Users Guide to the

R-to-MOSEK Interface

Made by:
Henrik Alsing Friberg <haf@mosek.com>

Package 'Rmosek' v1.2.5
October 10, 2012

Scope

The MOSEK optimization library provides a solver with industrial strength capable of solving huge convex problems: linear, quadratic, conic, continuous and mixed integer. This project was initiated to open the playing field for users of R and grant access to MOSEK through a package. The interface is simple to learn and utilize despite its many features, and thus provides a perfect tool for research and educational projects. However, it should be noted that the project is not part of the MOSEK product line. It does not reflect the full capabilities of the MOSEK optimization library, and it is not guaranteed to stay updated nor backward compatible. It merely provides the basic functionality for optimization, on which the advanced user is welcome to extend upon.

Contents

1 Installation

1.1 Windows platforms

The interface documented here is part of the *Rmosek* package available on CRAN (the Comprehensive R Archive Network) from where it can be downloaded. Notice, however, that a pre-compiled binary version of it has not been distributed[1]. Thus a small amount of configuration is necessary in order to install the package. It is not very difficult, but besides the basic installation of ***R*** you will need two pieces of software readily available through the Internet. In summary:

- **MOSEK** (the optimization library we interface to)[2]
- **Rtools** (the tools needed for R package development)

The following is a step-by-step guide through the installation of the package. We skip the installation of MOSEK as it has been well documented elsewhere and assume that the target machine already has a working installation. Please refer to the documentation available at *mosek.com* for the verification of this installation. The basic test for this is:

1. Does calling `mosek -f` from the Windows CMD cause errors?

For additional help installing this interface, the section on packages in the "R for Windows FAQ" available on the CRAN website may be useful. The "R Installation and Administration" document, also published by CRAN, is another good source of information.

[1] Such a binary would have to be built individually for each version of R and MOSEK.

[2] Check *mosek.com* on free licenses for academic use and trial evaluation periods. For commercial use, licenses needed to handle a problem with MOSEK is also those needed when using Rmosek.

Setting up the target machine

Assuming a working installation of R and MOSEK on the machine targeted for the R-to-MOSEK interface, the first step is to download Rtools for Windows[3]. From this program you will need to install the component called *R toolset*, the *Cygwin DLLs*, and the *R toolchain* (previously called MinGW for 32 bit and x64 MinGW for 64 bit).

x64 MinGW was not supported in MOSEK prior to version 6.0.0.122.

> *R toolset and Cygwin DLLs will extend the Windows CMD with Unix-style commands, while the R toolchain (based on MinGW compilers) makes it possible to compile the C++ source code in the package. These components can be replaced by any other Unix-style shell and C++ build chain, but in the remaining guide the use of Rtools will be assumed.*

After the installation of Rtools you will have to set the Windows environment variable called ***PATH***, in order to utilize the components. Assuming that the home directory of the Rtools installation was `c:\Rtools`, the entries shown below will have to be added to the existing PATH variable. Note that all entries in the PATH variable must be separated by a semicolon (;), and that all these entries have to represent folders that exist on the target machine.

Add `c:\Rtools\bin;` to enable the R toolset and Cygwin DLLs.
Add `c:\Rtools\gcc-VERSION\bin;` (for some VERSION) to enable the R toolchain.

That was it, but before we move on to the installation process, please ensure that the PATH variable also contains the "bin" folder of all MOSEK installations (32 and/or 64 bit) that you wish this interface to target. This is necessary for automatic configuration to work, and could look something like:

```
C:\Progra~1\Mosek\6\tools\platform\win64x86\bin
```

Installing the package with automatic configuration

Automatic configuration works equivalently to calling `where mosek` in the Windows CMD. It searches the environment variable called PATH for a folder with an executable called 'mosek'. Note that if more than one such folder exists, only the one mentioned first in the PATH variable is chosen. It then determines the most ordinary of the available optimization libraries within this folder (typically mosek.lib or mosek64.lib), along with other relevant information. This configuration should work for all users installing the package on a single architecture (64 or 32 bit) and only requiring the ordinary optimization library. Otherwise, manual configuration of the package will be needed.

[3] Newest release of Rtools available at cran.r-project.org/bin/windows/Rtools

Now open R for the architecture (64 or 32 bit) you wish to install the package on. Make sure all your packages are up to date by writing `update.packages()`, and execute a command similar to the one shown below. This will install the 'Rmosek' package:

```
install.packages("Rmosek", type="source", INSTALL_opts="--no-multiarch")
```

Note that this package will have to be installed from *source* as it needs a static link to the MOSEK optimization library. This unfortunately means that dependencies (i.e. the 'Matrix' package) will also be installed from source if new releases are available. Since it is more time-consuming to install the 'Matrix' package from source, it is recommended to start with a call to `update.packages()`. The availability of the 'Matrix' package should not be a problem, as it has been part of the standard R installation since version 2.9.0.

Installing the package with manual configuration

If the automatic configuration does not suit your particular needs, or fails for some reason, a manual configuration may work instead. Unfortunately the `configure.vars` parameter of the `install.packages` command does not work on Windows, meaning that the files of the 'Rmosek' source package will have to be edited. This can be difficult for non-savvy users, but have hopefully been documented sufficiently here.

How to manually configure the 'Rmosek' package?
When you download the package from CRAN it comes in a compressed archive called *Rmosek_VERSION.tar.gz* for some version number (VERSION). In order to configure the package, you will have to go through the following steps. A more thorough explanation of each step will be given afterwards.

1. Extract the archive into a directory.
2. Setup the local system descriptors, *Localsys.txt*, for each of the sub-architectures you will be using: 64 bit (e.g. x64) and/or 32 bit (e.g. i386). These files come with a guide written into them that clearly states how this should be done.
3. Compress the directory back into an archive.

The first step is to extract the *Rmosek_VERSION.tar.gz* archive that you downloaded from the CRAN website. This can either be done by using one of the many tools freely available online[4], or the `tar`-command that was installed with Rtools. If you choose to use

[4]Search the web for the keywords: `Windows`, `Packing`, `Unpacking`, `Tar`, `Gz`

the `tar`-command, you can extract the package by opening Windows CMD and executing a command similar to:

```
tar --no-same-owner -zxvf LOCATION\Rmosek_VERSION.tar.gz
```

Remember to exchange *LOCATION* and *VERSION* with the correct values. For those not familiar with Windows CMD, we recommend the use of an external tool as this configuration can then be performed entirely within Windows Explorer.

The second step is to tell the package where to find the MOSEK optimization library. Open the extracted directory called `Rmosek` in either Windows CMD or Windows Explorer, and navigate to the subdirectory called `src\setup`. If you want to install a 64 bit version of the package (making an interface between 64 bit R and 64 bit MOSEK), open the folder `x64` and follow the guide in Localsys.txt. If you want to install a 32 bit version of the package (making an interface between 32 bit R and 32 bit MOSEK), open instead the folder `i386` and follow the guide in this Localsys.txt. An example of this is shown in Figure 1.

The third and final step is to compress the (previously extracted and now altered) directory called `Rmosek`, back into the *Rmosek_VERSION.tar.gz* archive. Again you can either make use of the external tools from step one, or open the Windows CMD and execute a command similar to:

```
tar -zcvf Rmosek_VERSION.tar.gz Rmosek
```

How to install the manually configured 'Rmosek' package?
Open R, either the console or the graphical user interface, for the architecture (64 or 32 bit) you wish to install the package on. Make sure all your packages are up to date by writing `update.packages()`, and execute the following command to install the 'Rmosek' package:

```
install.packages("LOCATION\Rmosek_VERSION.tar.gz", repos=NULL,
                      type="source", INSTALL_opts="--no-multiarch")
```

Remember to exchange *LOCATION* and *VERSION* with the correct values. The first argument should be the path to your manually configured package. The second argument tells that the package is local and not in an online repository. The third argument tells that it is a source package and so should be compiled (this is where MinGW or x64 MinGW is implicitly used). The fourth and final argument specifies that you only wish to install the package to the specific architecture (64 or 32 bit) of the opened R program. If you wish to install on both architectures, and did configure the package for both 32 and 64 bit, simply remove this last argument.

Notice that if you wish to uninstall the 'Rmosek' package at some point later, this can be done as for any other package with the command `remove.packages("Rmosek")`.

Figure 1: The contents of file ./src/setup/x64/Localsys.txt

```
##
## Greetings user of the R-to-MOSEK interface!
##
## If you are sitting on a WINDOWS 64bit platform, this is the file that
## you will have to setup before this package can be installed.
## (see e.g. the R-to-MOSEK userguide)
##

###################
## Step 1 of 2  ##
###################
## Please substitute [MOSEK_HOME_PATH] below, with the path to the
## platform-specific folder within the MOSEK installation you want to
## use. Note that this path should contain a "bin" and a "h" folder.
## ---------------------------------------------------------------------
## For example you can write:
##    PKG_MOSEKHOME=C:\Progra~1\Mosek\6\tools\platform\win64x86
##
## If your computer contains the two directories:
##    C:\Progra~1\Mosek\6\tools\platform\win64x86\bin
##    C:\Progra~1\Mosek\6\tools\platform\win64x86\h
## ---------------------------------------------------------------------

PKG_MOSEKHOME=[MOSEK_HOME_PATH]


###################
## Step 2 of 2  ##
###################
## Please substitute [MOSEK_LIB_FILE] below, with the name of the library
## you wish to use within the "bin" folder of your PKG_MOSEKHOME path.
## This "bin" folder must contain a file called [MOSEK_LIB_FILE].lib.
## ---------------------------------------------------------------------
## Continuing the example from above, you can write:
##    PKG_MOSEKLIB=mosek64_6_0
##
## If your computer contains the file:
##    C:\Progra~1\Mosek\6\tools\platform\win64x86\bin\mosek64_6_0.lib
## ---------------------------------------------------------------------

PKG_MOSEKLIB=[MOSEK_LIB_FILE]
```

1.2 UNIX-alike platforms

The interface documented here is part of the *Rmosek* package available on CRAN (the Comprehensive R Archive Network) from where it can be downloaded. Notice, however, that a pre-compiled binary version of it has not been distributed[5]. Thus a small amount of configuration is necessary in order to install the package. It is not very difficult, but besides the basic installation of ***R*** you will need ***MOSEK*** (the optimization library we interface to)[6] readily available through the Internet.

The following is a step-by-step guide through the installation of the package. We skip the installation of MOSEK as it has been well documented elsewhere and assume that the target machine already has a working installation. Please refer to the documentation available at *mosek.com* for the verification of this installation. The two basic tests are:

1. Does calling `mosek` from a terminal window cause errors?
2. Does calling `Sys.getenv("LD_LIBRARY_PATH")` from the R console contain a "bin" directory from a MOSEK installation?
   – *Note that this variable is called* `"DYLD_LIBRARY_PATH"` *on Machintosh.*

For additional help on installing this interface, the section on installing packages in the document "R Installation and Administration", published by CRAN, is a good source of information.

Setting up the target machine

We assume here that you have a working installation of R and MOSEK on the machine targeted for the R-to-MOSEK interface. The architectures (32 or 64 bit) of these two programs must be exactly the same for consistency. We further assume that the target machine have a build chain installed. Notice, however, that the x64 MinGW compiler was not supported in MOSEK prior to version 6.0.0.122. Installations of Rmosek linking to older versions of the 64 bit version of MOSEK should thus use a different compiler.

`x64 MinGW was not supported in MOSEK prior to version 6.0.0.122.`

For automatic configuration to work, the PATH variable should contain the "bin" folder of the single MOSEK installation (32 or 64 bit) that you wish this interface to target. This could look something like:

[5] Such a binary would have to be built individually for each version of R and MOSEK.

[6] Check *mosek.com* on free licenses for academic use and trial evaluation periods. For commercial use, licenses needed to handle a problem with MOSEK is also those needed when using Rmosek.

```
~/mosek/6/tools/platform/linux64x86/bin
```

If more than one "bin" folder from a MOSEK installation are specified, only the first one will be found by automatic configuration. Additional architectures can be added afterwards if necessary.

Installing the package with automatic configuration

Automatic configuration works equivalently to calling `which mosek` in a terminal window. It searches the environment variable called PATH for a folder with an executable called 'mosek'. Note that if more than one such folder exists, only the one mentioned first in the PATH variable is chosen. It then determines the most ordinary of the available optimization libraries within this folder (typically libmosek or libmosek64 with the extension .so or .dylib), along with other relevant information. This configuration should work for all users only installing the package on a single architecture (64 or 32 bit) and only requiring the ordinary optimization library. Otherwise, manual configuration of the package will be needed.

Now open R for the architecture (64 or 32 bit) you wish to install the package on. Make sure all your packages are up to date by writing `update.packages()`, and execute a command similar to the one shown below. This will install the 'Rmosek' package:

```
install.packages("Rmosek", type="source", INSTALL_opts="--no-multiarch")
```

Note that this package will have to be installed from *source* as it needs a static link to the MOSEK optimization library. This unfortunately means that dependencies (i.e. the 'Matrix' package) will also be installed from source if new releases are available. Since it is more time-consuming to install the 'Matrix' package from source, it is recommended to start with a call to `update.packages()`. The availability of the 'Matrix' package should not be a problem, as it has been part of the standard R installation since version 2.9.0.

Installing the package with manual configuration

Open R for one of the architectures (64 or 32 bit) you wish to install the package on. Make sure all your packages are up to date by writing `update.packages()`, and execute a command similar to the one shown below, with a correct definition of PKG_MOSEKHOME and PKG_MOSEKLIB. This will install the 'Rmosek' package:

```
install.packages("Rmosek", type="source", INSTALL_opts="--no-multiarch",
          configure.vars="PKG_MOSEKHOME=...  PKG_MOSEKLIB=...")
```

Remember to exchange the ... of both PKG_MOSEKHOME and PKG_MOSEKLIB with the correct values, explained as follows. The definition of the first argument, PKG_MOSEKHOME, should be the folder in your MOSEK installation, containing a "bin" and "h" subdirectory for the platform and architecture matching that of the opened R program. This could for instance look something like:

```
~/mosek/6/tools/platform/linux64x86
```

Notice that if the folder definition contains spaces you will either have to wrap the definition in single-quotes (preventing auto-expansions such as the ~ above), or add backslashes in front of all spaces.

The definition of the argument PKG_MOSEKLIB should be the name of the optimization library in the "bin" subdirectory that you wish to utilize in the Rmosek package. This library will be statically linked to the package after a successful installation. Note that the name of the optimization library should be specified without the "lib" prefix, and without its file-extension. The PKG_MOSEKLIB would thus normally be either `mosek` or `mosek64` (linking to respectively libmosek.so and libmosek64.so, or respectively libmosek.dylib and libmosek64.dylib, depending on the Unix-alike system). Using `mosek64` requires a 64 bit version of the opened R program and MOSEK installation, while `mosek` implies 32 bit.

How to install on multiple architectures?

If you wish to install the 'Rmosek' package on multiple sub-architectures, you will first have to follow the above guide and install the package on one of the architectures (e.g. 32 or 64 bit). Afterwards as explained here, it can then be extended to other sub-architectures. This is also explained in "R Installation and Administration" published by CRAN, in the subsection "Multiple sub-architectures" under "Installing packages". For this to work you will need a preinstalled version of R and MOSEK in all the sub-architectures you wish this package to work with.

Open R in the sub-architecture you wish to extend your installation to, and execute a command similar to the one shown below:

```
install.packages("Rmosek", type="source", libs_only=TRUE,
       configure.vars="PKG_MOSEKHOME=...  PKG_MOSEKLIB=...")
```

Remember to exchange ... of the variables PKG_MOSEKHOME and PKG_MOSEKLIB, so that declared MOSEK installation always correspond to the current sub-architecture of the opened R program.

How to install from an offline location?
This is almost the same a described above except that you would have to add the argument `repos=NULL` to tell R that it should not use an online repository. Also, instead of writing the package name "Rmosek", you should write the entire file-location of the package source file which should look something like "LOCATION/Rmosek_VERSION.tar.gz".

What are the command-line equivalents?
Sometimes you do not want to open R for all sub-architectures, but instead perform the installation directly from the console. Notice though, that this will require you to download the package source as you will only be able to install from an offline location in this way. In the two commands below you will have to replace SUBARCH_PATH, DIR and VERSION with the correct values, but doing so should be straight forward. Notice that on newer versions of R (>= 2.12), you can also use the call `R --arch SUBARCH`, instead of specifying the SUBARCH_PATH. Remember to exchange ... of the variables PKG_MOSEKHOME and PKG_MOSEKLIB, with the correct values as previously explained.

For the first architecture:

```
SUBARCH_PATH/R CMD INSTALL DIR/Rmosek_VERSION.tar.gz --no-multiarch
          --configure-vars="PKG_MOSEKHOME=...  PKG_MOSEKLIB=..."
```

For subsequent architectures:

```
SUBARCH_PATH/R CMD INSTALL DIR/Rmosek_VERSION.tar.gz --libs-only
          --configure-vars="PKG_MOSEKHOME=...  PKG_MOSEKLIB=..."
```

2 A guided tour

2.1 Help: How do I use the interface?

To access the functions of the R-to-MOSEK interface, the *Rmosek* package have to be loaded first. Opening the R-console and typing the command:

```
require(Rmosek)
```

will load the package if it has been installed properly. If this does not work, you should go back to Section 1 and check your installation. Note that the *Matrix* package will automatically be loaded also as the *Rmosek* package depends on it for the representation of sparse matrices. The *Matrix* package have been part of the standard R installation since version 2.9.0, and should thus be available without further notice.

If the package loaded successfully without errors, you should now be able to use the functions of the R-to-MOSEK interface. For example, the following function will return the version of the MOSEK optimization library that this package is currently linked to:

```
mosek_version()
```

To see a summary of the *Rmosek* package with a complete list of all exported functions, the following command can be used:

```
library(help="Rmosek")
```

In addition to `mosek_version`, this list also includes `mosek`, which is the main function of the package able to solve a broad range of convex optimization problems using one of the optimizers from MOSEK. When `mosek` is called for the first time, a MOSEK environment is acquired occupying one user license. Given the extra overhead of acquiring and releasing licenses, this environment is typically held until R is terminated. If the user wish to release the environment and occupied license earlier than this, a call to `mosek_clean` will free such resources. To support other modeling languages, the functions `mosek_read` and `mosek_write` is able to handle model files in several formats. The functions `mosek_lptoprob` and `mosek_qptoprob` offers an alternative way of formulating linear and quadratic programs.

`NB!` If you share a limited number of licenses among multiple users, remember to call `mosek_clean()` - or terminate R - to release yours.

Being aware of how the user licenses of MOSEK are handled in this interface, we can now proceed with guidance on how to use the available functions. For instance, if you wish to know more about the `mosek` function, you would simple have to execute the following command:

```
?mosek
```

This will give you a detailed description of the function, including a list of recognized input arguments, an outline of the expected output, and an example of its usage. This information will also be shown if you fail to call a function with the correct number of input arguments. As an example, the input arguments and return value printed as part of the information given when calling `?mosek`, is shown in Figure 2.

Many of the input and output arguments are probably not relevant for your project and so will not all be explained here, but instead elaborated in the sections to which they belong. If you are new to the interface, you can simply to read on. If you on the other hand are looking for something specifically, it may be beneficial to look at the brief summary in Appendix A for references to more information.

Figure 2: Function manual summary: mosek

```
Solve an optimization problem

Usage:
    r <- mosek(problem, opts = list())

Arguments:
    problem                     LIST
    ..$sense                    STRING
    ..$c                        NUMERIC VECTOR
    ..$c0                       NUMERIC                 (OPTIONAL)
    ..$A                        SPARSE MATRIX
    ..$bc                       NUMERIC MATRIX (2 rows)
    ..$bx                       NUMERIC MATRIX (2 rows)
    ..$cones                    LIST MATRIX (2 rows)    (OPTIONAL)
    ..$intsub                   NUMERIC VECTOR          (OPTIONAL)
    ..$qobj                     LIST                    (OPTIONAL)
    ..$scopt                    LIST                    (OPTIONAL)
    ..$iparam/$dparam/$sparam   LIST                    (OPTIONAL)
    ....$<MSK_PARAM>            STRING / NUMERIC        (OPTIONAL)
    ..$sol                      LIST                    (OPTIONAL)
    ....$itr/$bas/$int          LIST                    (OPTIONAL)
```

```
    opts                        LIST                (OPTIONAL)
    ..$verbose                  NUMERIC             (OPTIONAL)
    ..$usesol                   BOOLEAN             (OPTIONAL)
    ..$useparam                 BOOLEAN             (OPTIONAL)
    ..$soldetail                NUMERIC             (OPTIONAL)
    ..$getinfo                  BOOLEAN             (OPTIONAL)
    ..$writebefore              STRING (filepath)   (OPTIONAL)
    ..$writeafter               STRING (filepath)   (OPTIONAL)


Value:
    r                     LIST
    ..$response           LIST
    ....$code             NUMERIC
    ....$msg              STRING
    ..$sol                LIST
    ....$itr/$bas/$int    LIST            (SOLVER DEPENDENT)
    ......$solsta         STRING
    ......$prosta         STRING
    ......$skc            STRING VECTOR
    ......$skx            STRING VECTOR
    ......$skn            STRING VECTOR   (NOT in $bas)
    ......$xc             NUMERIC VECTOR
    ......$xx             NUMERIC VECTOR
    ......$slc            NUMERIC VECTOR  (NOT IN $int)
    ......$suc            NUMERIC VECTOR  (NOT IN $int)
    ......$slx            NUMERIC VECTOR  (NOT IN $int)
    ......$sux            NUMERIC VECTOR  (NOT IN $int)
    ......$snx            NUMERIC VECTOR  (NOT IN $int/$bas)
    ......$pobjval        NUMERIC         *
    ......$dobjval        NUMERIC         *(NOT IN $int)
    ......$pobjbound      NUMERIC         *($int ONLY)
    ......$maxinfeas      LIST            *
    ........$pbound       NUMERIC         *
    ........$peq          NUMERIC         *
    ........$pcone        NUMERIC         *(NOT in $bas)
    ........$dbound       NUMERIC         *(NOT in $int)
    ........$deq          NUMERIC         *(NOT in $int)
    ........$dcone        NUMERIC         *(NOT in $int/$bas)
    ........$int          NUMERIC         *($int ONLY)
    ..$iinfo/$dinfo       LIST            *
    ....$<MSK_INFO>       NUMERIC         *

    *Starred items must be requested using an option.
```

2.2 Linear Programming

2.2.1 Solving LP problems

Linear optimization problems[7] posses a strong set of properties that makes them easy to solve. Any linear optimization problem can be written as shown below, with variables $x \in \mathbb{R}^n$, constraint matrix $A \in \mathbb{R}^{m \times n}$, objective coefficients $c \in \mathbb{R}^n$, objective constant $c_0 \in \mathbb{R}$, lower and upper constraint bounds $l^c \in \mathbb{R}^m$ and $u^c \in \mathbb{R}^m$, and lower and upper variable bounds $l^x \in \mathbb{R}^n$ and $u^x \in \mathbb{R}^n$. This is the so called primal problem.

$$\begin{array}{lccccc} \text{minimize} & & & c^T x + c_0 & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c\,, \\ & l^x & \leq & x & \leq & u^x\,. \end{array} \tag{2.1}$$

All LP problems can be written on this form where e.g. equality constraint will be given the same upper and lower bound, but what if you have an upper-bounded constraint with no lower-bound? To exclude such bounds, the interface utilizes the R constant *Inf* and allows you to specify lower bounds of minus infinity (*-Inf*) and upper bounds of plus infinity (*Inf*), that will render bounds as non-existing. The interface will always expect LP problems on this form, and will accordingly set the dual variables of the non-existing bounds to zero to satisfy complementarity - a condition for optimality.

The 'lo1' Example (Part 1 of 5) The following is an example of a linear optimization problem with one equality and two inequality constraints:

$$\begin{array}{lccccccccc} \text{maximize} & 3x_1 & + & 1x_2 & + & 5x_3 & + & 1x_4 & & \\ \text{subject to} & 3x_1 & + & 1x_2 & + & 2x_3 & & & = & 30, \\ & 2x_1 & + & 1x_2 & + & 3x_3 & + & 1x_4 & \geq & 15, \\ & & & 2x_2 & & & + & 3x_4 & \leq & 25, \end{array} \tag{2.2}$$

having the bounds

$$\begin{array}{ccccc} 0 & \leq & x_1 & \leq & \infty, \\ 0 & \leq & x_2 & \leq & 10, \\ 0 & \leq & x_3 & \leq & \infty, \\ 0 & \leq & x_4 & \leq & \infty. \end{array} \tag{2.3}$$

This is easily programmed in R as shown in Figure 4. The first line overwrites any previous definitions of the variable *lo1*, preparing for the new problem description. The problem is then defined and finally solved on the last line.

[7] Check out: mosek.com → Documentation → Optimization tools manual → Modeling → Linear optimization.

Figure 4: Linear Optimization (lo1)

```
lo1 <- list()
lo1$sense <- "max"
lo1$c <- c(3,1,5,1)
lo1$A <- Matrix(c(3,1,2,0,
                  2,1,3,1,
                  0,2,0,3), nrow=3, byrow=TRUE, sparse=TRUE)
blc <- c(30,15,-Inf)
buc <- c(30,Inf,25);          lo1$bc <- rbind(blc, buc);
blx <- c(0,0,0,0)
bux <- c(Inf,10,Inf,Inf);     lo1$bx <- rbind(blx, bux);
r <- mosek(lo1)
```

Notice how the R value *Inf* is used in both the constraint bounds (*blc* and *buc*) and the variable upper bound (*bux*), to avoid the specification of an actual bound. If this example does not work you should go back to Section 1 on setting up the interface.

□

From this example the input arguments for the linear program (2.1) follows easily (refer to the definitions of input arguments in Figure 2, Section 2.1).

Objective The string ***sense*** is the objective goal and could be either "minimize", "min", "maximize" or "max". The dense numeric vector ***c*** specifies the coefficients in front of the variables in the linear objective function, and the optional constant scalar ***c0*** (reads: c zero) is a constant in the objective corresponding to c_0 in problem (2.1), that will be assumed zero if not specified.

Constraint Matrix The sparse matrix ***A*** is the constraint matrix of the problem with the constraint coefficients written row-wise. Notice that for larger problems it may be more convenient to define an empty sparse matrix and specify the non-zero elements one at a time $A(i, j) = a_{ij}$, rather than writing out the full matrix as done in the 'lo1' example. E.g. `Matrix(0,nrow=30,ncol=50,sparse=TRUE)`.

Bounds The constraint bounds ***bc*** with rows *blc* (constraint lower bound) and *buc* (constraint upper bound), as well as the variable bounds ***bx*** with rows *blx* (variable lower bound) and *bux* (variable upper bound), are both given as dense numeric matrices. These are equivalent to the bounds of problem (2.1), namely l^c, u^c, l^x and u^x.

▶ Errors, warnings and response codes

If the *mosek* function is executed with a problem description as input, a log of the interface and optimization process is printed to the screen revealing any errors or warnings the process may have encountered. As a rule of thumb, errors will be given when a crucial part of the problem description is missing, or when an input argument is set to a value that does not make sense or is formatted incorrectly. Warnings on the other hand will be given if some ignorable part of the problem has an empty definition (`NULL`, `NA` or `NaN`), or if the interface has to convert or otherwise guess on an interpretation of input on a non-standard form. Errors will always interrupt the optimization process whereas warnings will not. Since warnings can hold valuable debugging information and may be important to notice, they are both printed in the log at the time they occurred and later summarized just before the interface returns.

Error messages works fine when you are interacting with the interface, but in automated optimization frameworks they are not easily handled. This is why a `response` is always returned as part of the result, when calling a function that may produce errors (see e.g. Figure 2). The `response` is a list containing a `code` and `msg`. When an error happens inside a function call to the MOSEK optimization library, the `code` is the response code returned by the function call[8] and `msg` is the corresponding error message. When an error happens within the interface, the `code` equals `NaN` and the `msg` is the error message which, in case of unexpected execution paths, may require technical knowledge to understand. When no errors are encountered, the `code` is zero, but beware that comparison operators such as `'=='` does not handle `NaN` very good. Using `identical` may be a better alternative.

`NB!` The interface may return only partially constructed output.
(always check the response code; e.g. *`success`* `<- identical(`*`response_code`*`, 0)`)

▶ Interpreting the solution

The default optimizer for linear problems is the interior-point algorithm in MOSEK which returns two solutions in the output structure. The interior-point solution (called *itr*) is the solution found directly by the interior-point algorithm. The basic solution (called *bas*) is a vertex solution derived from the values of *itr*, and could for instance be used to hot-start the simplex method if small changes was applied at some point later. If another optimizer using the simplex method was selected instead of the interior-point algorithm, the *bas* solution would have been found directly and the *itr* solution would not exist.

[8] Check out: mosek.com → Documentation → C API manual → Response codes.

The 'lo1' Example (Part 2 of 5) The 'lo1' example was solved using the default optimizer (the interior-point algorithm) and contains two solutions: the interior-point (*itr*) and the basic solution (*bas*) partly shown here.

As seen in Figure 5 and Figure 6 the solution space of the problem was not empty (as it is primal feasible) and the objective was not unbounded (as it is dual feasible). In addition the optimizer was able to identify the optimal solution.

Figure 5: Primal Solution I (lo1)

```
r$sol$itr
{
 $solsta
  [1] "OPTIMAL"
 $prosta
  [1] "PRIMAL_AND_DUAL_FEASIBLE"
 $skc
  [1] "EQ" "SB" "UL"
 $skx
  [1] "LL" "LL" "SB" "SB"
 $skn
  character(0)
 $xc
  [1] 30.00000 53.33333 25.00000
 $xx
  [1] 1.044111e-08 2.856068e-08 1.500000e+01 8.333333e+00
 ...
}
```

Figure 6: Primal Solution II (lo1)

```
r$sol$bas
{
 $solsta
  [1] "OPTIMAL"
 $prosta
  [1] "PRIMAL_AND_DUAL_FEASIBLE"
 $skc
  [1] "EQ" "BS" "UL"
 $skx
  [1] "LL" "LL" "BS" "BS"
 $xc
  [1] 30.00000 53.33333 25.00000
 $xx
  [1]  0.000000  0.000000 15.000000  8.333333
 ...
}
```

Notice that the basic solution *bas* is likely to have a higher numerical accuracy than the interior-point solution *itr* as is the case in this example considering the *xx* variables.

□

The solution you receive from the interface will contain the primal variable x (called ***xx***) and the activity of each constraint, ***xc***, defined by $x^c = Ax$. From the solution status (called ***solsta***) it can be seen how good this solution is, e.g. optimal, nearly optimal, feasible or infeasible. If the solution status is not as you expected, it might be that the problem is either ill-posed, infeasible or unbounded (dual infeasible). This can be read from the problem status (called ***prosta***). The solution and problem status are based on certificates found by the MOSEK optimization library[9], and should always be verified before the returned solution values are used (see Section 2.2.5).

The solution also contains status key vectors for both variables and constraints (called ***skx*** and ***skc***). The variable ***skn*** is not useful for linear programming problems. Each status key will be one of the following two-character strings.

`BS : Basic`
In basic (*bas*) solutions: The constraint or variable belongs to the basis of the corresponding simplex tableau.

`SB : Super Basic`
In interior-point (*itr*) and integer (*int*) solutions: The activity of a constraint or variable is in between its bounds.

`LL : Lower Level`
The constraint or variable is at its lower bound.

`UL : Upper Level`
The constraint or variable is at its upper bound.

`EQ : Equality`
The constraint or variable is at its fixed value (equal lower and upper bound).

`UN : Unknown`
The status of the constraint or variable could not be determined (in practice never returned by MOSEK).

In addition to the primal variables just presented, the returned solutions also contains dual variables not shown here. The dual variables can be used for sensitivity analysis of the problem parameters and are related to the dual problem explained in Section 2.2.2.

[9] More details on problem and solution status keys available at:
mosek.com → Documentation → Optimization tools manual → Symbolic constants reference.

2.2.2 Duality

The dual problem corresponding to the primal problem (2.1) defined in Section 2.2.1, is shown below in (2.4). Notice that the coefficients of the dual problem is the same as those used in the primal problem. Matrix A for example is still the constraint matrix of the primal problem, merely transposed in the dual problem.

In addition, the dual problem have dual variables for each lower and upper, constraint and variable bound in the primal problem: $s_l^c \in \mathbb{R}^m$, $s_u^c \in \mathbb{R}^m$, $s_l^x \in \mathbb{R}^n$ and $s_u^x \in \mathbb{R}^n$ (the latter being the dual variable of the upper variable bound).

The dual problem is given by

$$\begin{array}{lcl} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c_0 \\ \text{subject to} & A^T(s_l^c - s_u^c) + s_l^x - s_u^x = c\,, \\ & s_l^c, s_u^c, s_l^x, s_u^x \geq 0\,. \end{array} \tag{2.4}$$

Recall that equality constraints had to be specified using two inequalities (with $l^c = u^c$) by definition of the primal problem (2.1). This means that an equality constraint will have two dual variables instead of just one. If the user wants to calculate the one dual variable, as it would have been if equality constraints could be specified directly, then this is given by $s_l^c - s_u^c$. However, it is not always recommended to do so, as it is often easier to stay with the same problem formulation and do all calculations directly on that.

The 'lo1' Example (Part 3 of 5) The part of the solution to the 'lo1' example that was previously omitted, is now shown in Figure 7 and 8. The dual variables *slc*, *suc*, *slx* and *sux* corresponds naturally to s_l^c, s_u^c, s_l^x and s_u^x in the dual problem. The variable *snx* is not useful for linear programming problems.

Looking at the definition of the 'lo1' problem (2.2), the first constraint is an equality, the second is an lower bound, and the third is an upper bound. The dual variable of the inequalities should just be read from *slc* and *suc* respectively, while for the equality constraint, having two dual variables, you could also look at the combined lower minus upper constraint dual variable (*slc* - *suc*), which in this case would give you a dual value of 2.5.

Figure 7: Dual Solution I (lo1)

```
r$sol$itr
{
 $solsta
  [1] "OPTIMAL"
 $prosta
  [1] "PRIMAL_AND_DUAL_FEASIBLE"
 ...
 $slc
  [1] 0.000000e+00 1.557535e-09 0.000000e+00
 $suc
  [1] -2.5000000  0.0000000 -0.3333333
 $slx
  [1] -4.500000e+00 -2.166667e+00 -4.982961e-09 -1.027032e-08
 $sux
  [1] 0.0000e+00 5.5856e-10 0.0000e+00 0.0000e+00
 $snx
  [1] 0 0 0 0
}
```

Figure 8: Dual Solution II (lo1)

```
r$sol$bas
{
 $solsta
  [1] "OPTIMAL"
 $prosta
  [1] "PRIMAL_AND_DUAL_FEASIBLE"
 ...
 $slc
 [1] 0 0 0
 $suc
 [1] -2.5000000  0.0000000 -0.3333333
 $slx
 [1] -4.500000 -2.166667  0.000000  0.000000
 $sux
 [1] 0 0 0 0
}
```

Notice that the basic solution *bas* is likely to have a higher numerical accuracy than the interior-point solution *itr* as is the case here.

□

2.2.3 Switching Optimizer

The integer parameter[10] *OPTIMIZER* controls which optimizer to use within the MOSEK optimization library to solve the specified problem. The default value of this parameter is

[10] Check out Section 2.5 for more details on parameter settings.

the enum reference-string "OPTIMIZER_FREE" which imply that MOSEK should choose an optimizer on its own. Currently MOSEK always selects the interior-point algorithm for linear programming problems which performs especially well for large optimization problems, but for small to medium sized problems it might sometimes be beneficial to switch over to the simplex method.

To solve a linear programming problem using another user-specified optimizer, the *OPTIMIZER* parameter can be set to one of the following reference-strings[11]:

`OPTIMIZER_FREE`
The default parameter setting discussed above.

`OPTIMIZER_INTPNT`
The interior-point algorithm.

`OPTIMIZER_FREE_SIMPLEX`
The simplex method on either the primal or dual problem (MOSEK selects).

`OPTIMIZER_PRIMAL_SIMPLEX`
The simplex method on the primal problem.

`OPTIMIZER_DUAL_SIMPLEX`
The simplex method on the dual problem.

The 'lo1' Example (Part 4 of 5) In this example we shall try to solve the 'lo1' example from Section 2.2.1 using the primal simplex method. This is specified by setting the integer parameter to the reference-string "OPTIMIZER_PRIMAL_SIMPLEX" as shown in Figure 9, adding a single line to the 'lo1' definition from earlier.

Figure 9: Selecting the primal simplex method

```
lo1 <- list()
lo1$sense <- "max"
lo1$c <- c(3,1,5,1)
lo1$A <- Matrix(c(3,1,2,0,
                  2,1,3,1,
                  0,2,0,3), nrow=3, byrow=TRUE, sparse=TRUE)
blc <- c(30,15,-Inf)
buc <- c(30,Inf,25);   lo1$bc <- rbind(blc, buc);
blx <- c(30,15,-Inf)
bux <- c(30,Inf,25);   lo1$bx <- rbind(blx, bux);
lo1$iparam <- list(OPTIMIZER="OPTIMIZER_PRIMAL_SIMPLEX");
r <- mosek(lo1)
```

[11] More values for the *MSK_IPAR_OPTIMIZER* parameter available at:
mosek.com → Documentation → Optimization tools manual → Parameter reference.

The output only contains the optimal basic solution *bas*, which is equivalent to the basic solution found by the interior-point algorithm in the previous 'lo1' examples. To verify that the primal simplex method was actually the optimizer used for this problem, we can check the log printed to the screen and shown in Figure 10.

Figure 10: The log verifies the choice of optimizer

```
...
Optimizer started.
Simplex optimizer started.
Presolve started.
Linear dependency checker started.
Linear dependency checker terminated.
Eliminator - tries                  : 0            time : 0.00
Eliminator - elim's                 : 0
Lin. dep.  - tries                  : 1            time : 0.00
Lin. dep.  - number                 : 0
Presolve terminated. Time: 0.00
Primal simplex optimizer started.
Primal simplex optimizer setup started.
Primal simplex optimizer setup terminated.
...
```

□

2.2.4 Hot-starting

Hot-starting (also known as warm-starting) is a way to make the optimization process aware of a point in the solution space which, depending on the quality of it (feasibility and closeness to the optimal solution), can increase the speed of optimization. In linear programming it is typically used when you know the optimal solution to a similar problem with only few small changes to the constraints and objective. In these cases it is assumed that the next optimal solution is nearby in the solution space, and thus it would also makes sense to switch to the simplex optimizers excellent for small changes to the set of basic variables - even on large problems. In fact, currently, the user will have to use one of the simplex optimizers for hot-starting in linear programming, as the interior-point optimizer in MOSEK cannot take advantage of initial solutions.

Simplex optimizers only look for the basic solution *bas* in the input argument *$sol*, and do not consider the solution and problem statuses within. These may however be specified anyway for the convenience of the user, and warnings will only be given if no useful information could be given to the MOSEK optimizer despite the fact that *$sol* had a non-empty definition and hot-starting was attempted.

▶ When adding a new variable

In column generation it is necessary to reoptimize the problem after one or more variables have been added to the problem. Given a previous solution to the problem, the number of basis changes would be small and we can hot-start using a simplex optimizer.

Assume that we would like to solve the problem

$$\begin{array}{lrcrcrcrcrcl} \text{maximize} & 3x_1 & + & 1x_2 & + & 5x_3 & + & 1x_4 & - & 1x_5 & & \\ \text{subject to} & 3x_1 & + & 1x_2 & + & 2x_3 & & & - & 2x_5 & = & 30, \\ & 2x_1 & + & 1x_2 & + & 3x_3 & + & 1x_4 & - & 10x_5 & \geq & 15, \\ & & & 2x_2 & & & + & 3x_4 & + & 1x_5 & \leq & 25, \end{array} \tag{2.5}$$

having the bounds

$$\begin{array}{rcrcl} 0 & \leq & x_1 & \leq & \infty, \\ 0 & \leq & x_2 & \leq & 10, \\ 0 & \leq & x_3 & \leq & \infty, \\ 0 & \leq & x_4 & \leq & \infty, \\ 0 & \leq & x_5 & \leq & \infty, \end{array} \tag{2.6}$$

which is equal to the 'lo1' problem (2.2), except that a new variable x_5 has been added. To hot-start from the previous solution of Figure 4, which is still primal feasible, we can expand the problem description and include x_5 as shown.

Figure 11: Hot-starting when a new variable is added

```
# Define 'lo1' and obtain the solution 'r' (not shown)
lo1_backup <- lo1

# Append the new variable to the problem:
lo1$c  <- c(lo1$c, -1)
lo1$bx <- cbind(lo1$bx, c(0,Inf))
lo1$A  <- cBind(lo1$A, c(-2,-10,1))

# Extend and reuse the old basis solution:
oldbas <- r$sol$bas
oldbas$skx <- c(oldbas$skx, 'LL')
oldbas$xx  <- c(oldbas$xx , 0)
oldbas$slx <- c(oldbas$slx, 0)
oldbas$sux <- c(oldbas$sux, 0)

# Hot-start the simplex optimizer:
lo1$iparam <- list(OPTIMIZER="OPTIMIZER_PRIMAL_SIMPLEX")
lo1$sol <- list(bas=oldbas)
r_var <- mosek(lo1)
```

▶ When fixing a variable

In branch-and-bound methods for integer programming it is necessary to reoptimize the problem after a variable has been fixed to a value. From the solution of the 'lo1' problem (Figure 4), we fix the variable $x_4 = 2$, and hot-start using

Figure 12: Hot-starting when a variable has been fixed

```
# Define 'lo1' and obtain the solution 'r' (not shown)
lo1_backup <- lo1

# Fix the fourth variable in the problem
lo1$bx[,4] <- c(2,2)

# Reuse the old basis solution
oldbas <- r$sol$bas

# Hotstart the simplex optimizer
lo1$iparam <- list(OPTIMIZER="OPTIMIZER_DUAL_SIMPLEX")
lo1$sol <- list(bas=oldbas)
r_fix <- mosek(lo1)
```

▶ When adding a new constraint

In cutting plane algorithms it is necessary to reoptimize the problem after one or more constraints have been added to the problem. From the solution of the 'lo1' problem (Figure 4), we add the constraint $x_1 + x_2 \geq 2$, and hot-start using

Figure 13: Hot-starting when a constraint has been added

```
# Define 'lo1' and obtain the solution 'r' (not shown)
lo1_backup <- lo1

# Append the new constraint to the problem:
lo1$bc <- cbind(lo1$bc, c(2,Inf))
lo1$A  <- rBind(lo1$A, c(1,1,0,0))

# Extend and reuse the old basis solution:
oldbas <- r$sol$bas
oldbas$skc <- c(oldbas$skc, 'LL')
oldbas$xc  <- c(oldbas$xc , 2)
oldbas$slc <- c(oldbas$slc, 0)
oldbas$suc <- c(oldbas$suc, 0)

# Hot-start the simplex optimizer:
lo1$iparam <- list(OPTIMIZER="OPTIMIZER_DUAL_SIMPLEX")
lo1$sol <- list(bas=oldbas)
r_con <- mosek(lo1)
```

▶ Using numerical values to represent status keys

In the previous examples the status keys of constraints and variables were all defined as two-character string codes. Although this makes the status keys easy to read, it might sometimes be easier to work with numerical values. For this reason we now demonstrate how to achieve this with the R-to-MOSEK interface. The explanation of status keys can be found on page 21.

Figure 14: Hot-starting using an initial guess

```
# Define 'lo1' (not shown)

# Define the status key string array
stkeys <- c("BS","SB","LL","UL","EQ","UN")

# Try to guess the optimal status keys
mybas <- list()
mybas$skc = stkeys[c(5,1,4)]
mybas$skx = stkeys[c(3,3,1,1)]

# Hot-start the simplex optimizer
lo1$iparam <- list(OPTIMIZER="OPTIMIZER_PRIMAL_SIMPLEX")
lo1$sol <- list(bas=mybas)
r_guess <- mosek(lo1)
```

So basically `stkeys[[idx]]` will return the status key of index *idx*, and can be vectorized as `stkeys[idxvec]` for a vector of indexes *idxvec*. Going in the opposite direction from status keys to indexes requires a bit more work. For this you can use `match(str,stkeys)` to find the index of a status key *str*, and equivalently `match(strvec,stkeys)` for a vector of status keys *strvec*.

2.2.5 Complete Best Practice example

Linear programs, as well as other optimization models, are often solved as part of a larger framework where the solutions are not just printed on the screen, but instead given as input to other scripts and programs. Such frameworks include mathematical constructions such as branch-and-price, but also include programs with internal optimization models hidden from the end user. In these cases, special attention must be given to the handling of function calls and return values.

▶ Catch execution errors

From within the Rmosek package, execution errors similar to the ones generated by the built-in `stop` function, part of the R language definition, may be provoked. This typically happens when the number of input arguments does not match a valid calling convention, or when one of the input arguments can not be evaluated. Interface errors within the R API or Matrix package could, however, also generate these kinds of errors.

When execution errors are provoked in some function, the control is returned to the outer-most scope (the global environment) and no values are returned from the function call. That is, when calling `res <- some_function(...)` and an execution error is provoked, the variable `res` will remain unaffected and could host old obsolete values.

Luckily, execution errors can be caught with the built-in `try` function which takes a function call as input argument. The result of this `try` function (say `res`), will either be the returned value of the function call or some error class. The error class inherits from "try-error" and can be distinguished from normal output by the boolean result of `inherits(res, "try-error")`.

▶ Inspect the reponse code

The response code may either be a zero (success), a positive integer (error in the MOSEK optimization library) or simply `NaN` (error in the Rmosek interface). The list of response codes that can be returned by the MOSEK optimization library, are similar to those explained online[12]. As explained in "Errors, warnings and response codes" in Section 2.2.1, always use the built-in `identical` function to compare reponse codes.

When an error is encountered, the interface will still try to extract and return a solution. If for instance the optimization procedure ran out of memory or time, or was interrupted by the keyboard sequence <CTRL> + <C> (see e.g. Section 2.7.2), a good solution may have been identified anyway. However, if the extraction of the solution was what caused the

[12]Check out: mosek.com → Documentation → C API manual → Response codes.

error in the first place, the returned solution may only be partially constructed. Ultimately, no guarantees about the availability of variables can be given when the response code is not zero.

In case of a non-zero response code, the availability of a variable within a result `res`, can be tested with an expression similar to `!inherits(try({res$sol$itr$solsta})), "try-error")`, which is `TRUE` when a solution status to a interior-point solution in the result has been defined. Notice that a scope has been defined in the input argument of the `try` function using `'{'` and `'}'`. Within this scope, several variables can be tested for existence, or alternatively, entire blocks of code may be inserted and attempted evaluated.

▶ Inspect the solution status

The solution status is based on certificates found by the MOSEK optimization library, and classify the returned solution. In the list below, the two first status keys also exists for the dual case with `PRIMAL` replaced by `DUAL`. Also, all these seven status keys (excluding `UNKNOWN`) has an nearly equivalent status key with `NEAR_` added as a prefix.

`PRIMAL_INFEASIBLE_CER (linear problems only)`
The returned data contains a certificate that the problem is primal infeasible.

`PRIMAL_FEASIBLE`
The returned data contains primal feasible solution.

`PRIMAL_AND_DUAL_FEASIBLE`
The returned data contains a primal and dual feasible solution, but is not optimal.

`OPTIMAL`
The returned data contains an optimal solution.

`INTEGER_OPTIMAL (integer problems only)`
The returned data contains an integer optimal solution.

`UNKNOWN`
The returned data should not be used.

A response code of zero is not enough to conclude that a solution can be extracted from the returned data. In particular, the returned data may be a certificate of primal or dual infeasibility instead of a solution to the problem. Moreover, the returned primal variables may not contain any useful information if the solution status is e.g. `DUAL_FEASIBLE`. This also holds true for dual variables if the solution status is e.g. `PRIMAL_FEASIBLE`. Thus it is important to check and respond to the solution status before using the solution variables.

▶ Inspect the problem status

The problem status is based on certificates found by the MOSEK optimization library, and classify the specified problem description. In the list below, the three first status keys also exists for the dual case with `PRIMAL` replaced by `DUAL`, and with `PRIMAL` replaced by `PRIMAL_AND_DUAL`.

`PRIMAL_INFEASIBLE`
The problem description is primal infeasible.

`NEAR_PRIMAL_FEASIBLE`
The problem description satisfies a relaxed primal feasibility criterion.

`PRIMAL_FEASIBLE`
The problem description is primal feasible.

`ILL_POSED (non-linear problems only)`
The problem description has an unstable formulation.

Regarding the usefulness of the returned data, the solution status often tells the whole story. For mixed integer programs (see Section 2.4), there is however no certificates for infeasibility and this status have to be read from the problem status. Also, if the solution status is `UNKNOWN`, the problem status may still contain valuable information. Furthermore, for non-linear problems, it is a good idea to verify that the problem status is not `ILL_POSED`.

The 'lo1' Example (Part 5 of 5) In this example we define a function which shall try to solve the 'lo1' example from Section 2.2.1, and return the variable activities of the basic solution. The function is completely silent except for the explicit `cat`-commands, and will not fail to return a usable result unless explicitly stopped by a `stop`-command. Furthermore, the double-typed parameter *OPTIMIZE_MAX_TIME* (see e.g. the FAQ, Section 2.7.3) can be controlled by the input argument `maxtime`. If zero, MOSEK will have no time to identify a useful solution. The function is shown in Figure 15.

As seen, the verbosity is set to zero with the option *verbose* to specify that the interface should be completely silent (see the FAQ, Section 2.7.1). Also, any execution errors that may be thrown by the `mosek` function is caught. The function then verifies the response code, and chooses to continue even this is not zero (you may choose to do otherwise). It then extracts the basic solution and, at the same time, evaluate whether all the variables that we are going to use are defined. We do not bother to check if the problem status is ill-posed, as 'lo1' is a linear program, but instead continue to the solution status which is only accepted if it is close to optimal (again, you may choose to do otherwise). Finally the solution variable activities of the solution is returned.

Figure 15: Complete Best Practice 'lo1' example

```
get_lo1_solution_variables <- function(maxtime) {
  lo1 <- list(sense="max", c=c(3,1,5,1))
  lo1$A <- Matrix(c(3,1,2,0,2,1,3,1,0,2,0,3),
              nrow=3, byrow=TRUE, sparse=TRUE)
  lo1$bc <- rbind(blc=c(30,15,-Inf), buc=c(30,Inf,25));
  lo1$bx <- rbind(blx=c(0,0,0,0),    bux=c(Inf,10,Inf,Inf));
  lo1$dparam <- list(OPTIMIZER_MAX_TIME=maxtime)

  r <- try(mosek(lo1, list(verbose=0)), silent=TRUE)
  if (inherits(r, "try-error")) {
    stop("Rmosek failed somehow!")
  }

  if (!identical(r$response$code, 0)) {
    cat(paste("**", "Response code:", r$response$code, "\n"))
    cat(paste("**", r$response$msg, "\n"))
    cat("Trying to continue..\n")
  }

  isdef <- try({
    rbas <- r$sol$bas;
    rbas$solsta;  rbas$prosta;  rbas$xx;
  }, silent=TRUE)
  if (inherits(isdef, "try-error")) {
    stop("Basic solution was incomplete!")
  }

  switch(rbas$solsta,
    OPTIMAL = {
      cat("The solution was optimal, I am happy!\n")
    },
    NEAR_OPTIMAL = {
      cat("The solution was close to optimal, very good..\n")
    },
    #OTHERWISE:
    {
      cat(paste("**", "Solution status:", rbas$solsta, "\n"))
      cat(paste("**", "Problem status:", rbas$prosta, "\n"))
      stop("Solution could not be accepted!")
    }
  )
  return(rbas$xx)
}
```

□

2.3 Conic Quadratic Programming

2.3.1 Solving CQP problems

Conic quadratic programming[13] (also known as Second-order cone programming) is a generalization of linear, quadratic and all-quadratic programming. Conic quadratic programs can be written as shown below, and pose the properties of strong duality under Slater's condition. This is the primal problem.

$$\begin{array}{lccccc} \text{minimize} & & & c^T x + c_0 & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \\ & & & x \in \mathcal{C}. & & \end{array} \tag{2.7}$$

The convex cone $\mathcal{C}$ can be written as the Cartesian product over a finite set of convex cones $\mathcal{C} = \mathcal{C}_1 \times \cdots \times \mathcal{C}_p$, which basically means that the variables can be partitioned into subsets of variables belonging to different cones. In principle this also means that each variable can only belong to one cone, but in practice we can define several duplicates $\hat{x}_i$ of x_i belonging to different cones and connected by $\hat{x}_i = x_i$ in the linear constraints of (2.7).

The MOSEK optimization library currently allows three types of convex cones: The $\mathbb{R}$-cone, the quadratic cone and the rotated quadratic cone. The $\mathbb{R}$-cone contains the full set of real numbers and is the default cone in this interface for variables with no other specification. Notice that if all variables belonged to this cone the problem would reduce to a linear programming problem. The *quadratic cone* is defined by

$$\mathcal{C}_t = \left\{ x \in \mathbb{R}^{n_t} \ : \ x_1 \geq \sqrt{\sum_{j=2}^{n_t} x_j^2} \right\} \tag{2.8}$$

for which the indexes shown here refer only to the subset of variables belonging to the cone. Similarly the *rotated quadratic cone* is given by

$$\mathcal{C}_t = \left\{ x \in \mathbb{R}^{n_t} \ : \ 2x_1 x_2 \geq \sum_{j=3}^{n_t} x_j^2, \ x_1 \geq 0, \ x_2 \geq 0 \right\}. \tag{2.9}$$

These definitions may seem restrictive, but can model a large number of problems as shown by the transformations of Appendix B.

[13] Check out: mosek.com → Documentation → Optimization tools manual → Modeling → Conic optimization.

The 'cqo1' Example (Part 1 of 3) The following is an example of a conic optimization problem with one linear constraint, non-negative variables and two cones:

$$\begin{array}{lcll} \text{minimize} & x_4 + x_5 + x_6 & & \\ \text{subject to} & x_1 + x_2 + 2x_3 & = & 1\,, \\ & x_1, x_2, x_3 & \geq & 0\,, \\ & x_4 \geq \sqrt{x_1^2 + x_2^2}\,, & & \\ & 2x_5 x_6 \geq x_3^2\,. & & \end{array} \tag{2.10}$$

The first cone is of the quadratic cone type (*MSK_CT_QUAD*), while the second is of the rotated quadratic cone type (*MSK_CT_RQUAD.*) The subindexes of the variables used to define these cones follow naturally as seen in the following R code.

Figure 16: Conic Quadratic Optimization (cqo1)

```
cqo1 <- list(sense = "min")
cqo1$c <- c(0,0,0,1,1,1)
cqo1$A <- Matrix(c(1,1,2,0,0,0),
                 nrow=1, byrow=TRUE, sparse=TRUE)
cqo1$bc <- rbind(blc = 1, buc = 1)
cqo1$bx <- rbind(blx = c(0,0,0,-Inf,-Inf,-Inf),
                 bux = rep(Inf,6))
cqo1$cones <- cbind(
  list("QUAD", c(4,1,2)),
  list("RQUAD", c(5,6,3))
)
rownames(cqo1$cones) <- c("type","sub");
r <- mosek(cqo1)
```

□

From this example the input arguments for a conic program (2.7) follow easily (refer to Figure 2, Section 2.1). The objective function, the linear constraints and variable bounds should all be specified as for linear programs (see Section 2.2), and the only addition to this is the quadratic cones specified in the list-typed matrix ***cones***.

The *cones* matrix has a column for each cone, and a row for each descriptive element. The first row called ***type***, should specify the cone type in a string, being either quadratic "QUAD" or rotated quadratic "RQUAD". Notice that the MOSEK library cone type prefix "MSK_CT_" is optional. The second row called ***sub***, should specify the subset of variables belonging to the cone in a numeric vector - and the ordering does matter! The i'th element of *sub* will be the index of the variable referred by x_i, in the cone definitions (2.8) and (2.9). As an example, the rotated quadratic cone with subindexes `c(4,6,2,3)` would define the cone

$$\mathcal{C}_t = \left\{ x \in \mathbb{R}^4 \;:\; 2x_4 x_6 \geq x_2^2 + x_3^2,\; x_4 \geq 0,\; x_6 \geq 0 \right\}. \tag{2.11}$$

Figure 16 showed a simple way to specify cones given an explicit representation. In many practical cases, however, cones are more conveniently specified in chunks or within a loop. For this purpose, preallocation should always be preferred as shown here.

Figure 17: Preallocating cones (cqo1)

```
NUMCONES <- 2
cqo1$cones <- matrix(list(), nrow=2, ncol=NUMCONES)
  rownames(cqo1$cones) <- c("type","sub")
cqo1$cones[,1] <- list("QUAD", c(4,1,2))
cqo1$cones[,2] <- list("RQUAD", c(5,6,3))
```

□

▶ Errors, warnings and response codes (as in Linear Programming)

If the *mosek* function is executed with a problem description as input, a log of the interface and optimization process is printed to the screen revealing any errors or warnings the process may have encountered. As a rule of thumb, errors will be given when a crucial part of the problem description is missing, or when an input argument is set to a value that does not make sense or is formatted incorrectly. Warnings on the other hand will be given if some ignorable part of the problem has an empty definition (`NULL`, `NA` or `NaN`), or if the interface has to convert or otherwise guess on an interpretation of input on a non-standard form. Errors will always interrupt the optimization process whereas warnings will not. Since warnings can hold valuable debugging information and may be important to notice, they are both printed in the log at the time they occurred and later summarized just before the interface returns.

Error messages works fine when you are interacting with the interface, but in automated optimization frameworks they are not easily handled. This is why a `response` is always returned as part of the result, when calling a function that may produce errors (see e.g. Figure 2). The `response` is a list containing a `code` and `msg`. When an error happens inside a function call to the MOSEK optimization library, the `code` is the response code returned by the function call[14] and `msg` is the corresponding error message. When an error happens within the interface, the `code` equals `NaN` and the `msg` is the error message which, in case of unexpected execution paths, may require technical knowledge to understand. When no errors are encountered, the `code` is zero, but beware that comparison operators such as `'=='` does not handle `NaN` very good. Using `identical` may be a better alternative.

`NB!` The interface may return only partially constructed output.
(always check the response code; e.g. *`success`* `<- identical(`*`response_code`*`, 0)`)

[14] Check out: mosek.com → Documentation → C API manual → Response codes.

▶ Interpreting the solution

The default optimizer for conic quadratic problems is the interior-point algorithm which only returns the interior-point solution (called *itr*).

The 'cqo1' Example (Part 2 of 3) The 'cqo' example was solved using the default optimizer (the interior-point algorithm) and contains the interior-point solution partly presented here.

As seen in Figure 18 the solution space of the problem was not empty (as it is primal feasible) and the problem was not unbounded (as it is dual feasible). In addition the optimizer was able to identify one of the optimal solutions.

Figure 18: Primal Solution (cqo1)

```
r$sol$itr
{
 $solsta
  [1] "OPTIMAL"
 $prosta
  [1] "PRIMAL_AND_DUAL_FEASIBLE"
 $skc
  [1] "EQ"
 $skx
  [1] "SB" "SB" "SB" "SB" "SB" "SB"
 $skn
  [1] "UL" "UL"
 $xc
  [1] 1
 $xx
  [1] 0.2609205 0.2609205 0.2390795 0.3689974 0.1690547 0.1690547
 ...
}
```

□

The solution you receive from the interface will contain the primal variable x (called ***xx***) and the activity of each constraint, ***xc***, defined by $x^c = Ax$. From the solution status (called ***solsta***) it can be seen how good this solution is, e.g. optimal, nearly optimal, feasible or infeasible. If the solution status is not as you expected, it might be that the problem is either ill-posed, infeasible or unbounded (dual infeasible). This can be read from the problem status (called ***prosta***). The solution and problem status are based on certificates found by the MOSEK optimization library[15], and should always be verified before the returned solution values are used (see Section 2.2.5).

[15] More details on problem and solution status keys available at:
mosek.com → Documentation → Optimization tools manual → Symbolic constants reference.

The solution also contains status key vectors for both variables, linear constraints and conic constraints (called ***skx***, ***skc*** and ***skn***). Each status key will be one of the following two-character strings.

`BS : Basic`
In basic (*bas*) solutions: The constraint or variable belongs to the basis of the corresponding simplex tableau.

`SB : Super Basic`
In interior-point (*itr*) and integer (*int*) solutions: The constraint or variable is in between its bounds.

`LL : Lower Level`
The constraint or variable is at its lower bound.

`UL : Upper Level`
The constraint or variable is at its upper bound.

`EQ : Equality`
The constraint or variable is at its fixed value (equal lower and upper bound).

`UN : Unknown`
The status of the constraint or variable could not be determined.

In addition to the primal variables just presented, the returned solutions also contain dual variables not shown here. The dual variables can be used for sensitivity analysis of the problem parameters and are related to the dual problem explained in Section 2.3.2.

2.3.2 Duality

The dual problem corresponding to the primal conic quadratic problem (2.7) defined in Section 2.3.1, is shown below in (2.12). Notice that the coefficients of the dual problem is the same as those used in the primal problem. Matrix A for example is still the constraint matrix of the primal problem, merely transposed in the dual problem formulation.

The dual problem is very similar to that of linear programming (Section 2.2.2), except for the added variable s_n^x belonging to the dual cone of $\mathcal{C}$ given by $\mathcal{C}^*$.

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c - (u^c)^T s_u^c + (l^x)^T s_l^x - (u^x)^T s_u^x + c_0 \\ \text{subject to} & A^T(s_l^c - s_u^c) + s_l^x - s_u^x + s_n^x = c\,, \\ & s_l^c, s_u^c, s_l^x, s_u^x \geq 0\,, \\ & s_n^x \in \mathcal{C}^*\,. \end{array} \tag{2.12}$$

The dual cone $\mathcal{C}^*$ always has the same number of variables as its primal $\mathcal{C}$, and is in fact easily expressed for the three convex cone types supported by MOSEK. For the $\mathbb{R}$-cone its dual is the single point in space with dual variables s_n^x all zero, such that (2.12) coincides with the dual linear program (2.4), when all primal variables belong to the $\mathbb{R}$-cone.

Just as easily it holds true that both the quadratic and rotated quadratic cones are self dual such that $\mathcal{C} = \mathcal{C}^*$. These facts ease the formulation of dual conic problems and have been exploited by the MOSEK optimization library for fast computations.

The 'cqo1' Example (Part 3 of 3) The part of the solutions to the 'cqo1' example that was previously omitted, is now shown in Figure 19. The dual variables *slc*, *suc*, *slx*, *sux* and *snx* correspond naturally to s_l^c, s_u^c, s_l^x, s_u^x and s_n^x in the dual problem.

Looking at the definition of the 'cqo1' problem (2.10), the first and only constraint is an equality constraint why its dual variables can either be read individually from its implicit lower an upper bound, *slc* and *suc*, or from the combined lower minus upper constraint dual variable (*slc* - *suc*). Notice that since none of the primal variables have upper bounds, the corresponding dual variables *sux* are all fixed to zeros. Further more, since all primal variables belong to a self-dual quadratic cone, all of the *snx* variables can attain values from the corresponding quadratic cones.

Figure 19: Dual Solution (cqo1)

```
r$sol$itr
{
 $solsta
  [1] "OPTIMAL"
 $prosta
  [1] "PRIMAL_AND_DUAL_FEASIBLE"
 ...
 $slc
  [1] 0.7071068
 $suc
  [1] 0
 $slx
  [1] 2.582619e-09 2.582619e-09 3.845566e-09 0.000000e+00 0.000000e+00
  [6] 0.000000e+00
 $sux
  [1] 0 0 0 0 0 0
 $snx
  [1] -0.7071068 -0.7071068 -1.4142136  1.0000000  1.0000000  1.0000000
}
```

□

2.3.3 Notes on Quadratic Programming (QP/QCQP)

Quadratic programming, or more generally all-quadratic programming, is a smaller subset of conic quadratic programming (CQP) that can easily be handled and solved using modern CQP solvers. The primal problem of a quadratic program can be stated as shown below in (2.13), where matrices Q_0, Q_1, and columns a, b and c are assumed to have compliant dimensions. Notice that this problem is convex, and thereby easily solvable, only if Q_0 and Q_1 are both positive semidefinite.

$$\begin{array}{lll} \text{minimize} & 0.5\, x^T Q_0 x + c^T x & \\ \text{subject to} & 0.5\, x^T Q_1 x + a^T x & \leq \quad b. \end{array} \tag{2.13}$$

While quadratic terms are added directly to the objective function and constraint matrix in quadratic programming, CQP formulations instead retain both a linear objective and a linear constraint matrix, closely resembling a linear program. All non-linearities, such as quadratic terms, are in this case formulated by themselves using quadratic cones. Many types of non-linearities can be modeled using these quadratic cones as seen in Appendix B, but in the case of (2.13), the result is two of the simplest possible rotated quadratic cones - one for the $x^T Q_0 x$ term and one for the $x^T Q_1 x$ term.

CQP formulations also have the advantage that there are no conditions on when a model is convex or not, because it always is. The transformation from quadratic programs to conic quadratic programs is based on a matrix factorization requiring Q_0 and Q_1 to be positive semidefinite, which is exactly the convexity requirement for QCQP's. This means that $x^T Q_0 x$ can be transformed only if $x^T Q_0 x$ is convex. In some sense, this transformation can be seen as a model-strengthening preprocessing step that only has to be done once. All further changes to the model can of course be made directly to the CQP.

The transformation of a quadratic program, whether it is QP or QCQP, to a conic quadratic program (CQP), can be seen in Appendix B.1.

2.4 Mixed Integer Programming

2.4.1 Solving MILP and MICQP problems

Mixed Integer Programming is a common and very useful extension to both linear and conic optimization problems, where one or more of the variables in the problem is required only to attain integer values. For the user this will only require a modest change to the model as these variables simply have to be pointed out, but to the underlying optimization library the problem increases in complexity for every integer variable added[16].

This happens because mixed integer optimization problems have to be solved using continuous relaxations and branching strategies to force integrality. Consequently, the running time of the optimization process will be highly dependent on the strength of the continuous relaxation to the problem formulation - that is, how far from the optimal mixed integer solution the problem formulation is when solved without the integrality requirement. Some suggestions to reduce the solution time are thus:

- Relax the termination criterion: In case the run time is not acceptable, the first thing to do is to relax the termination criterion (see Section 2.4.3)
- 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[17].

A change that the user will notice when starting to enforce integrality is that the notion of a dual problem is no longer defined for the problem at hand. This means that dual variables will no longer be part of the solution to the optimization problem, and that only the primal variables, constraint activities and problem/solution status reports, can be expected from the output structure returned by the interface.

[16] Check out: mosek.com → Documentation → Optimization tools manual → The optimizer for mixed integer problems.

[17] See for instance: L. A. Wolsey, Integer programming, *John Wiley and Sons*, 1998

The 'milo1' Example (Part 1 of 2) The way of requiring integer variables is the same regardless of whether we solve linear or conic quadratic problems. The following is an example of a simple mixed integer linear optimization problem, with two inequalities and two non-negative integer variables:

$$\begin{array}{lrcl} \text{maximize} & x_1 + 0.64x_2 & & \\ \text{subject to} & 50x_1 + 31x_2 & \leq & 250, \\ & 3x_1 - 2x_2 & \geq & -4, \\ & x_1, x_2 \geq 0 & \text{and integer.} & \end{array} \tag{2.14}$$

This is easily programmed in R using the piece code shown in Figure 20, where x_1 and x_2 are pointed out as integer variables.

Figure 20: Mixed Integer Optimization (milo1)

```
milo1 <- list(sense = "max")
milo1$c <- c(1, 0.64)
milo1$A <- Matrix(c( 50, 31 ,
                      3, -2 ),
                  nrow  = 2,
                  byrow = TRUE,
                  sparse= TRUE)
milo1$bc <-
  rbind(blc = c(-Inf,-4),
        buc = c(250,Inf))
milo1$bx <-
  rbind(blx = c(0,0),
        bux = c(Inf,Inf))
milo1$intsub <- c(1,2)
r <- mosek(milo1)
```

Figure 21: Output object (milo1)

```
r$sol$int
{
 $solsta
  [1] "INTEGER_OPTIMAL"
 $prosta
  [1] "PRIMAL_FEASIBLE"
 $skc
  [1] "UL" "SB"
 $skx
  [1] "SB" "LL"
 $xc
  [1] 250  15
 $xx
  [1] 5 0
}
```

□

The input arguments follow those of a linear or conic program with the additional identification of the integer variables (refer to Figure 2). The column vector *intsub* should simply contain indexes to the subset of variables for which integrality is required. For instance if x should be a binary {0,1}-variable, its index in the problem formulation should be added to *intsub*, and its bounds $0 \leq x \leq 1$ should be specified explicitly.

If executed correctly you should be able to see the log of the interface and optimization process printed to the screen. The output structure shown in Figure 21, will only include

an integer solution *int*, since we are no longer in the continuous domain for which the interior-point algorithm operates. The structure also contains the problem status as well as the solution status based on certificates found by the MOSEK optimization library[18].

2.4.2 Hot-starting

Hot-starting (also known as warm-starting) is a way to make the optimization process aware of a feasible point in the solution space which, depending on the quality of it (closeness to the optimal solution), can increase the speed of optimization. In mixed integer programming there are many ways to exploit a feasible solution and for anything but small sized problems, it can only be recommended to let the optimizer know about a feasible solution if one is available.

For many users the main advantage of hot-starting a mixed integer program will be the increased performance, but others may also find appreciation for the fact that the returned solution can only be better (or in worst-case the same) as the solution fed in. This is important in many applications where infeasible or low-quality solutions are not acceptable even when time is short. Heuristics are thus combined with hot-started mixed integer programs to yield a more reliable tool of optimization.

The 'milo1' Example (Part 2 of 2) For a small problem like the previously introduced *milo1*, that can be solved to optimality without branching, it is not really useful to hot-start. This example nevertheless illustrates the principle of how it can be done.

Figure 22: Hot-starting from initial guess (milo1)

```
# Define 'milo1' (not shown)

# Try to guess the optimal solution
myint <- list()
myint$xx <- c(5.0, 0.0)

# Hot-start the mixed integer optimizer
milo1$sol <- list(int=myint)
r <- mosek(milo1)
```

□

[18] More details on problem and solution status keys available at:
mosek.com → Documentation → Optimization tools manual → Symbolic constants reference.

2.4.3 Termination criteria

In general, it is impossible to find an exactly feasible and optimal solution to an integer optimization problem in a reasonable amount of time. On some practical problems this may be possible, but for the most cases it works much better to employ a relaxed definition of feasibility and optimality in the integer optimizer, in order to determine when a satisfactory solution has been located.

A candidate solution, i.e. a solution to the linearly relaxed mixed integer problem, is said to be integer feasible if the criterion

$$\min\left(|x_j| - \lfloor |x_j| \rfloor, \lceil |x_j| \rceil - |x_j|\right) \;\leq\; \max\left(\delta_1, \delta_2 |x_j|\right) \tag{2.15}$$

is satisfied for all variables in the problem, $j \in J$. Hence, such a solution is defined as feasible to the mixed integer problem. Whenever the optimizer locates an integer feasible solution it will check if the criterion

$$|\overline{z} - \underline{z}| \;\leq\; \max\left(\delta_3, \delta_5 \max\left(1, |\overline{z}|\right)\right) \tag{2.16}$$

is satisfied. If this is the case, the integer optimizer terminates and reports the integer feasible solution as an optimal solution. Please note that $\overline{z}$ is a valid upper bound, and $\underline{z}$ a valid lower bound, to the optimal objective value z^*. In minimization problems, $\underline{z}$ normally increases gradually during the solution process, while $\overline{z}$ decreases each time a better integer solution is located starting at the warm-started or heuristically preprocessor generated solution value.

The δ tolerances are specified using parameters and have default values as shown below.

Tolerance	Parameter name (type: dparam)	Default value
δ_1	MIO_TOL_ABS_RELAX_INT	10^{-5}
δ_2	MIO_TOL_REL_RELAX_INT	10^{-6}
δ_3	MIO_TOL_ABS_GAP	0
δ_4	MIO_TOL_REL_GAP	10^{-4}
δ_5	MIO_TOL_NEAR_TOL_ABS_GAP	0
δ_6	MIO_TOL_NEAR_TOL_REL_GAP	10^{-3}

Table 2.17: Integer optimizer tolerances.
Default values are subject to change from MOSEK v6.0 shown here.

If an optimal solution can not be located within a reasonable amount of time, it may be advantageous to employ a relaxed termination criterion after some time. Whenever the integer optimizer locates an integer feasible solution, and has spent at least the number of seconds defined by the double-typed parameter MIO_DISABLE_TERM_TIME on solving the problem, it will check whether the criterion

$$|\overline{z} - \underline{z}| \ \leq \ \max\left(\delta_5, \delta_6 \max\left(1, |\overline{z}|\right)\right) \tag{2.18}$$

is satisfied. If it is satisfied, the optimizer will report that the candidate solution is **near optimal** and then terminate. Please note that the relaxed termination criterion is not active by default, as the delay (MIO_DISABLE_TERM_TIME) is -1 by default.

Given such a delay, the integer optimizer also offers three additional parameters to stop the optimizer prematurely once the specified delay have passed. Beware that these parameters, in the table below, are not able to guarantee any kind of near optimality. Using only MIO_MAX_NUM_SOLUTIONS will, however, ensure that at least one an integer feasible solution exists. With a default value of -1, these parameters are all ignored until explicitly defined.

Parameter name (type: iparam)	**Maximum number of ...**
MIO_MAX_NUM_BRANCHES	Branches allowed.
MIO_MAX_NUM_RELAX	Relaxations allowed.
MIO_MAX_NUM_SOLUTIONS	Integer feasible solutions allowed.

Finally, if the relaxed termination criteria also fail to be satisfied, the double-typed parameter OPTIMIZER_MAX_TIME can be used to define the number of seconds before it is accepted, that the problem was not tractable in its current formulation. See Section 2.5 on how to specify all these parameters.

2.5 Parameter settings in MOSEK

The MOSEK optimization library offers a lot of customization for the user to be able to control the behavior of the optimization process or the returned output information. All parameters have been documented on the homepage of MOSEK[19], and are all supported by this interface. Only a few is mentioned here.

Notice that the "MSK_" prefix, required by the MOSEK C API, can be ignored for string-typed parameter values in this interface. Similarly the "MSK_IPAR_", "MSK_DPAR_" and "MSK_SPAR_" prefixes for parameter names, have been removed in favor of the `iparam`, `dparam` and `sparam` structures. Also in this interface, parameters are case-insensitive. This means that parameter names and string-typed parameter values can be written in both upper and lower case.

The 'LOG'-Parameter Example This is a logging parameter controlling the amount of information printed to the screen from all channels within the MOSEK optimization library. The value of the parameter can be set to any integer between 0 (suppressing all information) to the R value *Inf* (releasing all information), and the default is 10.

Revisiting the 'lo1' example from Section 2.2.1, we can now try to silence the optimization process as shown below.

Figure 23: Suppressing the optimization log with parameter 'LOG'

```
lo1 <- list(sense = "max")
lo1$c <- c(3,1,5,1)
lo1$A <- Matrix(c(3,1,2,0,
                  2,1,3,1,
                  0,2,0,3), nrow=3, byrow=TRUE, sparse=TRUE)
blc <- c(30,15,-Inf)
buc <- c(30,Inf,25);   lo1$bc <- rbind(blc, buc);
blx <- c(30,15,-Inf)
bux <- c(30,Inf,25);   lo1$bx <- rbind(blx, bux);
lo1$iparam <- list(LOG = 0);
r <- mosek(lo1);
```

Notice that errors and warnings from the interface itself will not be affected by this parameter. These are controlled separately by the option *verbose*.

□

[19] Check out: mosek.com → Documentation → C API manual → Parameter reference.

Adding parameters to the input arguments of a convex optimization problem is, as seen, straightforward (refer to Figure 2). The list *iparam* should specify its variables with a name, matching that of an integer-typed parameter, and a value, being the corresponding parameter value. Similarly the list *dparam* is for double-typed parameters, and *sparam* is for string-typed parameters. Notice that reference-strings to an enum (a small set of named possibilities) is also integer-typed, as the input reference-strings are automatically converted to indexes by the interface.

The 'OPTIMIZER'-Parameter Example This parameter controls which optimizer used to solve a specific problem. The default value is "OPTIMIZER_FREE" meaning that MOSEK will try to figure out what optimizer to use on its own.

In this example we shall try to solve the 'lo1' example from Section 2.2.1 again, only this time using the dual simplex method. This is specified by setting the parameter to the enum reference-string value "OPTIMIZER_DUAL_SIMPLEX" as shown.

Figure 24: Selecting the dual simplex method with parameter 'OPTIMIZER'

```
lo1 <- list(sense = "max")
lo1$c <- c(3,1,5,1)
lo1$A <- Matrix(c(3,1,2,0,
                  2,1,3,1,
                  0,2,0,3), nrow=3, byrow=TRUE, sparse=TRUE)
blc <- c(30,15,-Inf)
buc <- c(30,Inf,25);   lo1$bc <- rbind(blc, buc);
blx <- c(30,15,-Inf)
bux <- c(30,Inf,25);   lo1$bx <- rbind(blx, bux);
lo1$iparam <- list(OPTIMIZER = "OPTIMIZER_DUAL_SIMPLEX");
r <- mosek(lo1)
```

□

Setting a parameter to `NULL` will remove it from the list according to the R language definition. Setting a parameter to `NA` or `NaN`, will on the other hand keep it on the list, only to be ignored by the interface with warnings confirming that this took place. Errors will be generated when a parameter name is not recognized or when the value defined for it is not within its feasible range.

2.6 Exporting and Importing optimization models

This section concerns the export and import of optimization models in all standard modeling formats (e.g. 'lp', 'mps' and 'opf') and the MOSEK binary format 'mbt'. These files can be converted to and from the problem descriptions (R-variables) that the `mosek` function can accept (refer to Figure 2).

	`lp`	`mps`	`opf`	`mbt`
Linear problem	✓	✓	✓	✓
Conic constraint		✓	✓	✓
Integer variable	✓	✓	✓	✓
MOSEK Parameter		✓	✓	✓
Solution			✓	✓

Figure 25: Supported file formats recognized by their file extension.

As seen in Figure 25, some of these modeling formats have more limitations than others. The 'lp' format does not support conic constraints, parameter settings or solutions. The 'mps' format also does not support solutions, but has a wider usage. Only the 'opf' and 'mbt' formats handles the full problem description. In all formats supporting parameter settings, either none or the entire list of all parameters in MOSEK are exported. This will lead to large problem descriptions when parameters are imported from a file. To save and reload the constructed R-variables exactly as they are, please use the `save` and `load` commands part of the R language definition.

2.6.1 The write and read functionality

▶ Exporting through the `mosek_write` function

A problem description can be exported with the `mosek_write` command taking the problem variable and destination as input, and returning a response that can be used to verify success. The destination should be a filepath with a file extension matching the modeling format to be used. If the file extension is not recognized as one of the supported modeling formats, the 'mps' format is used by default. Note, however, that this only affects the contents of the file and does not change the file extension to 'mps'. The specified filepath is overwritten if it already exists, and otherwise created.

Exporting the 'lo1' Example Revisiting the 'lo1' example from Section 2.2.1, we can now try to export the optimization model to the 'opf' format. In the current example we export the problem description to a file called 'lo1.opf' in the current working directory.

Beware that this file is overwritten if it already exists! Afterwards, using the standard `file.show` command, the file is displayed in the R-console.

Figure 26: Exporting a Linear Optimization Problem

```
lo1 <- list(sense = "max")
lo1$c <- c(3,1,5,1)
lo1$A <- Matrix(c(3,1,2,0,
                  2,1,3,1,
                  0,2,0,3), nrow=3, byrow=TRUE, sparse=TRUE)
blc <- c(30,15,-Inf)
buc <- c(30,Inf,25);         lo1$bc <- rbind(blc, buc);
blx <- c(0,0,0,0)
bux <- c(Inf,10,Inf,Inf);    lo1$bx <- rbind(blx, bux);

r <- mosek_write(lo1, "lo1.opf")
file.show("lo1.opf")
```

□

In the example we did not have either parameters or initial solutions defined in the problem description, but if we had, the options `usesol` and `useparam` could be used to indicate whether to write them to the file or not. By default, `usesol` is true and `useparam` is false. Parameters are thus usually omitted in exported models. Note that since it does not make sense for an initial solution to specify a solution or problem status, the solutions exported as part of a problem description will always have unknown statuses. The export functionality in the `mosek` function behaves differently in this regard (see Section 2.6.2) .

▶ Importing through the `mosek_read` function

A problem description can be imported with the `mosek_read` command taking the filepath as input. If the file extension does not match one of the supported modeling formats, the 'mps' format is assumed and errors are given if the contents of the file does not respect this syntax.

Importing the 'lo1' Example (Part 1 of 2) We will now try to import the 'lo1' optimization model that was exported in Figure 26. In the current example we import from a file called 'lo1.opf' in the current working directory. Notice that we use the options `usesol` and `useparam` to indicate that we do not wish to read solutions from the file, but

are interested in the parameter settings. Checking that the response code is fine, we print a message and assign the 'lo1' variable. Please note the size of the fields *iparam*, *dparam* and *sparam* of the imported problem description.

Figure 27: Importing a Linear Optimization Problem

```
r <- mosek_read("lo1.opf", list(usesol=FALSE, useparam=TRUE))

if (identical(r$response$code, 0)) {
  print("Successfully read the optimization model")
  lo1 <- r$prob
}
```

□

The options `usesol` and `useparam` behaves as for the `mosek_write` function, and can be used to specify whether to read the solutions and parameter settings from the file. By default, `usesol` is true and `useparam` is false. Note that for all parameters not defined in the file, the default value in the MOSEK optimization library is assigned instead. This means that the returned problem description will always be quite large when parameters are imported.

Another option of special interest is the string-typed `matrixformat` which can specify the format of the imported sparse constraint matrix called `A` in Figure 2. The recognized values for this option are shown below with `pkgMatrix:COO` (abbreviated `COO`) being the default. All string values are case insensitive.

`COO` / `pkgMatrix:COO`
The sparse coordinate format (aka triplet format) from package 'Matrix'.

`CSC` / `pkgMatrix:CSC`
The compressed sparse column format from package 'Matrix'.

`simple:COO`
The sparse coordinate format (aka triplet format) based on list-objects.

When setting `matrixformat` to the string-value `simple:COO`, the constraint matrix will be a named list with scalars ***nrow*** and ***ncol*** telling respective the number of rows and columns of matrix `A`, and lists ***i***, ***j*** and ***v*** telling the ij-coordinates of each value in v. All these matrix formats are allowed in the problem description given as input to the `mosek` and `mosek_write` functions.

Importing the 'lo1' Example (Part 2 of 2) We will again try to import the 'lo1' optimization model that was exported in Figure 26, only this time ignoring all parameters and with the `matrixformat` option set to `simple:COO`. This is shown in Figure 28 where a file called 'lo1.opf' is imported from the current working directory. The structure of the imported constraint matrix is shown in Figure 29.

Figure 28: Importing a Linear Optimization Problem with 'simple:COO'

```
r <- mosek_read("lo1.opf", list(matrixformat="simple:COO"))
```

Figure 29: Importing a Linear Optimization Problem with 'simple:COO'

```
r$prob$A
{
 $nrow
  [1] 3
 $ncol
  [1] 4
 $i
  [1] 1 2 1 2 3 1 2 2 3
 $j
  [1] 1 1 2 2 2 3 3 4 4
 $v
  [1] 3 2 1 1 2 2 3 1 3
}
```

□

2.6.2 Writing immediately before/after optimizing

As an alternative to `mosek_write`, the `mosek` function itself can be used to export the interpreted optimization model and identified solution to in all standard modeling formats (e.g. 'lp', 'opf', 'mps' and 'mbt'). Note that the here mentioned options only apply to the `mosek` function.

The option 'writebefore' will export the model exactly as MOSEK have received it, immediately before it starts the optimization procedure. This can for instance be used to inspect the problem description in formats that the user may be more familiar with, to see if what was intended, matched what had been expressed. This option can also be used as a log-file of the last solved model, such that if the MOSEK optimization library somehow behaves unexpected, the input model can easily be reported.

The option 'writeafter' will export the model and identified solutions, immediately after the optimization procedure has ended. In contrast to the `mosek_write` function, these

solutions are not seen as initial solutions, and do have the correct solution and problems statuses exported with them. Notice that only the 'opf' and 'mbt' formats support solutions (Figure 25), and that all other modeling formats would thus result in files similar to those written by 'writebefore'.

Finally, it may be of interest, that the options `usesol` and `useparam` can also be used in this context. These options were explained in the description of `mosek_write` and `mosek_read` previously in this chapter.

2.7 Frequently Asked Questions

2.7.1 How to make Rmosek silent?

The verbosity of the interface - that is, how many details that are printed - can be regulated by the option *verbose* (see e.g. Figure 2, Section 2.1). The default value is 10, but as of now there only exists four message priorities: Errors=1, MOSEK=2, Warnings=3 and Info=4. Setting *verbose* to the value of 1, will for example only allow the interface to speak when errors are encountered. Setting *verbose* to 2, will also open for the output stream from the MOSEK optimization library. Do note that verbosity levels less than three will ignore warnings which may hold valuable information.

`NB!` Setting the option *verbose* = 0 will completely silence everything...
(even when errors and warnings have occurred)

The verbosity of the MOSEK optimization library, can be regulated by a large range of parameters[20]. In many cases the integer-typed 'LOG'-parameter provides sufficient control over the logging details and can be set as shown in Figure 23, Section 2.5.

2.7.2 How to force a premature termination?

While the MOSEK optimization library is running, the interface will once in a while check for the keyboard sequence <CTRL> + <C> used by the user to signal that a premature exit would be preferable. Once the signal is registered the error message "Interruption caught, terminating at first chance..." will be printed to the screen and the best solution found so far will be processed and returned.

2.7.3 How to change the termination criteria?

The list of termination criteria is long, spanning over absolute and relative tolerances, maximum iterations, time limits and objective bounds[21]. Each criterion can be modified through a parameter (see Section 2.5 for help on this). As an example, the integer-typed parameters *SIM_MAX_ITERATIONS* and *INTPNT_MAX_ITERATIONS* respectively controls the maximum numbers of simplex and interior-point iterations, while the double-typed parameter *OPTIMIZER_MAX_TIME* sets the overall time limit in seconds. For mixed integer optimization problems, more information can be found in Section 2.4.3.

[20] Check out: mosek.com → Documentation → C API manual → Logging parameters.
[21] Check out: mosek.com → Documentation → C API manual → Termination criterion parameters.

2.7.4 How to find the objective value and other information items?

Extra details from a call to the *mosek* function, can be requested using the options *soldetail* and *getinfo* (see e.g. Figure 2, Section 2.1).

Currently there exists three levels of solution detail: NoExtras=0 (the default), ObjectiveValues=1 and MaxInfeas=2. Setting *soldetail* to at least 1, will add the primal objective value ***pobjval*** and the dual objective value ***dobjval*** (or another primal objective bound ***pobjbound*** in case of mixed-integer optimization.). Setting *soldetail* to at least 2, furthermore adds the list of maximal infeasibilities ***maxinfeas***. This list covers the primal inequality, equality and conic constraints (***pbound***, ***peq*** and ***pcone***), the dual inequality, equality and conic constraints (***dbound***, ***deq*** and ***dcone***), as well as the fractionality of primal integer variables ***int***.

Setting *getinfo* to the boolean value `TRUE`, will add two lists with information items extracted from the MOSEK optimization library. One contains all available integer-typed information ***iinfo***, based on items with a prefix of MSK_IINF_ and MSK_LIINF_ in the MOSEK C API[22]. The other list contains all available double-typed information ***dinfo***, based on items with a prefix of MSK_DINF_ in the MOSEK C API.

2.7.5 How to report a bug or get further help?

As was written in the scope of this user guide, this interface is currently not part of the MOSEK product line and no support can thus be expected to be offered by MOSEK. Instead, we invite all users of the R-to-MOSEK Interface to join our ***central MOSEK discussion group***[23]. This group is particularly intended for academics using MOSEK in their research.

In pursuit of answers, the user guide is a good place to start. Advanced users can also take a look at the source code published along with this interface, and are welcome to fix bugs, add features and make customizations.

Bug reports and suggestions - or even better, bug fixes and working improvements - can be sent to the main project developer Henrik Alsing Friberg, `haf@mosek.com`. The use of the discussion group is, however, the preferred form of contact as it reaches a broader user community which may be able to help or provide feedback to your posts.

[22] Check out: mosek.com → Documentation → C API manual → API constants.
[23] Check out: groups.google.com/group/mosek

3 Advanced topics

3.1 Parallel optimization

3.1.1 The multicore package

The R package called 'multicore', provides functions for parallel execution of R code on machines with multiple cores or CPUs. Windows is not currently supported, but the package should work on most UNIX-alike platforms.

The multicore package works by copying the full memory state of the R session to new processes. While this seems like a large overhead, in practice, the copy is delayed until modification assuring a smooth parallel execution. The downside is that this low-level memory state copy is not safe for all types of resources. As an example, parallel interactions with the GUI or on-screen devices can cause the R session to crash. It is thus recommended only to use the multicore package in console R.

In the Rmosek package a license is an externally acquired resource, and attempts to simply copy the memory state of this resource will provoke a session crash. Thus, licenses should always be released before the time of parallelization.

`NB!` Always call `mosek_clean()` before a parallelizing operator.
(failure to do so is likely to provoke session crashes)

A consequence of this is that each new process will be using a separate license. That is, your license system should allow 8 licenses to be checked out simultaneously, if you wish to solve 8 optimization problems in parallel. Please note that unlimited academic and commercial licenses are available at MOSEK.

Solving the 'lo1' Example in Parallel (Part 1 of 2) The following is an example of a linear optimization problem with one equality and two inequality constraints:

$$\begin{array}{lrcrcrcrcr} \text{maximize} & 3x_1 & + & 1x_2 & + & 5x_3 & + & 1x_4 & & \\ \text{subject to} & 3x_1 & + & 1x_2 & + & 2x_3 & & & = & \lambda, \\ & 2x_1 & + & 1x_2 & + & 3x_3 & + & 1x_4 & \geq & 15, \\ & & & 2x_2 & & & + & 3x_4 & \leq & 25, \end{array} \tag{3.1}$$

having the bounds

$$\begin{array}{rcrcl} 0 & \leq & x_1 & \leq & \infty, \\ 0 & \leq & x_2 & \leq & 10, \\ 0 & \leq & x_3 & \leq & \infty, \\ 0 & \leq & x_4 & \leq & \infty. \end{array} \tag{3.2}$$

This problem is infeasible for $\lambda < 40/9 \approx 4.44$. As shown in Figure 30, the problem is solved by 4 parallel processes for 10 different values of λ stepping from 3 to 30.

Figure 30: Parallel Optimization (permutations of lo1)

```
require(Rmosek)
lo1 <- list(sense = "max")
lo1$c <- c(3,1,5,1)
lo1$A <- Matrix(c(3,1,2,0,
                  2,1,3,1,
                  0,2,0,3), nrow=3, byrow=TRUE, sparse=TRUE)
blc <- c(30,15,-Inf)
buc <- c(30,Inf,25);         lo1$bc <- rbind(blc, buc)
blx <- c(0,0,0,0)
bux <- c(Inf,10,Inf,Inf);    lo1$bx <- rbind(blx, bux)

#
# Solve permutations of this problem in parallel
#
require(foreach); require(doMC)
registerDoMC(cores=4)

mosek_clean()
r <- foreach(lambda = seq(3,30,length=10)) %dopar%
{
  lo1$bc[,1] <- c(lambda, lambda)
  return( mosek(lo1) )
}
```

Note that we use a loop construct from the 'foreach' package, with the package 'doMC' (depending on package 'multicore') registered as the parallel back-end. This is easier than using the multicore package directly. Please see the documentation of the foreach package for more details on how to use this.

Furthermore, note the `mosek_clean()` call immediately before the `%dopar%` operator which is highly recommended for a smooth parallel execution.

□

▶ Controlling the output stream

One of the issues with parallel programming is that if multiple processes use the same output stream, the messages will be mixed up and hard to tell apart. One solution is to use the `capture.output` function part of the standard R installation, and write the output stream of each thread to a separate file.

Solving the 'lo1' Example in Parallel (Part 2 of 2) In this example we use the `capture.output` function to write the output of each parallel process to files called `RESULTS_FILE_`λ for the specific λ in use. These files will placed in the current working directory of the R session.

Figure 31: Parallel Optimization (permutations of lo1)

```
# Define problem and register parallel back-end as before..

capture.output.helper <- function(file, code) {
  capture.output(file=file, res <- code(), mosek_clean(), print("Done"))
  return(res)
}

mosek_clean()
r <- foreach(lambda = seq(3,30,length=10)) %dopar% capture.output.helper(
  file = paste("RESULTS_FILE_", lambda, sep=""),
  code = function()
  {
    lo1$bc[,1] <- c(lambda, lambda)
    return( mosek(lo1) )
  }
)
```

The defined `capture.output.helper` solves two problems. It makes sure that the return value is correctly propagated to the foreach-loop, as opposed to the standard `capture.output` function ignoring return values. It also calls `mosek_clean()`, to capture any cleaning messages from Rmosek, which would otherwise appear on screen when the parallel process is terminated outside the control of `capture.output`.

□

3.2 Quadratic Convex Optimization

3.2.1 Solving QP problems

Quadratic Convex Optimization is a generalization of linear programming for which quadratic terms can be added to the objective function. *Convex* here emphasize that the entire formulation should define a convex optimization problem.

$$\begin{array}{lccccc} \text{minimize} & & & 0.5x^T Qx + c^T x + c_0 & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c\,, \\ & l^x & \leq & x & \leq & u^x\,. \end{array} \tag{3.3}$$

Two facts should be noted, before engaging in quadratic convex optimization:

- The problem is convex only if Q is positive semidefinite (all eigenvalues non-negative), which can be numerically hard to verify for large Q with zero or near-zero eigenvalues.
- The MOSEK optimization library does not support the combination of quadratic convex optimization with conic constraints.

If the problem at hand involves quadratic objective terms, quadratic convex optimization may very well be the best way to solve it. Nevertheless, it is easy to try the conic quadratic formulation of these problems, for which the optimizer will sometimes perform even better (see Section 2.3.3).

The 'qo1' Example The following is an example of a linear constrained problem with a convex quadratic objective.

$$\begin{array}{lcccc} \text{minimize} & 0.5\left(2x_1^2 - 2x_3x_1 + 0.2x_2^2 + 2x_3^2\right) - x_2 & & & \\ \text{subject to} & x_1 + x_2 + x_3 & \geq & 1\,, \\ & x_1, x_2, x_3 \geq 0\,. & & & \end{array} \tag{3.4}$$

The quadratic objective terms are easily appended on top of the linear formulation as shown in the following R code (Figure 32).

Figure 32: Quadratic Convex Optimization (qo1)

```
qo1 <- list()
qo1$sense <- "min"
qo1$c <- c(0,-1,0)
qo1$A <- Matrix(c(1,1,1), nrow=1, byrow=TRUE, sparse=TRUE)
qo1$bc <- rbind(blc = 1,
                buc = Inf)
qo1$bx <- rbind(blx = rep(0,3),
                bux = rep(Inf,3))

qo1$qobj <- list(i = c(1,  3,   2, 3),
                 j = c(1,  1,   2, 3),
                 v = c(2, -1, 0.2, 2))

r <- mosek(qo1)
```

The objective term $-2x_3x_1$ has been specified with half the coefficient, i.e. -1, in the field `qobj`. Note that this is in contrary to squared variables, e.g. $2x_1^2$, as explained in the text. □

From this example the input arguments for a quadratic convex program (3.3) follow easily (refer to Figure 2, Section 2.1). The linear part of the objective function and constraints, as well as the constraint and variable bounds, should all be specified as for linear programs (see Section 2.2). The only addition to this is the list called ***qobj*** containing the triplets of row indexes `i`, column indexes `j` and values `v` that define Q. The number of rows and columns of matrix Q should not be specified, as these dimensions are already known from the number of variables in the linear part of the problem.

The inputted Q is always assumed symmetric, and note that this is without loss of generality. This is efficient as only the lower triangular part of Q thus needs to be specified. Setting a coefficient at the lower triangular position (i,j) will, by symmetry, imply that the same coefficient is also present at position (j,i). In order to add the objective term $-2x_3x_1$, one would see it as $-x_3x_1 - x_1x_3$ and only input a coefficient of -1 to row 3 of column 1.

3.3 Separable Convex Optimization

3.3.1 Solving SCOPT problems

Separable Convex Optimization is a generalization of linear programming for which arbitrary unary operators can be added to the objective function and constraints. *Separable* means that each operator should depend on one variable only, and *Convex* means that the entire formulation should define a convex optimization problem.

$$\begin{array}{lccccc} \text{minimize} & & & z(x) + c^T x + c_0 & & \\ \text{subject to} & l^c & \leq & w(x) + Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \end{array} \tag{3.5}$$

where

$$\begin{aligned} z(x) &= \sum_{j=1}^{n} z_j(x_j) \quad , \quad z_j : \mathbb{R} \to \mathbb{R}. \\ w(x) = \begin{pmatrix} \vdots \\ w_i(x) \\ \vdots \end{pmatrix} \quad , \quad w_i(x) &= \sum_{j=1}^{n} w_{ij}(x_j) \quad , \quad w_{ij} : \mathbb{R} \to \mathbb{R}. \end{aligned} \tag{3.6}$$

Three facts should be noted, before engaging in separable convex optimization:

- Since the operators $z_j(x_j)$ and $w_{ij}(x_j)$ are arbitrarily defined, the optimizer for separable convex optimization has less knowledge, and can be expected to perform worse, than the optimizer for conic quadratic programs.
- No standard modeling file-format exists for separable convex optimization. (MOSEK has its own 'sco' format defined for a subset of operators)
- The MOSEK optimization library does not support the combination of separable convex optimization with integer variables or conic constraints.

If the problem at hand involves exponential or logarithmic functions, separable convex optimization may be a good and efficient choice. Note, however, that undefined behaviour can be observed if the problem is not convex (MOSEK will try to detect this). If the problem does not involve involves exponential or logarithmic functions, it may very well be possible to reformulate the optimization problem as a conic quadratic program (see e.g. Section 2.3 and Appendix B).

For the sake of performance, and the usability of the 'sco' file-format, callback functionality to custom R evaluation code have not been supported. Instead, a list of common operators with customizable coefficients have been added to this interface, making it both fast and easy to use. The reader is encouraged to contact us, in case this list of operators is found to be insufficient.

With constants $f \in \mathbb{R}$, $g \in \mathbb{R}$ and $h \in \mathbb{R}$, the interface currently allows:

Operator	**Definition**	**Domain**
Entropy	$fx\ln(x)$	$0 < x < \infty$
Exponential	$f\exp(gx+h)$	$-\infty < x < \infty$
Logarithm	$f\ln(gx+h)$	If $g > 0$: $-h/g < x < \infty$ Otherwise: $-\infty < x < -h/g$
Power	$f(x+h)^g$	If $g > 0$ and integer: $-\infty < x < \infty$ If $g < 0$ and integer: choose this $-h < x < \infty$ or this $-\infty < x < -h$ Otherwise: $-h < x < \infty$

Figure 26: Supported unary operators.

The 'sco1' Example The following is an example of a problem with a non-linear objective optimized over a linear constraint and the interior of a unit circle.

$$\begin{array}{lrcl} \text{minimize} & x_1 - \ln(x_3) & & \\ \text{subject to} & x_1^2 + x_2^2 & \leq & 1\,, \\ & x_1 + 2x_2 - x_3 & = & 0\,, \\ & x_3 \geq 0\,. & & \end{array} \tag{3.7}$$

The objective contains one ***log***arithm operator, and the first constraint contains two ***pow***er operators. These operators are easily appended on top of the linear formulation as shown in the following R code (Figure 33).

Figure 33: Separable Convex Optimization (sco1)

```
sco1 <- list(sense = "min")
sco1$c <- c(1,0,0)
sco1$A <- Matrix(c(0, 0, 0,
                   1, 2, -1), nrow=2, byrow=TRUE, sparse=TRUE)

sco1$bc <- rbind(blc = c(-Inf,0),
                 buc = c(1,0))
sco1$bx <- rbind(blx = c(-Inf,-Inf,0),
                 bux = rep(Inf,3))

NUMOPRO <- 1; NUMOPRC <- 2;
opro <- matrix(list(), nrow=5, ncol=NUMOPRO)
oprc <- matrix(list(), nrow=6, ncol=NUMOPRC)
  rownames(opro) <- c("type","j","f","g","h")
  rownames(oprc) <- c("type","i","j","f","g","h")
opro[,1] <- list("LOG", 3, -1.0, 1.0, 0.0)
oprc[,1] <- list("POW", 1, 1, 1.0, 2.0, 0.0)
oprc[,2] <- list("POW", 1, 2, 1.0, 2.0, 0.0)
sco1$scopt <- list(opro=opro, oprc=oprc)

r <- mosek(sco1)
```

□

From this example the input arguments for a separable convex program (3.5) follow easily (refer to Figure 2, Section 2.1). The linear part of the objective function and constraints, as well as the constraint and variable bounds, should all be specified as for linear programs (see Section 2.2). The only addition to this is the list called ***scopt*** containing the list-typed operator matrices ***opro*** (for objective) and ***oprc*** (for constraints).

The operator matrices have a column for each operator and a row for each descriptive element. The *opro* matrix have five rows called $\{type, j, f, g, h\}$, while the *oprc* matrix have six rows called $\{type, i, j, f, g, h\}$. Row ***type*** should specify the operator type in a string, being either entropy "ENT", exponential "EXP", logarithm "LOG" or power "POW". Row ***i*** (not in *opro*) should specify the index of the constraint to which the non-linear term should be added. Row ***j*** should specify the variable index of the operator. Rows ***f***, ***g*** and ***h*** should specify the coefficients of the operator (see Figure 26).

Examples are:

Terms in opro	type	j	f	g	h
In objective: $\exp(1.4x_1 + 1.1)$	"EXP"	1	1.0	1.4	1.1
In objective: $\ln(1.5x_2 + 1.2)$	"LOG"	2	1.0	1.5	1.2
In objective: $2.1\sqrt{x_3}$	"POW"	3	2.1	0.5	0.0

Terms in oprc	type	i	j	f	g	h
In constraint one: $0.1x_1 \ln(x_1)$	"ENT"	1	1	0.1	NA	NA
In constraint two: $(x_1 + 2.0)^{1.75}$	"POW"	2	1	1.0	1.75	2.0

Note that the definition of the entropy operator (see Figure 26), was the only operator defined without g and h. Thus, for entropy operators, these two unused rows in the operator matrix can be set to either zero or any empty definition (`NULL`, `NA` or `NaN`).

3.3.2 Convexity, differentiability and safe bounds

▶ Ensuring convexity

The feasible solution space of a separable convex optimization problem is required to be convex. If a simple set of convexity-preserving rules are complied with, the problem may further be called disciplined convex. Disciplined convexity is not required in this interface, but will be checked to help the user identify modeling issues.

Disciplined convexity means that the expression $g_i(x)$ of a constraint $l_i^c \leq g_i(x) \leq u_i^c$, is required to be concave if the lower bound is finite (i.e. not `-Inf`), and required to be convex if the upper bound is finite (i.e. not `Inf`). This also implies that the expression should be linear if both the lower and upper bound is defined with finite values.

In case of separable convex optimization, this means that each unary operator of an expression - required to be convex (resp. concave) - is used only in its convex (resp. concave) domain. A problem which is disciplined convex is guaranteed to be convex.

Operator	Definition	Convexity (domains defined in Figure 26)
Entropy	$fx \ln(x)$	Convex in domain.
Exponential	$f \exp(gx + h)$	Convex in domain.
Logarithm	$f \ln(gx + h)$	Concave in domain.
Power	$f(x + h)^g$	If g is even integer: convex in domain. If g is odd integer: concave ($x < -h$), convex ($-h < x$) If $0 < g < 1$: concave in domain. Otherwise: convex in domain.

Figure 27: Convexity of unary operators assuming $f > 0$.
Convexities are exactly opposite for $f < 0$.

▶ Ensuring differentiability with safe bounds

Figure 26 displayed the domains in which each operator was well-defined and differentiable. If the MOSEK optimizer for some reason tried to evaluate one of these operators outside of its domain, an error would occur and the optimization would terminate. Since MOSEK may not be able to guess an initial feasible solution to the problem, it may actually have to work its way through the infeasible region of the problem.

Essentially, operators may be evaluated in points where the *constraint bounds* are not satisfied. To avoid causing the optimizer to terminate due to evaluations outside operator domains, *variable bounds* should be specified such that operator domains are always satisfied. This is called safe bounds, and will work because the MOSEK optimizer is guaranteed to satisfy variable bounds in each iteration.

3.3.3 Importing and exporting modelfiles

In case the problem description contains separable convex operators, the string-typed option `scofile` must be used to specify the filepath to which these operators should be written. Failing to specify this option when calling `mosek_write` (see Section 2.6) will result in an error. It is recommended to use the 'sco' file extension, but the interface will allow any (even a missing one) to be used without warning.

Exporting the 'sco1' Example Revisiting the 'sco1' example from Section 3.3.1, we can now try to export the optimization model to the 'opf' format with operators in an 'sco' file. In particular, the current example will export the problem description to respectively 'sco1.opf' and 'sco1.sco' in the current working directory. Beware that these files are overwritten if they already exists! Afterwards, using the standard `file.show` command, the files are displayed in the R-console.

Figure 34: Exporting a Separable Convex Optimization Problem

```
# Define 'sco1' (not shown)

r <- mosek_write(sco1, "sco1.opf", list(scofile="sco1.sco"))
file.show("sco1.opf", "sco1.sco")
```

□

The same option, 'scofile', can be used when calling `mosek_read` (see Section 2.6). This function will read in a problem description of a separable convex optimization problem from the location of two files. These two files can also be read independently of each other.

4 Developers corner

4.1 Introduction

In the project's *src* folder you can find the header and source files (**.h* and **.cc*) containing the C++ code of the R-to-MOSEK interface. Moreover, the *Rmosek.cc* file contains the functions that are callable from R.

This code has been written and commented so that it should be easy to understand how the interface handles the input and output structures, prints to the screen and communicates with the MOSEK optimization library. Four important references proved to be useful in the development of this interface.

- The MOSEK C API Reference[24]
- R Language Definition[25]
- Writing R Extensions [25]
- R Installation and Administration[25]

The program is free software and has been released under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 2.1 of the License, or any later version. You are welcome to extend upon this release and modify the interface to your personal needs. The source code is distributed as part of the *Rmosek* package.

Notes on the Matrix package: A number of bugfixes were introduced in the Matrix package posterior to dependencies on R(>=2.12.0). These fixes have been backported in this package, to older versions of the package Matrix(>=0.9996875-3) depending only on R(>=2.10.0). Effectively, more users will be able to run Rmosek. The source code for this backport is located in the *compatibility* folder from which it is included in the *local_stubs.h* and *local_stubs.cc* files.

[24] Check out: mosek.com → Documentation → C API manual → API reference.
[25] Check out: r-project.org → Manuals

4.2 Limitations and Missing Features

The interface currently provides the features needed for basic research and educational purposes, but has many shortcomings when compared to the Matlab-to-MOSEK interface[26]. This is a short list of missing features.

1. **Direct input of Quadratic Programs**: At the moment the user will need to manually transform the Quadratic Problem to a Conic Program as the interface only provides for the direct input of linear and conic programs. The MOSEK optimization library actually supports the direct input of Quadratic programs, so this should only require a small amount of coding.

2. **Direct input of any problem that can be modeled as a Conic Program**: To completely avoid the manual transformations of non-linear constraints to conic subspaces, the interface should allow for the direct input of these problems. This is a hard task, but might soon become much easier when the package "MOSEK Fusion" is released as part of the optimization library. This package allows for a more dynamic input of problems and will make this task much easier to code.

3. **Custom R-coded callback functions**: While the MOSEK optimization library executes an optimization procedure, it is possible to define a callback function to check on the progress and terminate ahead of time. It could be useful to make this possibility available from R, such that a callback option pointing to an R-function could be specified.

[26] Check out: mosek.com → Optimization toolbox for MATLAB.

A Command reference

▶ `r <- mosek_version()`

Retrieves a string containing the version number of the utilized MOSEK optimization library. Briefly mentioned in Section 2.1.

▶ `r <- mosek(problem, opts = list())`

Solve an optimization problem using the MOSEK optimization library. The input variable ***problem*** could have any name, but should be a list object describing the optimization problem using the following fields.

`problem`	Problem description
`..$sense`	Objective sense, e.g. "max" or "min"
`..$c`	Objective coefficients
`..$c0`	Objective constant
`..$A`	Constraint matrix
`..$bc`	Lower and upper constraint bounds
`..$bx`	Lower and upper variable bounds
`..$cones`	Conic constraints
`..$intsub`	Integer variable indexes
`..$qobj`	Quadratic convex optimization
`..$scopt`	Separable convex optimization
`..$iparam/$dparam/$sparam`	Parameter list
`....$<MSK_PARAM>`	Value of any <MSK_PARAM>
`..$sol`	Initial solution list
`....$itr/$bas/$int`	Initial solution description

The argument ***sense*** is the goal of the optimization and should indicate whether we wish to maximize or minimize the objective function given by $f(x) = c^T x + c_0$, where $\boldsymbol{c}$ is the objective coefficients and ***c0*** the objective constant. The matrix $\boldsymbol{A}$ together with bounds ***bc*** describes the linear constraints of the problem, while variable bounds are given by ***bx***. These input arguments describe a linear program (see Section 2.2).

The argument ***cones*** is used for Conic Programming and has been described in Section 2.3. The issue regarding the transformation of other convex problems to the formulation of a Conic Program has been addressed in Appendix B, with more details on the transformation of Quadratic Programs in Section 2.3.3.

The argument ***intsub*** is used to specify whether some of the variables should only be allowed to take integer values as part of a feasible solution. This is necessary for the class of Mixed Integer Programs described in Section 2.4.

The argument ***qobj*** is used for quadratic convex optimization and allows quadratic terms in the objective function. This argument has been described in Section 3.2.

The argument ***scopt*** is used for separable convex optimization and allows non-linear unary operators in formulations involving exponential or logarithmic terms. This argument has been described in Section 3.3.

The arguments ***iparam***, ***dparam*** and ***sparam*** are used to specify lists of integer-, double- and string-typed parameters for the MOSEK optimization library in order to aid or take control of the optimization process. This has been described in Section 2.5.

The argument ***sol*** is used to specify an initial solution used to hot-start the optimization process, which is likely to increase the solution speed. This has been described for linear programming in Section 2.2.4 and for mixed integer programming in Section 2.4.2.

The options ***opts*** could have any name, and are, in fact, often input directly as an anonymous list. It has the following fields.

`opts`	Options for the 'mosek' function
`..$verbose`	Output logging verbosity
`..$usesol`	Whether to use the initial solution
`..$useparam`	Whether to use the specified parameter settings
`..$soldetail`	Level of detail used to describe solutions
`..$getinfo`	Whether to extract MOSEK information items
`..$writebefore`	Filepath used to export model
`..$writeafter`	Filepath used to export model and solution

The argument ***verbose*** is used to specify the amount of logging information given by the interface. This is described in Section 2.7.1.

The argument ***usesol*** can be used to ignore the problem field *sol*, while ***useparam*** can be used to ignore the fields *iparam*, *dparam* and *sparam*, without having to change the problem description. This usage has not been covered by the user guide, but it should be noted that both options are true by default in contrary to the context of Section 2.6.

The arguments ***soldetail*** and ***getinfo*** are used to control the amount of information returned about the solution, and the MOSEK optimization library internals, respectively. This is described in Section 2.7.4.

The arguments ***writebefore*** and ***writeafter*** are used to see the optimization model (and identified solution) constructed by MOSEK, written out to a file just before (or immediately after) the optimization process. This is described in Section 2.6.

The resulting function output variable ***r***, returned by the interface, is capable of holding

the three types of solutions listed below, along with the response of the function call. The existence of a solution depends on the optimization problem and the algorithm used to solve it.

`r`	Result of the 'mosek' function
`..$response`	Response from the MOSEK optimization library
`....$code`	ID-code of response
`....$msg`	Human-readable message
`..$sol`	Solution list
`....$itr/$bas/$int`	Solution description
`......$solsta`	Solution status
`......$prosta`	Problem status
`......$skx`	Variable bound status keys
`......$skc`	Linear constraint status keys
`......$skn`	Conic constraint status keys
`......$xx`	Variable activities
`......$xc`	Constraint activities
`......$slc`	Dual variable for constraint lower bounds
`......$suc`	Dual variable for constraint upper bounds
`......$slx`	Dual variable for variable lower bounds
`......$sux`	Dual variable for variable lower bounds
`......$snx`	Dual variable of conic constraints
`......$pobjval`	Primal objective value[27]
`......$dobjval`	Dual objective value[27]
`......$pobjbound`	Best primal objective bound from relaxations[27]
`......$maxinfeas`	Maximal solution infeasibilities[27]
`........$pbound`	Primal inequality constraints
`........$peq`	Primal equality constraints
`........$pcone`	Primal cone constraints
`........$dbound`	Dual inequality constraints
`........$deq`	Dual equality constraints
`........$dcone`	Dual cone constraints
`........$int`	Integer variables
`..$iinfo/$dinfo`	MOSEK information list[28]
`....$<MSK_INFO>`	Value of any <MSK_INFO>

The ***response*** can be used to determine whether a function call returned successfully. It contains a numeric response code, ***code***, and a string response message, ***msg***, both explained in "Errors, warnings and response codes" in Section 2.2.1. Please also see Section

[27] Only available if requested by option `soldetail`.
[28] Only available if requested by option `getinfo`.

2.2.5 on how to use this information.

The solution ***itr*** will exist whenever the interior-point algorithm has been executed. This algorithm is used by default for all continuous optimization problems, and has been described in Section 2.2 and 2.3.

The solution ***bas*** will exist whenever the interior-point algorithm or the simplex method has been executed to solve a linear optimization problem (see Section 2.2).

The solution ***int*** will exist whenever variables are required to take integer values and the corresponding Mixed Integer Program has been solved (see Section 2.4).

The ***iinfo*** and ***dinfo*** lists hold information on the MOSEK optimization library internals, and have been described in Section 2.7.4.

▶ `mosek_clean()`

Forces the early release of any previously acquired MOSEK license. If you do not share a limited number of licenses among multiple users, you do not need to use this function. The acquisition of a new MOSEK license will automatically take place at the next call to the function `mosek` given a valid problem description, using a small amount of extra time. This usage is briefly mentioned in Section 2.1.

For advanced users: If you utilize the .Call convention directly, bypassing the mosek R-function definition, an `Rf_error` will result in an unclean memory space. For this reason you can also use `mosek_clean` to tidy up uncleaned resources in case an error occurs. Otherwise this cleaning will not happen until the next call to `mosek` or until the library is unloaded. This usage have not been documented elsewhere.

▶ `r <- mosek_write(problem, filepath, opts = list())`

Outputs a model of an optimization problem in any standard modeling fileformat (e.g. lp, opf, mps, mbt, etc.), controlled by a set of options. The modeling fileformat is selected based on the extension of the modelfile. This function is used and explained in Section 2.6.

The input variable ***problem*** could have any name, but should be a list object describing the optimization problem using the same fields as for the `mosek` function on page 67.

The input variable ***filepath*** should be a string describing the path to modelfile. This path can either be absolute or relative to the working directory, and will overwrite any existing data on this location. The specified location will be the destination of the exported model.

The options ***opts*** could have any name, and are, in fact, often specified directly as an anonymous list. It has the following fields.

`opts`	Options for the 'mosek_write' function
`..$verbose`	Output logging verbosity
`..$usesol`	Whether to write an initial solution
`..$useparam`	Whether to write all parameter settings
`..$getinfo`	Whether to extract MOSEK information items
`..$scofile`	Destination of operators from scopt

The argument ***verbose*** is used to specify the amount of logging information given by the interface. This is described in Section 2.5.

The argument ***usesol*** can be used to ignore the ***problem*** field *sol*, while ***useparam*** can be used to ignore the fields *iparam*, *dparam* and *sparam*, when writing the optimization model. By default, the argument ***usesol*** is true and ***useparam*** is false, in contrary to the defaults of the `mosek` function. These two options are covered in Section 2.6.

The argument ***getinfo*** is used to indicate whether to extract information about the MOSEK optimization library internals. This is described in Section 2.7.4.

The argument ***scofile*** is used in separable convex optimization to specify the file to which the operators must be saved. This is described in Section 3.3.3.

The resulting function output variable ***r***, returned by the interface, holds the response of the function call.

`r`	Result of the 'mosek_write' function
`..$response`	Response from the MOSEK optimization library
`....$code`	ID-code of response
`....$msg`	Human-readable message
`..$iinfo/$dinfo`	MOSEK information list[29]
`....$<MSK_INFO>`	Value of any <MSK_INFO>

The ***response*** can be used to determined whether a function call returned successfully. It contains a numeric response code, ***code***, and a string response message, ***msg***, both explained in "Errors, warnings and response codes" in Section 2.2.1. Please also see Section 2.6 for an example on how to use this information.

The ***iinfo*** and ***dinfo*** lists hold information on the MOSEK optimization library internals, and have been described in Section 2.7.4.

[29] Only available if requested by option `getinfo`.

▶ `r <- mosek_read(filepath, opts = list())`

Interprets a model from any standard modeling fileformat (e.g. lp, opf, mps, mbt, etc.), controlled by a set of options. The result contains an optimization problem which is compliant with the input specifications of function `mosek`. This function is used and explained in Section 2.6.

The input variable ***filepath*** should be a string describing the path to file. This path can either be absolute or relative to the working directory. The specified location will be the source of the optimization model to be read.

The options ***opts*** could have any name, and are, in fact, often input directly as an anonymous list. It has the following fields.

`opts`	Options for the 'mosek_read' function
`..$verbose`	Output logging verbosity
`..$usesol`	Whether to write an initial solution
`..$useparam`	Whether to write all parameter settings
`..$getinfo`	Whether to extract MOSEK information items
`..$scofile`	Source of operators read to scopt
`..$matrixformat`	The sparse format of the constraint matrix

The argument ***verbose*** is used to specify the amount of logging information given by the interface. This is described in Section 2.5.

The argument ***usesol*** can be used to ignore the ***problem*** field *sol*, while ***useparam*** can be used to ignore the fields *iparam*, *dparam* and *sparam*, when reading the optimization model. By default, the argument ***usesol*** is true and ***useparam*** is false, in contrary to the defaults of the `mosek` function. These two options are covered in Section 2.6.

The argument ***getinfo*** is used to indicate whether to extract information about the MOSEK optimization library internals. This is described in Section 2.7.4.

The argument ***scofile*** is used in separable convex optimization to specify the file from which the operators should be read. This is described in Section 3.3.3.

The argument ***matrixformat*** determines the matrix format of the ***problem*** field A. Currently a sparse coordinate (COO) and a compressed sparse column (CSC) format is supported from package 'Matrix', along with a simple list-based coordinate format. This is described in Section 2.6.

The resulting function output variable ***r***, returned by the interface, holds the response of the function call.

`r`	Result of the 'mosek_read' function
`..$response`	Response from the MOSEK optimization library
`....$code`	ID-code of response
`....$msg`	Human-readable message
`..$prob`	Problem description
`..$iinfo/$dinfo`	MOSEK information list[30]
`....$<MSK_INFO>`	Value of any <MSK_INFO>

The ***response*** can be used to determined whether a function call returned successfully. It contains a numeric response code, ***code***, and a string response message, ***msg***, both explained in "Errors, warnings and response codes" in Section 2.2.1. Please see Section 2.6 for an example on how to use this information.

The output variable ***prob*** is a list object describing the imported optimization problem. It has the same fields as the input variable 'problem' for the `mosek` function on page 67.

The ***iinfo*** and ***dinfo*** lists hold information on the MOSEK optimization library internals, and have been described in Section 2.7.4.

[30] Only available if requested by option `getinfo`.

B Conic transformations

This appendix will introduce the reader to some of the transformations that make it possible to formulate a large class of non-linear problems as Conic Quadratic Programs[31]. Transformations do not have to be hard, and some non-linear constraints easily take the shape of a quadratic cone (see Section 2.3.1 for definitions). Linear constraints can be added to a Conic Quadratic Program directly without transformations.

The $\sqrt{x}$ Example Given constraint $\sqrt{x} \geq t$ with $x, t \geq 0$, we rewrite it to a rotated quadratic cone as follows:

$$\begin{array}{rcl} \sqrt{x} & \geq & t \\ x & \geq & t^2 \\ 2xr & \geq & t^2 \end{array} \tag{B.1}$$

with linear relationship

$$r = 1/2\,. \tag{B.2}$$

□

The definition of linear relationships hardly have any effect on speed of the optimization process, and many techniques are implemented in MOSEK to effectively reduce the size of the problem. Thus many CQP formulations solve faster than what could be expected from the larger model that results from almost any transformation.

The $x^{3/2}$ Example Given constraint $x^{3/2} \leq t$ with $x, t \geq 0$, we rewrite it to a pair of rotated quadratic cones and three linear relationship as follows. Note that from line five to six we use the results of the $\sqrt{x}$ example above.

$$\begin{array}{rcl} x^{3/2} & \leq & t \\ x^{2-1/2} & \leq & t \\ x^2 & \leq & \sqrt{x}t \\ x^2 & \leq & 2st, \quad 2s \leq \sqrt{x} \\ x^2 & \leq & 2st, \quad w \leq \sqrt{v}, \quad w = 2s, \quad v = x \\ x^2 & \leq & 2st, \quad w^2 \leq 2vr, \quad w = 2s, \quad v = x, \quad r = 1/2 \\ x^2 & \leq & 2st, \quad w^2 \leq 2vr, \quad w = s, \quad v = x, \quad r = 1/8\,. \end{array} \tag{B.3}$$

□

[31] More examples and transformation rules can be found online:
Check out: mosek.com → Documentation → Optimization tools manual → Modeling → Conic optimization.

Monomials in General The crucial step in transforming the individual terms (monomials) of a polynomial function to the form of a conic program, is the following recursion holding true for positive integers n and non-negative y_j variables. Here, each line implies the next, and the inequality ends up having the same form as to begin with, only with n reduced by one. Repeating this procedure until $n = 1$, the inequality will finally take the form of a rotated quadratic cone. From the first expression, we move the coefficients around to get line two, substitute variables by the cones in (B.5) to get line three, and take the square root on both sides to reach line four.

$$\begin{array}{lcl} s^{2^n} & \leq & 2^{n2^{n-1}} \left[y_1 \; y_2 \; \cdots \; y_{2^n} \right] \\ s^{2^n} & \leq & 2^{(n-1)2^{n-1}} \left[(2y_1y_2) \; (2y_3y_4) \; \cdots \; (2y_{(2^n-1)}y_{2^n}) \right] \\ s^{2^n} & \leq & 2^{(n-1)2^{n-1}} \left[x_1^2 \; x_2^2 \; \cdots \; x_{2^{n-1}}^2 \right] \\ s^{2^{(n-1)}} & \leq & 2^{(n-1)2^{n-2}} \left[x_1 \; x_2 \; \cdots \; x_{2^{n-1}} \right] \end{array} \tag{B.4}$$

Also note that the definition of new variables created by this recursion all takes the form of a rotated quadratic cone as shown below, with all $y_j \geq 0$.

$$x_j^2 \; \leq \; 2 \, y_{(2j-1)} \, y_{2j} \quad \forall j \tag{B.5}$$

Type I Having the inequality $x^{p/q} \leq t$ for some rational exponent $p/q \geq 1$, we are now able to transform it into a set of cones by substituting $p = 2^n - w$ for positive integers n and w. This will yield a form close to the top of (B.4) as shown below, where the variables x and t just needs to be coupled with s and y for the recursion to be usable. An example of this follows in (B.9).

$$\begin{array}{rcl} x^{p/q} & \leq & t \\ x^{(2^n-w)/q} & \leq & t \\ x^{2^n} & \leq & x^w t^q \end{array} \tag{B.6}$$

Type II The inequality $x^{p/q} \geq t$ for some rational exponent $0 \leq p/q \leq 1$ and $x \geq 0$, can be transformed into the form of a type I monomial quite easily.

$$\begin{array}{rcl} x^{p/q} & \geq & t \\ x & \geq & t^{q/p} \end{array} \tag{B.7}$$

Type III Having instead the inequality $x^{-p/q} \leq t$ for any integers $p \geq 1$, $q \geq 1$, we can use the transform shown below. This again will yield a form close to the top of (B.4), where s is a constant and the variables just need to be coupled with y. You need to choose n such that $2^n \geq p + q$ in order to make this transformation work. See example (B.10).

$$\begin{array}{rcl} x^{-p/q} & \leq & t \\ 1 & \leq & x^p t^q \end{array} \tag{B.8}$$

Monomials - Example 1 Given $x^{5/3} \leq t$, we rewrite it as shown below.

$$\begin{array}{rcl} x^{5/3} & \leq & t \\ x^5 & \leq & t^3 \\ x^8 & \leq & x^3 t^3 \\ s^8 & \leq & 2^{12} y_1 y_2 \cdots y_8 \end{array} \tag{B.9}$$
$$s = x, \quad y_1 = y_2 = y_3 = x, \quad y_4 = y_5 = y_6 = t, \quad y_7 = 2^{-12}, \quad y_8 = 1$$

Monomials - Example 2 Given $x^{-5/2} \leq t$, we rewrite it as shown below.

$$\begin{array}{rcl} x^{-5/2} & \leq & t \\ x^{-5} & \leq & t^2 \\ 1 & \leq & x^5 t^2 \\ s^8 & \leq & 2^{12} y_1 y_2 \cdots y_8 \end{array} \tag{B.10}$$
$$s = (2^{12})^{1/8}, \quad y_1 = y_2 = \ldots = y_5 = x, \quad y_6 = y_7 = t, \quad y_8 = 1$$

Polynomials Now that we have seen how to transform the individual terms (monomials) of a polynomial function to the form of a conic program, it is easy to transform entire polynomial constraints. We assume $a_j \geq 0$ for monomials of type I and III, and $a_j \leq 0$ for monomials of type II.

$$\sum_{j=1}^{n} a_j x_j^{p_j/q_j} \quad \leq \quad b \tag{B.11}$$

Substituting each monomial by a new variable u_j, we are able to isolate each monomial by itself and use the previously defined transformation rules.

$$\begin{array}{rcll} \sum_{j=1}^{n} a_j u_j & \leq & b & \\ x_j^{p_j/q_j} & \leq & u_j & \forall j \text{ of type I and III} \\ x_j^{p_j/q_j} & \geq & u_j & \forall j \text{ of type II} \end{array} \tag{B.12}$$

Regarding the objective function, minimization problems can be rewritten as shown below matching (B.11) and, therefore, it is applicable to the isolation of monomials and the whole transformation used above. In this way the objective function can also take the linear form required by conic programs.

$$\text{minimize} \;\; f(x) \quad \Longleftrightarrow \quad \begin{array}{l} \text{minimize} \;\; b \\ f(x) \leq b \end{array} \tag{B.13}$$

Polynomials - Example 1 This is a problem with type III monomials

$$\begin{array}{lc} \text{minimize} & c^T x \\ \text{subject to} & \sum_{j=1}^n \frac{f_j}{x_j} \leq b \\ & x \geq 0 \end{array} \tag{B.14}$$

where it is assumed that $f_j > 0$ and $b > 0$. It is equivalent to

$$\begin{array}{lcll} \text{minimize} & c^T x & & \\ \text{subject to} & \sum_{j=1}^n f_j z_j & = & b \\ & v_j & = & \sqrt{2} \quad j = 1, \ldots, n \\ & v_j^2 & \leq & 2 z_j x_j \quad j = 1, \ldots, n \\ & x, z \geq 0 & & \end{array} \tag{B.15}$$

Polynomials - Example 2 The objective function with a mixture of type I and type III monomials

$$\text{minimize} \quad x^2 + x^{-2} \tag{B.16}$$

is used in statistical matching and can be formulated as

$$\begin{array}{lcl} \text{minimize} & u + v & \\ \text{subject to} & x^2 & \leq u \\ & x^{-2} & \leq v \end{array} \tag{B.17}$$

which is equivalent to the quadratic conic optimization problem

$$\begin{array}{lcll} \text{minimize} & u + v & & \\ \text{subject to} & x^2 & \leq & 2uw \\ & s^2 & \leq & 2y_{21}y_{22} \\ & y_{21}^2 & \leq & 2y_1 y_2 \\ & y_{22}^2 & \leq & 2y_3 y_4 \\ & w & = & 1 \\ & s & = & 2^{3/4} \\ & y_1 & = & y_2 = x \\ & y_3 & = & v \\ & y_4 & = & 1 \end{array} \tag{B.18}$$

B.1 The Quadratic Program

Any convex quadratic problem can be stated on the form

$$\begin{array}{lll} \text{minimize} & 0.5\|Fx\|^2 + c^T x & \\ \text{subject to} & 0.5\|Gx\|^2 + a^T x & \leq \quad b \end{array} \tag{B.19}$$

where F and G are matrices and c and a are column vectors. Here a convex quadratic term $x^T Q x$ would be written $\|Fx\|^2$, where $F^T F = Q$ is the Cholesky factorization. For simplicity's sake we assume that there is only one constraint, but it should be obvious how to generalize the methods to an arbitrary number of constraints. Problem (B.19) can be reformulated as

$$\begin{array}{lrcl} \text{minimize} & 0.5\|t\|^2 + c^T x & & \\ \text{subject to} & 0.5\|z\|^2 + a^T x & \leq & b \\ & Fx - t & = & 0 \\ & Gx - z & = & 0 \end{array} \tag{B.20}$$

after the introduction of the new variables t and z. It is easy to convert this problem to a conic quadratic optimization problem, i.e.

$$\begin{array}{lrcll} \text{minimize} & v + c^T x & & & \\ \text{subject to} & p + a^T x & = & b & \\ & Fx - t & = & 0 & \\ & Gx - z & = & 0 & \\ & w & = & 1 & \\ & q & = & 1 & \\ & \|t\|^2 & \leq & 2vw & v, w \geq 0 \\ & \|z\|^2 & \leq & 2pq & p, q \geq 0 \end{array} \tag{B.21}$$

In this case the last two inequalities both take the form of a rotated quadratic cone, and the entire problem can be solved as a conic quadratic program, Section 2.3, using the interior-point algorithm.

If F is a non-singular matrix - e.g. a diagonal matrix - then x can be eliminated from the problem using the substitution $x = F^{-1}t$. In most cases the MOSEK optimization library will perform this reduction automatically during the presolve phase before the actual optimization is performed.