = -250.0
#     | b[1] = 4.0
BCOORD
2
0 -250.0
1 4.0

# New problem instance defined in terms of changes.
CHANGE

# Two coordinate changes in a^{obj}_j coefficients. Now it is:
#     | a^{obj}[0] = 1.11
#     | a^{obj}[1] = 0.76
OBJACOORD
2
0 1.11
1 0.76

# New problem instance defined in terms of changes.
CHANGE

# One coordinate change in a^{obj}_j coefficients. Now it is:
#     | a^{obj}[0] = 1.11
#     | a^{obj}[1] = 0.85
OBJACOORD
1
1 0.85
```

17.5 The XML (OSiL) Format

MOSEK can write data in the standard OSiL xml format. For a definition of the OSiL format please see http://www.optimizationservices.org/.

Only linear constraints (possibly with integer variables) are supported. By default output files with the extension `.xml` are written in the OSiL format.

The parameter *MSK_IPAR_WRITE_XML_MODE* controls if the linear coefficients in the A matrix are written in row or column order.

17.6 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 quadratic, 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:

- The task format *does not* support General Convex problems since these are defined by arbitrary user-defined functions.
- 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.

17.7 The JSON Format

MOSEK provides the possibility to read/write problems in valid JSON format.

> JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

The official JSON website http://www.json.org provides plenty of information along with the format definition.

MOSEK defines two JSON-like formats:

- *jtask*
- *jsol*

> **Warning:** Despite being text-based human-readable formats, *jtask* and *jsol* files will include no indentation and no new-lines, in order to keep the files as compact as possible. We therefore strongly advise to use JSON viewer tools to inspect *jtask* and *jsol* files.

17.7.1 *jtask* format

It stores a problem instance. The *jtask* format contains the same information as a *task format*.

You can read and write *jtask* files using `MSK_readdata` and `MSK_writedata` specifying the extension *.jtask*.

Even though a *jtask* file is human-readable, we do not recommend users to create it by hand, but to rely on **MOSEK**.

17.7.2 *jsol* format

It stores a problem solution. The *jsol* format contains all solutions and information items.

You can write a *jsol* file using `MSK_writejsonsol`. You **can not** read a *jsol* file into **MOSEK**.

17.7.3 A *jtask* example

In Listing 17.6 we present a file in the *jtask* format that corresponds to the sample problem from `lo1.lp`. The listing has been formatted for readability.

Listing 17.6: A formatted *jtask* file for the `lo1.lp` example.

```
{
    "$schema":"http://mosek.com/json/schema#",
    "Task/INFO":{
        "taskname":"lo1",
        "numvar":4,
        "numcon":3,
        "numcone":0,
        "numbarvar":0,
        "numanz":9,
        "numsymmat":0,
        "mosekver":[
            8,
            0,
            0,
            9
        ]
    },
    "Task/data":{
        "var":{
            "name":[
                "x1",
                "x2",
                "x3",
                "x4"
            ],
            "bk":[
                "lo",
                "ra",
                "lo",
                "lo"
            ],
            "bl":[
                0.0,
                0.0,
                0.0,
                0.0
            ],
            "bu":[
                1e+30,
                1e+1,
                1e+30,
                1e+30
            ],
            "type":[
                "cont",
                "cont",
                "cont",
                "cont"
            ]
        },
        "con":{
            "name":[
                "c1",
                "c2",
                "c3"
            ],
```

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

```
                1e+0,
                3e+0,
                1e+0,
                2e+0,
                3e+0
            ]
        }
    },
    "Task/parameters":{
        "iparam":{
            "ANA_SOL_BASIS":"ON",
            "ANA_SOL_PRINT_VIOLATED":"OFF",
            "AUTO_SORT_A_BEFORE_OPT":"OFF",
            "AUTO_UPDATE_SOL_INFO":"OFF",
            "BASIS_SOLVE_USE_PLUS_ONE":"OFF",
            "BI_CLEAN_OPTIMIZER":"OPTIMIZER_FREE",
            "BI_IGNORE_MAX_ITER":"OFF",
            "BI_IGNORE_NUM_ERROR":"OFF",
            "BI_MAX_ITERATIONS":1000000,
            "CACHE_LICENSE":"ON",
            "CHECK_CONVEXITY":"CHECK_CONVEXITY_FULL",
            "COMPRESS_STATFILE":"ON",
            "CONCURRENT_NUM_OPTIMIZERS":2,
            "CONCURRENT_PRIORITY_DUAL_SIMPLEX":2,
            "CONCURRENT_PRIORITY_FREE_SIMPLEX":3,
            "CONCURRENT_PRIORITY_INTPNT":4,
            "CONCURRENT_PRIORITY_PRIMAL_SIMPLEX":1,
            "FEASREPAIR_OPTIMIZE":"FEASREPAIR_OPTIMIZE_NONE",
            "INFEAS_GENERIC_NAMES":"OFF",
            "INFEAS_PREFER_PRIMAL":"ON",
            "INFEAS_REPORT_AUTO":"OFF",
            "INFEAS_REPORT_LEVEL":1,
            "INTPNT_BASIS":"BI_ALWAYS",
            "INTPNT_DIFF_STEP":"ON",
            "INTPNT_FACTOR_DEBUG_LVL":0,
            "INTPNT_FACTOR_METHOD":0,
            "INTPNT_HOTSTART":"INTPNT_HOTSTART_NONE",
            "INTPNT_MAX_ITERATIONS":400,
            "INTPNT_MAX_NUM_COR":-1,
            "INTPNT_MAX_NUM_REFINEMENT_STEPS":-1,
            "INTPNT_OFF_COL_TRH":40,
            "INTPNT_ORDER_METHOD":"ORDER_METHOD_FREE",
            "INTPNT_REGULARIZATION_USE":"ON",
            "INTPNT_SCALING":"SCALING_FREE",
            "INTPNT_SOLVE_FORM":"SOLVE_FREE",
            "INTPNT_STARTING_POINT":"STARTING_POINT_FREE",
            "LIC_TRH_EXPIRY_WRN":7,
            "LICENSE_DEBUG":"OFF",
            "LICENSE_PAUSE_TIME":0,
            "LICENSE_SUPPRESS_EXPIRE_WRNS":"OFF",
            "LICENSE_WAIT":"OFF",
            "LOG":10,
            "LOG_ANA_PRO":1,
            "LOG_BI":4,
            "LOG_BI_FREQ":2500,
            "LOG_CHECK_CONVEXITY":0,
            "LOG_CONCURRENT":1,
            "LOG_CUT_SECOND_OPT":1,
            "LOG_EXPAND":0,
            "LOG_FACTOR":1,
            "LOG_FEAS_REPAIR":1,
            "LOG_FILE":1,
            "LOG_HEAD":1,
```

```
            "LOG_INFEAS_ANA":1,
            "LOG_INTPNT":4,
            "LOG_MIO":4,
            "LOG_MIO_FREQ":1000,
            "LOG_OPTIMIZER":1,
            "LOG_ORDER":1,
            "LOG_PRESOLVE":1,
            "LOG_RESPONSE":0,
            "LOG_SENSITIVITY":1,
            "LOG_SENSITIVITY_OPT":0,
            "LOG_SIM":4,
            "LOG_SIM_FREQ":1000,
            "LOG_SIM_MINOR":1,
            "LOG_STORAGE":1,
            "MAX_NUM_WARNINGS":10,
            "MIO_BRANCH_DIR":"BRANCH_DIR_FREE",
            "MIO_CONSTRUCT_SOL":"OFF",
            "MIO_CUT_CLIQUE":"ON",
            "MIO_CUT_CMIR":"ON",
            "MIO_CUT_GMI":"ON",
            "MIO_CUT_KNAPSACK_COVER":"OFF",
            "MIO_HEURISTIC_LEVEL":-1,
            "MIO_MAX_NUM_BRANCHES":-1,
            "MIO_MAX_NUM_RELAXS":-1,
            "MIO_MAX_NUM_SOLUTIONS":-1,
            "MIO_MODE":"MIO_MODE_SATISFIED",
            "MIO_MT_USER_CB":"ON",
            "MIO_NODE_OPTIMIZER":"OPTIMIZER_FREE",
            "MIO_NODE_SELECTION":"MIO_NODE_SELECTION_FREE",
            "MIO_PERSPECTIVE_REFORMULATE":"ON",
            "MIO_PROBING_LEVEL":-1,
            "MIO_RINS_MAX_NODES":-1,
            "MIO_ROOT_OPTIMIZER":"OPTIMIZER_FREE",
            "MIO_ROOT_REPEAT_PRESOLVE_LEVEL":-1,
            "MT_SPINCOUNT":0,
            "NUM_THREADS":0,
            "OPF_MAX_TERMS_PER_LINE":5,
            "OPF_WRITE_HEADER":"ON",
            "OPF_WRITE_HINTS":"ON",
            "OPF_WRITE_PARAMETERS":"OFF",
            "OPF_WRITE_PROBLEM":"ON",
            "OPF_WRITE_SOL_BAS":"ON",
            "OPF_WRITE_SOL_ITG":"ON",
            "OPF_WRITE_SOL_ITR":"ON",
            "OPF_WRITE_SOLUTIONS":"OFF",
            "OPTIMIZER":"OPTIMIZER_FREE",
            "PARAM_READ_CASE_NAME":"ON",
            "PARAM_READ_IGN_ERROR":"OFF",
            "PRESOLVE_ELIMINATOR_MAX_FILL":-1,
            "PRESOLVE_ELIMINATOR_MAX_NUM_TRIES":-1,
            "PRESOLVE_LEVEL":-1,
            "PRESOLVE_LINDEP_ABS_WORK_TRH":100,
            "PRESOLVE_LINDEP_REL_WORK_TRH":100,
            "PRESOLVE_LINDEP_USE":"ON",
            "PRESOLVE_MAX_NUM_REDUCTIONS":-1,
            "PRESOLVE_USE":"PRESOLVE_MODE_FREE",
            "PRIMAL_REPAIR_OPTIMIZER":"OPTIMIZER_FREE",
            "QO_SEPARABLE_REFORMULATION":"OFF",
            "READ_DATA_COMPRESSED":"COMPRESS_FREE",
            "READ_DATA_FORMAT":"DATA_FORMAT_EXTENSION",
            "READ_DEBUG":"OFF",
            "READ_KEEP_FREE_CON":"OFF",
            "READ_LP_DROP_NEW_VARS_IN_BOU":"OFF",
```

```
            "READ_LP_QUOTED_NAMES":"ON",
            "READ_MPS_FORMAT":"MPS_FORMAT_FREE",
            "READ_MPS_WIDTH":1024,
            "READ_TASK_IGNORE_PARAM":"OFF",
            "SENSITIVITY_ALL":"OFF",
            "SENSITIVITY_OPTIMIZER":"OPTIMIZER_FREE_SIMPLEX",
            "SENSITIVITY_TYPE":"SENSITIVITY_TYPE_BASIS",
            "SIM_BASIS_FACTOR_USE":"ON",
            "SIM_DEGEN":"SIM_DEGEN_FREE",
            "SIM_DUAL_CRASH":90,
            "SIM_DUAL_PHASEONE_METHOD":0,
            "SIM_DUAL_RESTRICT_SELECTION":50,
            "SIM_DUAL_SELECTION":"SIM_SELECTION_FREE",
            "SIM_EXPLOIT_DUPVEC":"SIM_EXPLOIT_DUPVEC_OFF",
            "SIM_HOTSTART":"SIM_HOTSTART_FREE",
            "SIM_HOTSTART_LU":"ON",
            "SIM_INTEGER":0,
            "SIM_MAX_ITERATIONS":10000000,
            "SIM_MAX_NUM_SETBACKS":250,
            "SIM_NON_SINGULAR":"ON",
            "SIM_PRIMAL_CRASH":90,
            "SIM_PRIMAL_PHASEONE_METHOD":0,
            "SIM_PRIMAL_RESTRICT_SELECTION":50,
            "SIM_PRIMAL_SELECTION":"SIM_SELECTION_FREE",
            "SIM_REFACTOR_FREQ":0,
            "SIM_REFORMULATION":"SIM_REFORMULATION_OFF",
            "SIM_SAVE_LU":"OFF",
            "SIM_SCALING":"SCALING_FREE",
            "SIM_SCALING_METHOD":"SCALING_METHOD_POW2",
            "SIM_SOLVE_FORM":"SOLVE_FREE",
            "SIM_STABILITY_PRIORITY":50,
            "SIM_SWITCH_OPTIMIZER":"OFF",
            "SOL_FILTER_KEEP_BASIC":"OFF",
            "SOL_FILTER_KEEP_RANGED":"OFF",
            "SOL_READ_NAME_WIDTH":-1,
            "SOL_READ_WIDTH":1024,
            "SOLUTION_CALLBACK":"OFF",
            "TIMING_LEVEL":1,
            "WRITE_BAS_CONSTRAINTS":"ON",
            "WRITE_BAS_HEAD":"ON",
            "WRITE_BAS_VARIABLES":"ON",
            "WRITE_DATA_COMPRESSED":0,
            "WRITE_DATA_FORMAT":"DATA_FORMAT_EXTENSION",
            "WRITE_DATA_PARAM":"OFF",
            "WRITE_FREE_CON":"OFF",
            "WRITE_GENERIC_NAMES":"OFF",
            "WRITE_GENERIC_NAMES_IO":1,
            "WRITE_IGNORE_INCOMPATIBLE_CONIC_ITEMS":"OFF",
            "WRITE_IGNORE_INCOMPATIBLE_ITEMS":"OFF",
            "WRITE_IGNORE_INCOMPATIBLE_NL_ITEMS":"OFF",
            "WRITE_IGNORE_INCOMPATIBLE_PSD_ITEMS":"OFF",
            "WRITE_INT_CONSTRAINTS":"ON",
            "WRITE_INT_HEAD":"ON",
            "WRITE_INT_VARIABLES":"ON",
            "WRITE_LP_FULL_OBJ":"ON",
            "WRITE_LP_LINE_WIDTH":80,
            "WRITE_LP_QUOTED_NAMES":"ON",
            "WRITE_LP_STRICT_FORMAT":"OFF",
            "WRITE_LP_TERMS_PER_LINE":10,
            "WRITE_MPS_FORMAT":"MPS_FORMAT_FREE",
            "WRITE_MPS_INT":"ON",
            "WRITE_PRECISION":15,
            "WRITE_SOL_BARVARIABLES":"ON",
```

```
        "WRITE_SOL_CONSTRAINTS":"ON",
        "WRITE_SOL_HEAD":"ON",
        "WRITE_SOL_IGNORE_INVALID_NAMES":"OFF",
        "WRITE_SOL_VARIABLES":"ON",
        "WRITE_TASK_INC_SOL":"ON",
        "WRITE_XML_MODE":"WRITE_XML_MODE_ROW"
    },
    "dparam":{
        "ANA_SOL_INFEAS_TOL":1e-6,
        "BASIS_REL_TOL_S":1e-12,
        "BASIS_TOL_S":1e-6,
        "BASIS_TOL_X":1e-6,
        "CHECK_CONVEXITY_REL_TOL":1e-10,
        "DATA_TOL_AIJ":1e-12,
        "DATA_TOL_AIJ_HUGE":1e+20,
        "DATA_TOL_AIJ_LARGE":1e+10,
        "DATA_TOL_BOUND_INF":1e+16,
        "DATA_TOL_BOUND_WRN":1e+8,
        "DATA_TOL_C_HUGE":1e+16,
        "DATA_TOL_CJ_LARGE":1e+8,
        "DATA_TOL_QIJ":1e-16,
        "DATA_TOL_X":1e-8,
        "FEASREPAIR_TOL":1e-10,
        "INTPNT_CO_TOL_DFEAS":1e-8,
        "INTPNT_CO_TOL_INFEAS":1e-10,
        "INTPNT_CO_TOL_MU_RED":1e-8,
        "INTPNT_CO_TOL_NEAR_REL":1e+3,
        "INTPNT_CO_TOL_PFEAS":1e-8,
        "INTPNT_CO_TOL_REL_GAP":1e-7,
        "INTPNT_NL_MERIT_BAL":1e-4,
        "INTPNT_NL_TOL_DFEAS":1e-8,
        "INTPNT_NL_TOL_MU_RED":1e-12,
        "INTPNT_NL_TOL_NEAR_REL":1e+3,
        "INTPNT_NL_TOL_PFEAS":1e-8,
        "INTPNT_NL_TOL_REL_GAP":1e-6,
        "INTPNT_NL_TOL_REL_STEP":9.95e-1,
        "INTPNT_QO_TOL_DFEAS":1e-8,
        "INTPNT_QO_TOL_INFEAS":1e-10,
        "INTPNT_QO_TOL_MU_RED":1e-8,
        "INTPNT_QO_TOL_NEAR_REL":1e+3,
        "INTPNT_QO_TOL_PFEAS":1e-8,
        "INTPNT_QO_TOL_REL_GAP":1e-8,
        "INTPNT_TOL_DFEAS":1e-8,
        "INTPNT_TOL_DSAFE":1e+0,
        "INTPNT_TOL_INFEAS":1e-10,
        "INTPNT_TOL_MU_RED":1e-16,
        "INTPNT_TOL_PATH":1e-8,
        "INTPNT_TOL_PFEAS":1e-8,
        "INTPNT_TOL_PSAFE":1e+0,
        "INTPNT_TOL_REL_GAP":1e-8,
        "INTPNT_TOL_REL_STEP":9.999e-1,
        "INTPNT_TOL_STEP_SIZE":1e-6,
        "LOWER_OBJ_CUT":-1e+30,
        "LOWER_OBJ_CUT_FINITE_TRH":-5e+29,
        "MIO_DISABLE_TERM_TIME":-1e+0,
        "MIO_MAX_TIME":-1e+0,
        "MIO_MAX_TIME_APRX_OPT":6e+1,
        "MIO_NEAR_TOL_ABS_GAP":0.0,
        "MIO_NEAR_TOL_REL_GAP":1e-3,
        "MIO_REL_GAP_CONST":1e-10,
        "MIO_TOL_ABS_GAP":0.0,
        "MIO_TOL_ABS_RELAX_INT":1e-5,
        "MIO_TOL_FEAS":1e-6,
```

```
            "MIO_TOL_REL_DUAL_BOUND_IMPROVEMENT":0.0,
            "MIO_TOL_REL_GAP":1e-4,
            "MIO_TOL_X":1e-6,
            "OPTIMIZER_MAX_TIME":-1e+0,
            "PRESOLVE_TOL_ABS_LINDEP":1e-6,
            "PRESOLVE_TOL_AIJ":1e-12,
            "PRESOLVE_TOL_REL_LINDEP":1e-10,
            "PRESOLVE_TOL_S":1e-8,
            "PRESOLVE_TOL_X":1e-8,
            "QCQO_REFORMULATE_REL_DROP_TOL":1e-15,
            "SEMIDEFINITE_TOL_APPROX":1e-10,
            "SIM_LU_TOL_REL_PIV":1e-2,
            "SIMPLEX_ABS_TOL_PIV":1e-7,
            "UPPER_OBJ_CUT":1e+30,
            "UPPER_OBJ_CUT_FINITE_TRH":5e+29
        },
        "sparam":{
            "BAS_SOL_FILE_NAME":"",
            "DATA_FILE_NAME":"examples/tools/data/lo1.mps",
            "DEBUG_FILE_NAME":"",
            "INT_SOL_FILE_NAME":"",
            "ITR_SOL_FILE_NAME":"",
            "MIO_DEBUG_STRING":"",
            "PARAM_COMMENT_SIGN":"%%",
            "PARAM_READ_FILE_NAME":"",
            "PARAM_WRITE_FILE_NAME":"",
            "READ_MPS_BOU_NAME":"",
            "READ_MPS_OBJ_NAME":"",
            "READ_MPS_RAN_NAME":"",
            "READ_MPS_RHS_NAME":"",
            "SENSITIVITY_FILE_NAME":"",
            "SENSITIVITY_RES_FILE_NAME":"",
            "SOL_FILTER_XC_LOW":"",
            "SOL_FILTER_XC_UPR":"",
            "SOL_FILTER_XX_LOW":"",
            "SOL_FILTER_XX_UPR":"",
            "STAT_FILE_NAME":"",
            "STAT_KEY":"",
            "STAT_NAME":"",
            "WRITE_LP_GEN_VAR_NAME":"XMSKGEN"
        }
    }
}
```


17.8 The Solution File Format

MOSEK provides several solution files depending on the problem type and the optimizer used:

- *basis solution file* (extension `.bas`) if the problem is optimized using the simplex optimizer or basis identification is performed,
- *interior solution file* (extension `.sol`) if a problem is optimized using the interior-point optimizer and no basis identification is required,
- *integer solution file* (extension `.int`) if the problem contains integer constrained variables.

All solution files have the format:

```
NAME                : <problem name>
PROBLEM STATUS      : <status of the problem>
SOLUTION STATUS     : <status of the solution>
```

```
OBJECTIVE NAME      : <name of the objective function>
PRIMAL OBJECTIVE    : <primal objective value corresponding to the solution>
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>
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>
```

In the example the fields `?` and `<>` will be filled with problem and solution specific information. As can be observed a solution report consists of three sections, i.e.

- `HEADER` In this section, first the name of the problem is listed and afterwards the problem and solution status are shown. Next the primal and dual objective values are displayed.
- `CONSTRAINTS` For each constraint i of the form

$$l_i^c \leq \sum_{j=1}^{n} a_{ij} x_j \leq u_i^c, \tag{17.10}$$

  the following information is listed:

  - `INDEX`: A sequential index assigned to the constraint by **MOSEK**
  - `NAME`: The name of the constraint assigned by the user.
  - `AT`: The status of the constraint. In Table 17.4 the possible values of the status keys and their interpretation are shown.

Table 17.4: Status keys.

Status key	Interpretation
`UN`	Unknown status
`BS`	Is basic
`SB`	Is superbasic
`LL`	Is at the lower limit (bound)
`UL`	Is at the upper limit (bound)
`EQ`	Lower limit is identical to upper limit
`**`	Is infeasible i.e. the lower limit is greater than the upper limit.

  - `ACTIVITY`: the quantity $\sum_{j=1}^{n} a_{ij} x_j^*$, where x^* is the value of the primal solution.
  - `LOWER LIMIT`: the quantity l_i^c (see (17.10).)
  - `UPPER LIMIT`: the quantity u_i^c (see (17.10).)
  - `DUAL LOWER`: the dual multiplier corresponding to the lower limit on the constraint.
  - `DUAL UPPER`: the dual multiplier corresponding to the upper limit on the constraint.
- `VARIABLES` The last section of the solution report lists information about the variables. This information has a similar interpretation as for the constraints. However, the column with the header `CONIC DUAL` is included for problems having one or more conic constraints. This column shows the dual variables corresponding to the conic constraints.

Example: `lo1.sol`

In Listing 17.7 we show the solution file for the `lo1.opf` problem.

Listing 17.7: An example of `.sol` file.

```
NAME                : 
PROBLEM STATUS      : PRIMAL_AND_DUAL_FEASIBLE
SOLUTION STATUS     : OPTIMAL
OBJECTIVE NAME      : obj
PRIMAL OBJECTIVE    : 8.33333333e+01
DUAL OBJECTIVE      : 8.33333332e+01

CONSTRAINTS
INDEX      NAME           AT ACTIVITY                 LOWER LIMIT        UPPER LIMIT        ␣
↪DUAL LOWER                DUAL UPPER
0          c1             EQ 3.00000000000000e+01     3.00000000e+01     3.00000000e+01     -0.
↪00000000000000e+00    -2.49999999741654e+00
1          c2             SB 5.33333333049188e+01     1.50000000e+01     NONE               2.
↪09157603759397e-10     -0.00000000000000e+00
2          c3             UL 2.49999999842049e+01     NONE               2.50000000e+01     -0.
↪00000000000000e+00    -3.33333332895110e-01

VARIABLES
INDEX      NAME           AT ACTIVITY                 LOWER LIMIT        UPPER LIMIT        ␣
↪DUAL LOWER                DUAL UPPER
0          x1             LL 1.67020427073508e-09     0.00000000e+00     NONE               -4.
↪49999999528055e+00    -0.00000000000000e+00
1          x2             LL 2.93510446280504e-09     0.00000000e+00     1.00000000e+01     -2.
↪16666666494916e+00    6.20863861687316e-10
2          x3             SB 1.49999999899425e+01     0.00000000e+00     NONE               -8.
↪79123177454657e-10    -0.00000000000000e+00
3          x4             SB 8.33333332273116e+00     0.00000000e+00     NONE               -1.
↪69795978899185e-09    -0.00000000000000e+00
```

CHAPTER

EIGHTEEN

LIST OF EXAMPLES

List of examples shipped in the distribution of Optimizer API for C:

Table 18.1: List of distributed examples

File	Description
`blas_lapack.c`	Demonstrates the **MOSEK** interface to BLAS/LAPACK linear algebra routines
`callback.c`	An example of data/progress callback
`case_portfolio_1.c`	Implements a basic portfolio optimization model
`case_portfolio_2.c`	Implements a basic portfolio optimization model with efficient frontier
`case_portfolio_3.c`	Implements a basic portfolio optimization model with market impact costs
`cqo1.c`	A simple conic quadratic problem
`dgopt.c`	Dual geometric optimization library (DGopt)
`dgopt.h`	Header file for DGopt
`errorreporting.c`	Demonstrates how error reporting can be customized
`expopt.c`	Exponential optimization library (EXPopt)
`expopt.h`	Header file for EXPopt
`feasrepairex1.c`	A simple example of how to repair an infeasible problem
`lo1.c`	A simple linear problem
`lo2.c`	A simple linear problem
`milo1.c`	A simple mixed-integer linear problem
`mioinitsol.c`	A simple mixed-integer linear problem with an initial guess
`mskdgopt.c`	Dual geometric optimization command-line solver (mskdgopt)
`mskexpopt.c`	Exponential optimization command-line solver (mskexpopt)
`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
`parameters.c`	Shows how to set optimizer parameters and read information items
`production.c`	Demonstrate how to modify and re-optimize a linear problem
`qcqo1.c`	A simple quadratically constrained quadratic problem
`qo1.c`	A simple quadratic problem
`response.c`	Demonstrates proper response handling
`scopt-ext.c`	Separable convex optimization library (SCopt)
`scopt-ext.h`	Header file for SCopt
`sdo1.c`	A simple semidefinite optimization problem
`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

Continued on next page

Table 18.1 – continued from previous page

File	Description
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
tstexpopt.c	A small exponential optimization example
tstscopt.c	A small separable convex optimization example
unicode.c	Demonstrates string conversion to Unicode

Additional examples can be found on the **MOSEK** website and in other **MOSEK** publications.

CHAPTER

NINETEEN

INTERFACE CHANGES

The section show interface-specific changes to the **MOSEK** Optimizer API for C in version 8. See the release notes for general changes and new features of the **MOSEK** Optimization Suite.

19.1 Functions

Added

Changed

Removed

- MSK_getglbdllname
- MSK_init
- MSK_putdllpath
- MSK_putkeepdlls
- MSK_set_stream
- MSK_strdupdbgenv
- MSK_strdupenv
- MSK_getdbi
- MSK_getdcni
- MSK_getdeqi
- MSK_getinti
- MSK_getnumqconknz64
- MSK_getpbi
- MSK_getpcni
- MSK_getpeqi
- MSK_getqobj64
- MSK_getsolutionincallback
- MSK_getsolutioninf
- MSK_getvarbranchdir
- MSK_getvarbranchorder
- MSK_getvarbranchpri

- MSK_optimizeconcurrent
- MSK_progress
- MSK_putvarbranchorder
- MSK_readbranchpriorities
- MSK_relaxprimal
- MSK_set_stream
- MSK_writebranchpriorities

19.2 Parameters

Added

- *MSK_DPAR_DATA_SYM_MAT_TOL*
- *MSK_DPAR_DATA_SYM_MAT_TOL_HUGE*
- *MSK_DPAR_DATA_SYM_MAT_TOL_LARGE*
- *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_SEMIDEFINITE_TOL_APPROX*
- *MSK_IPAR_INTPNT_MULTI_THREAD*
- *MSK_IPAR_LICENSE_TRH_EXPIRY_WRN*
- *MSK_IPAR_LOG_ANA_PRO*
- *MSK_IPAR_MIO_CUT_CLIQUE*
- *MSK_IPAR_MIO_CUT_GMI*
- *MSK_IPAR_MIO_CUT_IMPLIED_BOUND*
- *MSK_IPAR_MIO_CUT_KNAPSACK_COVER*
- *MSK_IPAR_MIO_CUT_SELECTION_LEVEL*
- *MSK_IPAR_MIO_PERSPECTIVE_REFORMULATE*
- *MSK_IPAR_MIO_ROOT_REPEAT_PRESOLVE_LEVEL*
- *MSK_IPAR_MIO_VB_DETECTION_LEVEL*
- *MSK_IPAR_PRESOLVE_ELIMINATOR_MAX_FILL*
- *MSK_IPAR_REMOVE_UNUSED_SOLUTIONS*
- *MSK_IPAR_WRITE_LP_FULL_OBJ*
- *MSK_IPAR_WRITE_MPS_FORMAT*
- *MSK_SPAR_REMOTE_ACCESS_TOKEN*

Removed

- MSK_DPAR_FEASREPAIR_TOL
- MSK_DPAR_MIO_HEURISTIC_TIME
- MSK_DPAR_MIO_MAX_TIME_APRX_OPT
- MSK_DPAR_MIO_REL_ADD_CUT_LIMITED
- MSK_DPAR_MIO_TOL_MAX_CUT_FRAC_RHS
- MSK_DPAR_MIO_TOL_MIN_CUT_FRAC_RHS
- MSK_DPAR_MIO_TOL_REL_RELAX_INT
- MSK_DPAR_MIO_TOL_X
- MSK_DPAR_NONCONVEX_TOL_FEAS
- MSK_DPAR_NONCONVEX_TOL_OPT
- MSK_IPAR_ALLOC_ADD_QNZ
- MSK_IPAR_CONCURRENT_NUM_OPTIMIZERS
- MSK_IPAR_CONCURRENT_PRIORITY_DUAL_SIMPLEX
- MSK_IPAR_CONCURRENT_PRIORITY_FREE_SIMPLEX
- MSK_IPAR_CONCURRENT_PRIORITY_INTPNT
- MSK_IPAR_CONCURRENT_PRIORITY_PRIMAL_SIMPLEX
- MSK_IPAR_FEASREPAIR_OPTIMIZE
- MSK_IPAR_INTPNT_FACTOR_DEBUG_LVL
- MSK_IPAR_INTPNT_FACTOR_METHOD
- MSK_IPAR_LIC_TRH_EXPIRY_WRN
- MSK_IPAR_LOG_CONCURRENT
- MSK_IPAR_LOG_FACTOR
- MSK_IPAR_LOG_HEAD
- MSK_IPAR_LOG_NONCONVEX
- MSK_IPAR_LOG_OPTIMIZER
- MSK_IPAR_LOG_PARAM
- MSK_IPAR_LOG_SIM_NETWORK_FREQ
- MSK_IPAR_MIO_BRANCH_PRIORITIES_USE
- MSK_IPAR_MIO_CONT_SOL
- MSK_IPAR_MIO_CUT_CG
- MSK_IPAR_MIO_CUT_LEVEL_ROOT
- MSK_IPAR_MIO_CUT_LEVEL_TREE
- MSK_IPAR_MIO_FEASPUMP_LEVEL
- MSK_IPAR_MIO_HOTSTART
- MSK_IPAR_MIO_KEEP_BASIS
- MSK_IPAR_MIO_LOCAL_BRANCH_NUMBER
- MSK_IPAR_MIO_OPTIMIZER_MODE

- MSK_IPAR_MIO_PRESOLVE_AGGREGATE
- MSK_IPAR_MIO_PRESOLVE_PROBING
- MSK_IPAR_MIO_PRESOLVE_USE
- MSK_IPAR_MIO_STRONG_BRANCH
- MSK_IPAR_MIO_USE_MULTITHREADED_OPTIMIZER
- MSK_IPAR_NONCONVEX_MAX_ITERATIONS
- MSK_IPAR_PRESOLVE_ELIM_FILL
- MSK_IPAR_PRESOLVE_ELIMINATOR_USE
- MSK_IPAR_QO_SEPARABLE_REFORMULATION
- MSK_IPAR_READ_ANZ
- MSK_IPAR_READ_CON
- MSK_IPAR_READ_CONE
- MSK_IPAR_READ_MPS_KEEP_INT
- MSK_IPAR_READ_MPS_OBJ_SENSE
- MSK_IPAR_READ_MPS_RELAX
- MSK_IPAR_READ_QNZ
- MSK_IPAR_READ_VAR
- MSK_IPAR_SIM_INTEGER
- MSK_IPAR_WARNING_LEVEL
- MSK_IPAR_WRITE_IGNORE_INCOMPATIBLE_CONIC_ITEMS
- MSK_IPAR_WRITE_IGNORE_INCOMPATIBLE_NL_ITEMS
- MSK_IPAR_WRITE_IGNORE_INCOMPATIBLE_PSD_ITEMS
- MSK_SPAR_FEASREPAIR_NAME_PREFIX
- MSK_SPAR_FEASREPAIR_NAME_SEPARATOR
- MSK_SPAR_FEASREPAIR_NAME_WSUMVIOL

19.3 Constants

Added

- *MSK_BRANCH_DIR_FAR*
- *MSK_BRANCH_DIR_GUIDED*
- *MSK_BRANCH_DIR_NEAR*
- *MSK_BRANCH_DIR_PSEUDOCOST*
- *MSK_BRANCH_DIR_ROOT_LP*
- *MSK_CALLBACK_BEGIN_ROOT_CUTGEN*
- *MSK_CALLBACK_BEGIN_TO_CONIC*
- *MSK_CALLBACK_END_ROOT_CUTGEN*
- *MSK_CALLBACK_END_TO_CONIC*

- *MSK_CALLBACK_IM_ROOT_CUTGEN*
- *MSK_CALLBACK_SOLVING_REMOTE*
- *MSK_DATA_FORMAT_JSON_TASK*
- *MSK_DINF_MIO_CLIQUE_SEPARATION_TIME*
- *MSK_DINF_MIO_CMIR_SEPARATION_TIME*
- *MSK_DINF_MIO_GMI_SEPARATION_TIME*
- *MSK_DINF_MIO_IMPLIED_BOUND_TIME*
- *MSK_DINF_MIO_KNAPSACK_COVER_SEPARATION_TIME*
- *MSK_DINF_QCQO_REFORMULATE_MAX_PERTURBATION*
- *MSK_DINF_QCQO_REFORMULATE_WORST_CHOLESKY_COLUMN_SCALING*
- *MSK_DINF_QCQO_REFORMULATE_WORST_CHOLESKY_DIAG_SCALING*
- *MSK_DINF_SOL_BAS_NRM_BARX*
- *MSK_DINF_SOL_BAS_NRM_SLC*
- *MSK_DINF_SOL_BAS_NRM_SLX*
- *MSK_DINF_SOL_BAS_NRM_SUC*
- *MSK_DINF_SOL_BAS_NRM_SUX*
- *MSK_DINF_SOL_BAS_NRM_XC*
- *MSK_DINF_SOL_BAS_NRM_XX*
- *MSK_DINF_SOL_BAS_NRM_Y*
- *MSK_DINF_SOL_ITG_NRM_BARX*
- *MSK_DINF_SOL_ITG_NRM_XC*
- *MSK_DINF_SOL_ITG_NRM_XX*
- *MSK_DINF_SOL_ITR_NRM_BARS*
- *MSK_DINF_SOL_ITR_NRM_BARX*
- *MSK_DINF_SOL_ITR_NRM_SLC*
- *MSK_DINF_SOL_ITR_NRM_SLX*
- *MSK_DINF_SOL_ITR_NRM_SNX*
- *MSK_DINF_SOL_ITR_NRM_SUC*
- *MSK_DINF_SOL_ITR_NRM_SUX*
- *MSK_DINF_SOL_ITR_NRM_XC*
- *MSK_DINF_SOL_ITR_NRM_XX*
- *MSK_DINF_SOL_ITR_NRM_Y*
- *MSK_DINF_TO_CONIC_TIME*
- *MSK_IINF_MIO_ABSGAP_SATISFIED*
- *MSK_IINF_MIO_CLIQUE_TABLE_SIZE*
- *MSK_IINF_MIO_NEAR_ABSGAP_SATISFIED*
- *MSK_IINF_MIO_NEAR_RELGAP_SATISFIED*
- *MSK_IINF_MIO_NODE_DEPTH*
- *MSK_IINF_MIO_NUM_CMIR_CUTS*

- *MSK_IINF_MIO_NUM_IMPLIED_BOUND_CUTS*
- *MSK_IINF_MIO_NUM_KNAPSACK_COVER_CUTS*
- *MSK_IINF_MIO_NUM_REPEATED_PRESOLVE*
- *MSK_IINF_MIO_PRESOLVED_NUMBIN*
- *MSK_IINF_MIO_PRESOLVED_NUMCON*
- *MSK_IINF_MIO_PRESOLVED_NUMCONT*
- *MSK_IINF_MIO_PRESOLVED_NUMINT*
- *MSK_IINF_MIO_PRESOLVED_NUMVAR*
- *MSK_IINF_MIO_RELGAP_SATISFIED*
- *MSK_LIINF_MIO_PRESOLVED_ANZ*
- *MSK_LIINF_MIO_SIM_MAXITER_SETBACKS*
- *MSK_MPS_FORMAT_CPLEX*
- *MSK_SOL_STA_DUAL_ILLPOSED_CER*
- *MSK_SOL_STA_PRIM_ILLPOSED_CER*

Changed

- *MSK_SOL_STA_INTEGER_OPTIMAL*
- *MSK_SOL_STA_NEAR_DUAL_FEAS*
- *MSK_SOL_STA_NEAR_DUAL_INFEAS_CER*
- *MSK_SOL_STA_NEAR_INTEGER_OPTIMAL*
- *MSK_SOL_STA_NEAR_OPTIMAL*
- *MSK_SOL_STA_NEAR_PRIM_AND_DUAL_FEAS*
- *MSK_SOL_STA_NEAR_PRIM_FEAS*
- *MSK_SOL_STA_NEAR_PRIM_INFEAS_CER*
- *MSK_LICENSE_BUFFER_LENGTH*

Removed

- MSK_CALLBACKCODE_BEGIN_CONCURRENT
- MSK_CALLBACKCODE_BEGIN_NETWORK_DUAL_SIMPLEX
- MSK_CALLBACKCODE_BEGIN_NETWORK_PRIMAL_SIMPLEX
- MSK_CALLBACKCODE_BEGIN_NETWORK_SIMPLEX
- MSK_CALLBACKCODE_BEGIN_NONCONVEX
- MSK_CALLBACKCODE_BEGIN_PRIMAL_DUAL_SIMPLEX
- MSK_CALLBACKCODE_BEGIN_PRIMAL_DUAL_SIMPLEX_BI
- MSK_CALLBACKCODE_BEGIN_SIMPLEX_NETWORK_DETECT
- MSK_CALLBACKCODE_END_CONCURRENT
- MSK_CALLBACKCODE_END_NETWORK_DUAL_SIMPLEX
- MSK_CALLBACKCODE_END_NETWORK_PRIMAL_SIMPLEX

- MSK_CALLBACKCODE_END_NETWORK_SIMPLEX
- MSK_CALLBACKCODE_END_NONCONVEX
- MSK_CALLBACKCODE_END_PRIMAL_DUAL_SIMPLEX
- MSK_CALLBACKCODE_END_PRIMAL_DUAL_SIMPLEX_BI
- MSK_CALLBACKCODE_END_SIMPLEX_NETWORK_DETECT
- MSK_CALLBACKCODE_IM_MIO_PRESOLVE
- MSK_CALLBACKCODE_IM_NETWORK_DUAL_SIMPLEX
- MSK_CALLBACKCODE_IM_NETWORK_PRIMAL_SIMPLEX
- MSK_CALLBACKCODE_IM_NONCONVEX
- MSK_CALLBACKCODE_IM_PRIMAL_DUAL_SIMPLEX
- MSK_CALLBACKCODE_NONCOVEX
- MSK_CALLBACKCODE_UPDATE_NETWORK_DUAL_SIMPLEX
- MSK_CALLBACKCODE_UPDATE_NETWORK_PRIMAL_SIMPLEX
- MSK_CALLBACKCODE_UPDATE_NONCONVEX
- MSK_CALLBACKCODE_UPDATE_PRIMAL_DUAL_SIMPLEX
- MSK_CALLBACKCODE_UPDATE_PRIMAL_DUAL_SIMPLEX_BI
- MSK_DINFITEM_BI_CLEAN_PRIMAL_DUAL_TIME
- MSK_DINFITEM_CONCURRENT_TIME
- MSK_DINFITEM_MIO_CG_SEPERATION_TIME
- MSK_DINFITEM_MIO_CMIR_SEPERATION_TIME
- MSK_DINFITEM_SIM_NETWORK_DUAL_TIME
- MSK_DINFITEM_SIM_NETWORK_PRIMAL_TIME
- MSK_DINFITEM_SIM_NETWORK_TIME
- MSK_DINFITEM_SIM_PRIMAL_DUAL_TIME
- MSK_FEATURE_PTOM
- MSK_FEATURE_PTOX
- MSK_IINFITEM_CONCURRENT_FASTEST_OPTIMIZER
- MSK_IINFITEM_MIO_NUM_BASIS_CUTS
- MSK_IINFITEM_MIO_NUM_CARDGUB_CUTS
- MSK_IINFITEM_MIO_NUM_COEF_REDC_CUTS
- MSK_IINFITEM_MIO_NUM_CONTRA_CUTS
- MSK_IINFITEM_MIO_NUM_DISAGG_CUTS
- MSK_IINFITEM_MIO_NUM_FLOW_COVER_CUTS
- MSK_IINFITEM_MIO_NUM_GCD_CUTS
- MSK_IINFITEM_MIO_NUM_GUB_COVER_CUTS
- MSK_IINFITEM_MIO_NUM_KNAPSUR_COVER_CUTS
- MSK_IINFITEM_MIO_NUM_LATTICE_CUTS
- MSK_IINFITEM_MIO_NUM_LIFT_CUTS
- MSK_IINFITEM_MIO_NUM_OBJ_CUTS

- MSK_IINFITEM_MIO_NUM_PLAN_LOC_CUTS
- MSK_IINFITEM_SIM_NETWORK_DUAL_DEG_ITER
- MSK_IINFITEM_SIM_NETWORK_DUAL_HOTSTART
- MSK_IINFITEM_SIM_NETWORK_DUAL_HOTSTART_LU
- MSK_IINFITEM_SIM_NETWORK_DUAL_INF_ITER
- MSK_IINFITEM_SIM_NETWORK_DUAL_ITER
- MSK_IINFITEM_SIM_NETWORK_PRIMAL_DEG_ITER
- MSK_IINFITEM_SIM_NETWORK_PRIMAL_HOTSTART
- MSK_IINFITEM_SIM_NETWORK_PRIMAL_HOTSTART_LU
- MSK_IINFITEM_SIM_NETWORK_PRIMAL_INF_ITER
- MSK_IINFITEM_SIM_NETWORK_PRIMAL_ITER
- MSK_IINFITEM_SIM_PRIMAL_DUAL_DEG_ITER
- MSK_IINFITEM_SIM_PRIMAL_DUAL_HOTSTART
- MSK_IINFITEM_SIM_PRIMAL_DUAL_HOTSTART_LU
- MSK_IINFITEM_SIM_PRIMAL_DUAL_INF_ITER
- MSK_IINFITEM_SIM_PRIMAL_DUAL_ITER
- MSK_IINFITEM_SOL_INT_PROSTA
- MSK_IINFITEM_SOL_INT_SOLSTA
- MSK_IINFITEM_STO_NUM_A_CACHE_FLUSHES
- MSK_IINFITEM_STO_NUM_A_TRANSPOSES
- MSK_LIINFITEM_BI_CLEAN_PRIMAL_DUAL_DEG_ITER
- MSK_LIINFITEM_BI_CLEAN_PRIMAL_DUAL_ITER
- MSK_LIINFITEM_BI_CLEAN_PRIMAL_DUAL_SUB_ITER
- MSK_MIOMODE_LAZY
- MSK_OPTIMIZERTYPE_CONCURRENT
- MSK_OPTIMIZERTYPE_MIXED_INT_CONIC
- MSK_OPTIMIZERTYPE_NETWORK_PRIMAL_SIMPLEX
- MSK_OPTIMIZERTYPE_NONCONVEX
- MSK_OPTIMIZERTYPE_PRIMAL_DUAL_SIMPLEX

19.4 Response Codes

Added

- *MSK_RES_ERR_CBF_DUPLICATE_PSDVAR*
- *MSK_RES_ERR_CBF_INVALID_PSDVAR_DIMENSION*
- *MSK_RES_ERR_CBF_TOO_FEW_PSDVAR*
- *MSK_RES_ERR_DUPLICATE_AIJ*
- *MSK_RES_ERR_FINAL_SOLUTION*

- *MSK_RES_ERR_JSON_DATA*
- *MSK_RES_ERR_JSON_FORMAT*
- *MSK_RES_ERR_JSON_MISSING_DATA*
- *MSK_RES_ERR_JSON_NUMBER_OVERFLOW*
- *MSK_RES_ERR_JSON_STRING*
- *MSK_RES_ERR_JSON_SYNTAX*
- *MSK_RES_ERR_LAU_INVALID_LOWER_TRIANGULAR_MATRIX*
- *MSK_RES_ERR_LAU_INVALID_SPARSE_SYMMETRIC_MATRIX*
- *MSK_RES_ERR_LAU_NOT_POSITIVE_DEFINITE*
- *MSK_RES_ERR_MIXED_CONIC_AND_NL*
- *MSK_RES_ERR_SERVER_CONNECT*
- *MSK_RES_ERR_SERVER_PROTOCOL*
- *MSK_RES_ERR_SERVER_STATUS*
- *MSK_RES_ERR_SERVER_TOKEN*
- *MSK_RES_ERR_SYM_MAT_HUGE*
- *MSK_RES_ERR_SYM_MAT_INVALID*
- *MSK_RES_ERR_TASK_WRITE*
- *MSK_RES_ERR_TOCONIC_CONSTR_NOT_CONIC*
- *MSK_RES_ERR_TOCONIC_CONSTR_Q_NOT_PSD*
- *MSK_RES_ERR_TOCONIC_CONSTRAINT_FX*
- *MSK_RES_ERR_TOCONIC_CONSTRAINT_RA*
- *MSK_RES_ERR_TOCONIC_OBJECTIVE_NOT_PSD*
- *MSK_RES_WRN_SYM_MAT_LARGE*

Removed

- MSK_RES_ERR_AD_INVALID_OPERAND
- MSK_RES_ERR_AD_INVALID_OPERATOR
- MSK_RES_ERR_AD_MISSING_OPERAND
- MSK_RES_ERR_AD_MISSING_RETURN
- MSK_RES_ERR_CONCURRENT_OPTIMIZER
- MSK_RES_ERR_INV_CONIC_PROBLEM
- MSK_RES_ERR_INVALID_BRANCH_DIRECTION
- MSK_RES_ERR_INVALID_BRANCH_PRIORITY
- MSK_RES_ERR_INVALID_NETWORK_PROBLEM
- MSK_RES_ERR_MBT_INCOMPATIBLE
- MSK_RES_ERR_MBT_INVALID
- MSK_RES_ERR_MIO_NOT_LOADED
- MSK_RES_ERR_MIXED_PROBLEM
- MSK_RES_ERR_NO_DUAL_INFO_FOR_ITG_SOL

- MSK_RES_ERR_ORD_INVALID
- MSK_RES_ERR_ORD_INVALID_BRANCH_DIR
- MSK_RES_ERR_TOCONIC_CONVERSION_FAIL
- MSK_RES_ERR_TOO_MANY_CONCURRENT_TASKS
- MSK_RES_WRN_TOO_MANY_THREADS_CONCURRENT

BIBLIOGRAPHY

[AA95] E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Math. Programming*, 71(2):221–245, 1995.

[AGMX96] 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.

[AY98] E. D. Andersen and Y. Ye. A computational study of the homogeneous algorithm for large-scale convex optimization. *Computational Optimization and Applications*, 10:243–269, 1998.

[AY99] E. D. Andersen and Y. Ye. On a homogeneous algorithm for the monotone complementarity problem. *Math. Programming*, 84(2):375–399, February 1999.

[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.

[BSS93] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear programming: Theory and algorithms.* John Wiley and Sons, New York, 2 edition, 1993.

[Chv83] V. Chvátal. *Linear programming.* W.H. Freeman and Company, 1983.

[CT07] Gerard Cornuejols and Reha Tütüncü. *Optimization methods in finance.* Cambridge University Press, New York, 2007.

[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.

[MOSEKApS12] MOSEK ApS. *The MOSEK Modeling Cookbook*. MOSEK ApS, Fruebjergvej 3, Boks 16, 2100 Copenhagen O, 2012. URL: https://docs.mosek.com/modeling-cookbook/index.html.

SYMBOL INDEX

Enumerations

Functions

Parameters

Response codes

Types

INDEX