Users Guide to the

Octave-to-MOSEK Interface

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

December 13, 2011

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 Octave 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 Setting up the interface

1.1 On Windows platforms

The interface documented here is part of the *OctMOSEK* package available on Google Code Project Hosting 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 and all the tools you need comes with the basic installation of ***Octave***. Besides this, you will need to download two items freely available on the Internet. In summary:

- **MOSEK** (the optimization library we interface to)[2]
- **OctMOSEK** (the source code of this package)

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 "Using the packages" in the FAQ available on the Octave-Forge website may be useful. The chapter "Packages" in the online Octave Manual[3] is another good source of information.

Setting up the target machine

We assume here that you have a working installation of Octave and MOSEK on the machine targeted for the Octave-to-MOSEK interface. The architectures (32 or 64 bit) of these two programs must be exactly the same for consistency. Thus, since no stable 64 bit release of Octave exists as of now, the 64 bit version of the MOSEK optimization library is not supported.

No support for 64bit MOSEK in Octave!

The first step is to set up the Windows environment variable called ***PATH***, in order to utilize the package building tools shipped with the basic installation of Octave. This will extend the Windows CMD wih Unix-style commands, and make it possible to compile the source code of the Octave packages.

[1] Such a binary would have to be built individually for each version of Octave and MOSEK.
[2] Check *mosek.com* on trial and free academic licenses for MOSEK if needed.
[3] Check out: www.gnu.org/software/octave → Docs → Manual.

Assuming that the home directory of the Octave installation was `c:\Octave\BUILDNAME`, for some octave-build (BUILDNAME), 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:\Octave\BUILDNAME\bin;` to enable the Octave toolset.
Add `c:\Octave\BUILDNAME\mingw32\bin;` to enable 32 bit installations.
Add `c:\Octave\BUILDNAME\MSYS\bin;` to enable build scripts.

Please ensure that the PATH variable also contains the "bin" folder of the 32 bit MOSEK installation that you wish this interface to target. This is necessary for automatic configuration to work, and could look something like:

```
C:\Progra~2\Mosek\6\tools\platform\win32x86\bin
```

Finally, if you have not already done so, please download the source code for the OctMOSEK package[4]. The name of the file containing the source code will be on the form `octmosek_VERSION.tar.gz`, for some version (VERSION), and should just be saved to a location that you can find again.

Installing the package with automatic configuration

Automatic configuration works equivalently to calling `where mosek` in the Windows CMD. That is, it searches the environment variable called PATH, for folders that contain an executable called 'mosek'. From the first such folder encountered, it then determines the most basic of the available optimization libraries within (typically mosek.lib), along with other relevant information. This configuration should work for all users requiring only the basic optimization library. Otherwise, manual configuration of the package will be needed.

Open Octave and execute a command similar to the one shown below:

```
pkg install -verbose 'LOCALTION\octmosek_VERSION.tar.gz'
```

Remember to exchange *LOCATION* with the path to the downloaded package, and *VERSION* with the version of it. This will make a static link to the automatically located MOSEK optimization library, and install the 'OctMOSEK' package. Please note that the command above uses single quotes for an uninterpreted string. Strings delimitted by double quotes have a backslash interpretation that may conflict with Windows paths.

[4] Newest release of OctMOSEK available at: code.google.com/p/octmosek → Downloads

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. Manual configuration works by defining both `PKG_MOSEKHOME` and `PKG_MOSEKLIB` as global variables. If only one of these global variables are defined in the Octave environment, the package will not attempt to guess the other. To perform a manual configuration, execute commands similar to the ones shown below, with correct values for `PKG_MOSEKHOME` and `PKG_MOSEKLIB` as discussed next.

```
global PKG_MOSEKHOME; global PKG_MOSEKLIB;
PKG_MOSEKHOME = '...';
PKG_MOSEKLIB = '...';
pkg install -verbose 'LOCATION\octmosek_VERSION.tar.gz'
```

Remember to exchange the ... of both PKG_MOSEKHOME and PKG_MOSEKLIB with the correct values, as explained next. As with automatic configuration, you should also substitute *LOCATION* with the path to the downloaded package, and *VERSION* with the version of it. Moreover, refrain from using the error-prone construct `global VAR = VALUE`, as the *VALUE* will then only be used in case of initialization, when the variable is not already defined. Please note that the command and definitions above uses single quotes for an uninterpreted string. Strings delimitted by double quotes have a backslash interpretation that may conflict with Windows paths.

The definition of the 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 Octave program. This could for instance look something like:

```
C:/Progra~2/Mosek/6/tools/platform/win32x86
```

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 OctMOSEK 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 its file-extension. The PKG_MOSEKLIB would thus normally be `mosek` (linking to mosek.lib).

Notice that if you wish to uninstall the 'OctMOSEK' package at some point later, this can be done as for any other package with the command `pkg uninstall octmosek`.

1.2 On Unix-alike platforms

The interface documented here is part of the *OctMOSEK* package available on Google Code Project Hosting 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 and all the tools you need comes with the . Besides the basic installation of ***Octave***, you will need to download two things freely available on the Internet. In summary:

- **MOSEK** (the optimization library we interface to)[6]
- **OctMOSEK** (the source code of this package)

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

For additional help installing this interface the section on "Using the packages" in the FAQ available on the Octave-Forge website may be useful. The chapter "Packages" in the online Octave Manual[7] is another good source of information.

[5] Such a binary would have to be built individually for each version of Octave and MOSEK.
[6] Check *mosek.com* on trial and free academic licenses for MOSEK if needed.
[7] Check out: www.gnu.org/software/octave → Docs → Manual.

Setting up the target machine

We assume here that you have a working installation of Octave and MOSEK on the machine targeted for the Octave-to-MOSEK interface. The architectures (32 or 64 bit) of these two programs must be exactly the same for consistency.

For auto-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 like:

```
~/mosek/6/tools/platform/linux32x86/bin
```

If more than one "bin" folder from a MOSEK installation is specified, only the first one will be found by automatic configuration.

Finally, if you have not already done so, please download the source code for the OctMOSEK package[8]. The name of the file containing the source code will be on the form `octmosek_VERSION.tar.gz`, for some version (VERSION), and should just be saved to a location that you can find again.

▶ ***A note to users of Machintosh***
The `mkoctfile` command in Octave defaults to building 64 bit versions on newer versions of Macintosh (from Snow Leopard). To setup the target machine for a 32 bit build of OctMOSEK, the user will have to set a compiler flag manually. Navigate to the Applications folder, show the contents of Octave.app, and locate the following file where *VERSION* is the version of the program.

```
Octave.app/Contents/Resources/bin/mkoctfile-VERSION
```

Open this file with a text editor (e.g. TextEdit.app), and add the following lines immidiately after the "set -e" line.

```
CFLAGS="-m32 ${CFLAGS}"
FFLAGS="-m32 ${FFLAGS}"
CPPFLAGS="-m32 ${CPPFLAGS}"
CXXFLAGS="-m32 ${CXXFLAGS}"
LDFLAGS="-m32 ${LDFLAGS}"
```

This will force 32 bit builds of all packages.

[8] Newest release of OctMOSEK available at: code.google.com/p/octmosek → Downloads

Installing the package with automatic configuration

Automatic configuration works equivalently to calling `which mosek` in a terminal window. That is, it searches the environment variable called PATH, for folders that contain an executable called 'mosek'. From the first such folder encountered, it then determines the most basic of the available optimization libraries within (typically `mosek` with the extension `.so` or `.dylib`), along with other relevant information. This configuration should work for all users requiring only the basic optimization library. Otherwise, manual configuration of the package will be needed.

Open Octave in the architecture (32 or 64 bit) that you wish to install the package on, and execute a command similar to the one shown below:

```
pkg install -verbose 'LOCATION/octmosek_VERSION.tar.gz'
```

Remember to exchange *LOCATION* with the path to the downloaded package, and *VERSION* with the version of it. This will make a static link to the automatically located MOSEK optimization library, and install the 'OctMOSEK' package. Please note that the command above uses single quotes for an uninterpreted string. Strings delimitted by double quotes have a backslash interpretation.

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. Manual configuration works by defining both `PKG_MOSEKHOME` and `PKG_MOSEKLIB` as global variables. If only one of these global variables are defined in the Octave environment, the package will not attempt to guess the other. To perform a manual configuration, execute commands similar to the ones shown below, with correct values for `PKG_MOSEKHOME` and `PKG_MOSEKLIB` as discussed next.

```
global PKG_MOSEKHOME; global PKG_MOSEKLIB;
PKG_MOSEKHOME = '...';
PKG_MOSEKLIB = '...';
pkg install -verbose 'LOCATION/octmosek_VERSION.tar.gz'
```

Remember to exchange the ... of both PKG_MOSEKHOME and PKG_MOSEKLIB with the correct values, as explained next. As with automatic configuration, you should also substitute *LOCATION* with the path to the downloaded package, and *VERSION* with the version of it. Moreover, refrain from using the error-prone construct `global VAR = VALUE`, as the *VALUE* will then only be used in case of initialization, when the variable is not already defined. Please note that the command and definitions above uses single

quotes for an uninterpreted string. Strings delimitted by double quotes have a backslash interpretation.

The definition of the 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 Octave program. This could for instance look like:

```
~/mosek/6/tools/platform/linux32x86
```

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 OctMOSEK 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 Octave program and MOSEK installation, while `mosek` implies 32 bit.

2 A guided tour

2.1 Help: How do I use the interface?

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

```
pkg load octmosek
```

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.

If the package loaded successfully without errors, you should now be able to use the functions of the Octave-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 the complete list, with short descriptions, of all functions in the *OctMOSEK* package, the following command can be used:

```
pkg describe -verbose octmosek
```

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 Octave 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 read from, and write to, a specific model file.

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

```
help 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. The input arguments and return value printed as part of the information given when calling `help mosek`, is shown in Figure 1.

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, we recommend that you simply read on. If you on the other hand is looking for something specifically, it may be beneficial to look at the brief summary, and references to more information, that have been made available in Appendix A.

Figure 1: Help information (input arguments and return value)

```
-- Loadable Function: R = mosek (PROBLEM, OPTS = struct())
    >> Solve an optimization problem

    ========== Arguments ==========

    problem                     STRUCTURE
    ..sense                     STRING
    ..c                         REAL VECTOR
    ..c0                        SCALAR                (OPTIONAL)
    ..A                         SPARSE MATRIX
    ..blc                       REAL VECTOR
    ..buc                       REAL VECTOR
    ..blx                       REAL VECTOR
    ..bux                       REAL VECTOR
    ..cones                     CELL                  (OPTIONAL)
    ....{i}.type                STRING
    ....{i}.sub                 INTEGER VECTOR
    ..intsub                    INTEGER VECTOR        (OPTIONAL)
    ..iparam/dparam/sparam      STRUCTURE             (OPTIONAL)
    ....<MSK_PARAM>             STRING / SCALAR       (OPTIONAL)
    ..sol                       STRUCTURE             (OPTIONAL)
    ....itr/bas/int             STRUCTURE             (OPTIONAL)

    opts                        STRUCTURE             (OPTIONAL)
    ..verbose                   SCALAR                (OPTIONAL)
    ..usesol                    BOOLEAN               (OPTIONAL)
    ..useparam                  BOOLEAN               (OPTIONAL)
    ..writebefore               STRING (filepath)     (OPTIONAL)
    ..writeafter                STRING (filepath)     (OPTIONAL)

    ========== Value ==========

    r                           STRUCTURE
    ..response                  STRUCTURE
    ....code                    SCALAR
    ....msg                     STRING
    ..sol                       STRUCTURE
    ....itr/bas/int             STRUCTURE           (SOLVER DEPENDENT)
    ......solsta                STRING
    ......prosta                STRING
    ......skx                   STRING VECTOR
    ......skc                   STRING VECTOR
    ......xx                    REAL VECTOR
    ......xc                    REAL VECTOR
    ......slc                   REAL VECTOR         (NOT IN int)
    ......suc                   REAL VECTOR         (NOT IN int)
    ......slx                   REAL VECTOR         (NOT IN int)
    ......sux                   REAL VECTOR         (NOT IN int)
    ......snx                   REAL VECTOR         (NOT IN int/bas)
```

2.2 Linear Programming

2.2.1 Solving LP problems

Linear optimization problems[9] 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 Octave 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 Octave as shown in Figure 2. The first line clears 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.

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

Figure 2: Linear Optimization (lo1)

```
clear -v lo1;
lo1.sense = "max";
lo1.c = [3 1 5 1];
lo1.A = sparse([3 1 2 0;
                2 1 3 1;
                0 2 0 3]);
lo1.blc = [30 15 -Inf];
lo1.buc = [30 Inf 25];
lo1.blx = [0 0 0 0];
lo1.bux = [Inf 10 Inf Inf];
r = mosek(lo1);
```

Notice how the Octave 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 1, Section 2.1).

Objective The string ***sense*** is the objective goal and could be either "minimize", "min", "maximize" or "max". The dense real 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. `sparse(nrows, ncols)`.

Bounds The constraint bounds ***blc*** (constraint lower bound) and ***buc*** (constraint upper bound), as well as the variable bounds ***blx*** (variable lower bound) and ***bux*** (variable upper bound), are all given as dense real vectors. 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 (`""`, `[]`, `{}`, `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 1). 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[10] 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. Beware that if you wish to check for NaN, you should use the `'isnan'` function as the comparison operators such as `'=='` and `'isequal'` will not work.

`NB!` The interface may return only partially constructed output.
(always check the response code; e.g. *success* `= isequal(`*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.

[10] 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 3 and Figure 4 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 3: Primal Solution I (lo1)

```
r.sol.itr = {
 solsta = OPTIMAL
 prosta = PRIMAL_AND_DUAL_FEASIBLE

 skc = {                    skx = {
   [1,1] = EQ                 [1,1] = LL
   [1,2] = SB                 [1,2] = LL
   [1,3] = UL                 [1,3] = SB
 }                            [1,4] = SB
                            }
 xc =
   30.000    53.333    25.000
 xx =
   1.0441e-08    2.8561e-08    1.5000e+01    8.3333e+00
 ...
}
```

Figure 4: Primal Solution II (lo1)

```
r.sol.bas = {
 solsta = OPTIMAL
 prosta = PRIMAL_AND_DUAL_FEASIBLE

 skc = {                    skx = {
   [1,1] = EQ                 [1,1] = LL
   [1,2] = BS                 [1,2] = LL
   [1,3] = UL                 [1,3] = BS
 }                            [1,4] = BS
                            }
 xc =
   30.000    53.333    25.000
 xx =
    0.00000     0.00000    15.00000     8.33333
 ...
}
```

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[11], and should always be verified before the returned solution values are used (see Section 2.2.5).

The solution also contains status keys for both variables and constraints (called ***skx*** and ***skc***). Each status key can take the value of any of the following 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.

[11] 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

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

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 5 and 6. 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* on the other hand is a by-product of the interior-point algorithm that 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 5: Dual Solution I (lo1)

```
r.sol.itr = {
 solsta = OPTIMAL
 prosta = PRIMAL_AND_DUAL_FEASIBLE

 ...

 slc =
  -0.0000e+00   1.5575e-09  -0.0000e+00
 suc =
  -2.50000  -0.00000  -0.33333
 slx =
  -4.5000e+00  -2.1667e+00  -4.9830e-09  -1.0270e-08
 sux =
  -0.0000e+00   5.5856e-10  -0.0000e+00  -0.0000e+00
 snx =
   0   0   0   0
}
```

Figure 6: Dual Solution II (lo1)

```
r.sol.bas = {
 solsta = OPTIMAL
 prosta = PRIMAL_AND_DUAL_FEASIBLE

 ...

 slc =
  -0  -0  -0
 suc =
  -2.50000  -0.00000  -0.33333
 slx =
  -4.50000  -2.16667  -0.00000  -0.00000
 sux =
  -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[12] *OPTIMIZER* controls which optimizer to use within the MOSEK optimization library to solve the specified problem. The default value of this parameter is the enum reference-string "OPTIMIZER_FREE" which imply that MOSEK should choose

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

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[13]:

`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 7, adding a single line to the 'lo1' definition from earlier.

Figure 7: Selecting the primal simplex method

```
clear -v lo1;
lo1.sense = "max";
lo1.c = [3 1 5 1];
lo1.A = sparse([3 1 2 0;
                2 1 3 1;
                0 2 0 3]);
lo1.blc = [30 15 -Inf];
lo1.buc = [30 Inf 25];
lo1.blx = [0 0 0 0];
lo1.bux = [Inf 10 Inf Inf];
lo1.iparam.OPTIMIZER = "OPTIMIZER_PRIMAL_SIMPLEX";
r = mosek(lo1);
```

[13] 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 8.

Figure 8: 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 2, which is still primal feasible, we can expand the problem description and include x_5 as shown.

Figure 9: 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(end+1)   = -1;
lo1.A(:,end+1) = sparse([-2,-10,0])';
lo1.blx(end+1) = 0;
lo1.bux(end+1) = Inf;

% Extend and reuse the old basis solution
bas = r.sol.bas;
bas.skx{end+1} = 'LL';
bas.xx (end+1) = 0;
bas.slx(end+1) = 0;
bas.sux(end+1) = 0;

% Hot-start the simplex optimizer
lo1.iparam.OPTIMIZER = "OPTIMIZER_PRIMAL_SIMPLEX";
lo1.sol.bas = bas;
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 2), we fix the variable $x_4 = 2$, and hot-start using

Figure 10: 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.blx(4) = 2;
lo1.bux(4) = 2;

% Reuse the old basis solution
bas = r.sol.bas;

% Hotstart the simplex optimizer
lo1.iparam.OPTIMIZER = "OPTIMIZER_PRIMAL_SIMPLEX";
lo1.sol.bas = bas;
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 2), we add the constraint $x_1 + x_2 \geq 2$, and hot-start using

Figure 11: 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.A(end+1,:) = sparse([1,1,0,0]);
lo1.blc(end+1) = 2;
lo1.buc(end+1) = Inf;

% Extend and reuse the old basis solution
bas = r.sol.bas;
bas.skc{end+1} = 'LL';
bas.xc (end+1) = 2;
bas.slc(end+1) = 0;
bas.suc(end+1) = 0;

% Hot-start the simplex optimizer
lo1.iparam.OPTIMIZER = "OPTIMIZER_PRIMAL_SIMPLEX";
lo1.sol.bas = bas;
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 Octave-to-MOSEK interface. The explanation of status keys can be found on page 18.

Figure 12: Hot-starting using an initial guess

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

% Define the status key cell array
stkeys = {'BS','SB','LL','UL','EQ','UN'};

% Try to guess the optimal status keys
clear -v bas;
bas.skc = stkeys([5,1,4]);
bas.skx = stkeys([3,3,1,1]);

% Hot-start the simplex optimizer
lo1.iparam.OPTIMIZER = "OPTIMIZER_PRIMAL_SIMPLEX";
lo1.sol.bas = bas;
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 `find(strcmp(str,stkeys))` to find the index of a status key *str*, and `cellfun(@(str) find(strcmp(str,stkeys)), strcell)` for a cell of status keys *strcell*.

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 OctMOSEK package, execution errors similar to the ones generated by the built-in `error` function, part of the Octave language definition, may be provoked. This typically happens when the number of input arguments does not match a valid calling convention. Interface errors within the Octave API 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 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` and `catch` statements terminated by `end_try_catch`. Normally only the `try` block will be executed, but in case of an execution error the control is sent straight to the `catch` block. From here you can inspect the error by parsing the string `lasterr` and either rethrow the execution error with `rethrow(lasterror)`, throw your own error with `error("MSG")`, or do nothing. If you do nothing, execution will continue on the other side of the `end_try_catch` statement.

▶ 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 OctMOSEK interface). The list of response codes that can be returned by the MOSEK optimization library, are similar to those explained online[14].

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, a good solution may have been identified anyway. However, if the extraction of the solution was what caused the error in the first place, the returned solution may only be partially constructed. Ultimately,

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

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 by evaluating the variable within a `try` and `catch` statement. In the following example, the boolean variable `isdef` is determined by whether a solution status to a interior-point solution in the result has been defined.

```
try; res.sol.itr.solsta; isdef=true; catch; isdef=false; end_try_catch;
```

▶ 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 `printf`-commands, and will not fail to return a usable result unless explicitly stopped by an `error`-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 13.

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 13: Complete Best Practice 'lo1' example

```
function xx = get_lo1_solution_variables(maxtime)

  lo1 = struct("sense", "max", "c", [3 1 5 1]);
  lo1.A = sparse([3 1 2 0; 2 1 3 1; 0 2 0 3]);
  lo1.blc = [30 15 -Inf];     lo1.buc = [30 Inf 25];
  lo1.blx = [0 0 0 0];        lo1.bux = [Inf 10 Inf Inf];
  lo1.dparam.OPTIMIZER_MAX_TIME = maxtime;

  try
    rr = mosek(lo1, struct("verbose", 0));
  catch
    error("OctMOSEK failed somehow!")
  end_try_catch

  if (rr.response.code ~= 0)
    printf(["** ", "Response code: ", num2str(rr.response.code), "\n"]);
    printf(["** ", rr.response.msg, "\n"]);
    printf("Trying to continue..\n");
  endif

  try
    rbas = rr.sol.bas;
    rbas.solsta;  rbas.prosta;  rbas.xx;
  catch
    error("Basic solution was incomplete!");
  end_try_catch

  switch (rbas.solsta)
    case "OPTIMAL"
      printf("The solution was optimal, I am happy!\n");
    case "NEAR_OPTIMAL"
      printf("The solution was close to optimal, very good..\n");
    otherwise
      printf(["** ", "Solution status: ", rbas.solsta, "\n"]);
      printf(["** ", "Problem status: ", rbas.prosta, "\n"]);
      error("Solution could not be accepted!");
  endswitch

  xx = rbas.xx;

endfunction
```

□

2.3 Conic Quadratic Programming

2.3.1 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 (2.7), where matrices F, G, and columns a, b and c are assumed to have compliant dimensions. Notice that this problem is convex, and thereby easily solvable, only if F and G are both positive semi-definite.

$$\begin{array}{lll} \text{minimize} & 0.5\,x^T F x + c^T x & \\ \text{subject to} & 0.5\,x^T G x + a^T x & \leq \quad b. \end{array} \tag{2.7}$$

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 (2.7), the result is two of the simplest possible rotated quadratic cones - one for the $x^T F x$ term and one for the $x^T G x$ 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 F and G to be positive semi-definite, which is exactly the convexity requirement for QCQP's. This means that $x^T F x$ can be transformed only if $x^T F x$ 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.

2.3.2 Solving CQP problems

Conic quadratic programming[15] (also known as Second-order cone programming) is a generalization of linear, quadratic and all-quadratic programming. Conic quadratic programs pose the properties of strong duality and can be written as shown below. This is the primal problem.

$$\begin{array}{lccccc} \text{minimize} & & & c^T x + c_0 & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c, \\ & l^x & \leq & x & \leq & u^x, \\ & & & x \in \mathcal{C}. & & \end{array} \tag{2.8}$$

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 (2.8).

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

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

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

$$\mathcal{C}_t = \left\{ x \in \mathbb{R}^{n_t} \ : \ 2x_1 x_2 \geq \sum_{j=3}^{n_t} x_j^2, \ x_1 \geq 0, \ x_2 \geq 0 \right\}. \tag{2.10}$$

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

[15] 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:

$$\begin{array}{lrcl} \text{minimize} & x_5 + x_6 & & \\ \text{subject to} & x_1 + x_2 + x_3 + x_4 & = & 1\,, \\ & x_1, x_2, x_3, x_4 & \geq & 0\,, \\ & x_5 \geq \sqrt{x_1^2 + x_3^2}\,, & & \\ & x_6 \geq \sqrt{x_2^2 + x_4^2}\,. & & \end{array} \tag{2.11}$$

The two cones are of the quadratic cone type (CT_QUAD), and the subindexes of the variables follow naturally as seen in the following Octave code.

Figure 14: Conic Quadratic Optimization (cqo1)

```
clear -v cqo1;
cqo1.sense = "min";
cqo1.c = [0 0 0 0 1 1];
cqo1.A = sparse([1 1 1 1 0 0]);
cqo1.blc = 1;
cqo1.buc = 1;
cqo1.blx = [0 0 0 0 -Inf -Inf];
cqo1.bux = Inf(1,6);
cqo1.cones{1}.type = "CT_QUAD";
cqo1.cones{1}.sub  = [5 1 3];
cqo1.cones{2}.type = "CT_QUAD";
cqo1.cones{2}.sub  = [6 2 4];
r = mosek(cqo1);
```

□

From this example the input arguments for a conic program (2.8) follow easily (refer to Figure 1, 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 cell object ***cones***.

The cone descriptions in *cones* should all be structure-typed objects. These structures should specify the cone type in the string ***type***, being either quadratic "CT_QUAD" or rotated quadratic "CT_RQUAD". They should also contain the integer vector ***sub***, specifying the subset of variables belonging to the cone in a proper ordering. The i'th element of *sub* will be the index of the variable referred by x_i in the cone definitions (2.9) and (2.10). As an examples, the rotated quadratic cone with subindexes `[4,6,2,3]` would define the cone

$$\mathcal{C}_t = \left\{ x \in \mathbb{R}^4 \;:\; 2x_4 x_6 \geq x_2^2 + x_3^2,\; x_4 \geq 0,\; x_6 \geq 0 \right\}. \tag{2.12}$$

▶ 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 (`""`, `[]`, `{}`, `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 1). 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[16] 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. Beware that if you wish to check for NaN, you should use the `'isnan'` function as the comparison operators such as `'=='` and `'isequal'` will not work.

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

[16] 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 15 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 the optimal solution.

Figure 15: Primal Solution (cqo1)

```
r.sol.itr = {
 solsta = OPTIMAL
 prosta = PRIMAL_AND_DUAL_FEASIBLE

 skc = {                    skx = {
    [1,1] = EQ                 [1,1] = SB
 }                             [1,2] = SB
                               [1,3] = SB
                               [1,4] = SB
                               [1,5] = SB
                               [1,6] = SB
                            }
 xc = 1.0000
 xx =
   0.25000   0.25000   0.25000   0.25000   0.35355   0.35355
 ...
}
```

□

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[17], and should always be verified before the returned solution values are used (see Section 2.2.5).

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

The solution also contains status keys for both variables and constraints (called ***skx*** and ***skc***). Each status key can take the value of any of the following 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 practice never returned by MOSEK).

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 2.3.3.

2.3.3 Duality

The dual problem corresponding to the primal conic quadratic problem (2.8) defined in Section 2.3.2, is shown below in (2.13). 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 $\mathcal{C}^*$.

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c + (u^c)^T s_u^c + (l^x)^T s_l^x + (u^x)^T s_u^x + c_0 \\ \text{subject to} & A^T(s_l^c - s_u^c) + s_l^x - s_u^x + s_n^x = c\,, \\ & s_l^c, s_u^c, s_l^x, s_u^x \geq 0\,, \\ & s_n^x \in \mathcal{C}^*\,. \end{array} \tag{2.13}$$

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 (2.13) 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 16. 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 (2.11), 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 16: Dual Solution (cqo1)

```
r.sol.itr = {
 solsta = OPTIMAL
 prosta = PRIMAL_AND_DUAL_FEASIBLE

 ...

 slc = 0.70711
 suc = 0
 slx =
   Columns 1 through 5:
     9.7619e-10   9.7618e-10   9.7619e-10   9.7618e-10   0.0000e+00
   Column 6:
     0.0000e+00
 sux =
   0   0   0   0   0   0
 snx =
  -0.70711  -0.70711  -0.70711  -0.70711   1.00000   1.00000
}
```

□

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[18].

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[19].

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.

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

[19] 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 Octave using the piece code shown in Figure 17, where x_1 and x_2 are pointed out as integer variables.

Figure 17: Mixed Integer Optimization (milo1)

```
clear -v milo1;
milo1.sense = "max";
milo1.c = [1 0.64];
milo1.A = sparse([50 31;
                   3 -2]);
milo1.blc = [-Inf -4];
milo1.buc = [250 Inf];
milo1.blx = [0 0];
milo1.bux = [Inf Inf];
milo1.intsub = [1 2];
r = mosek(milo1);
```

Figure 18: Output structure (milo1)

```
r.sol.int = {
 solsta = INTEGER_OPTIMAL
 prosta = PRIMAL_FEASIBLE

 skc = {
    [1,1] = UL
    [1,2] = SB
 }
 skx = {
    [1,1] = SB
    [1,2] = LL
 }
 xc =
   250    15
 xx =
   5    0
}
```

□

The input arguments follow those of a linear or conic program with the additional identification of the integer variables (refer to Figure 1). The integer 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 18, 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[20].

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 19: Hot-starting from initial guess (milo1)

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

% Try to gues the optimal solution
clear -v myint;
myint.xx = [5 0];

% Hot-start the mixed integer optimizer
milo1.sol.int = myint;
r = mosek(milo1);
```

□

[20] 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-valued 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 integer 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 Parameters in MOSEK

2.5.1 Setting the Parameters

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[21], 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 Octave 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 20: Suppressing the optimization log with parameter 'LOG'

```
clear -v lo1;
lo1.sense = "max";
lo1.c = [3 1 5 1];
lo1.A = sparse([3 1 2 0;
                2 1 3 1;
                0 2 0 3]);
lo1.blc = [30 15 -Inf];
lo1.buc = [30 Inf 25];
lo1.blx = [0 0 0 0];
lo1.bux = [Inf 10 Inf Inf];
lo1.iparam.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*.

□

[21] 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 1). The structure *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 decimal-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 21: Selecting the dual simplex method with parameter 'OPTIMIZER'

```
clear -v lo1;
lo1.sense = "max";
lo1.c = [3 1 5 1];
lo1.A = sparse([3 1 2 0;
                2 1 3 1;
                0 2 0 3]);
lo1.blc = [30 15 -Inf];
lo1.buc = [30 Inf 25];
lo1.blx = [0 0 0 0];
lo1.bux = [Inf 10 Inf Inf];
lo1.iparam.OPTIMIZER = "OPTIMIZER_DUAL_SIMPLEX";
r = mosek(lo1);
```

□

A parameter can be removed with the `rmfield` function according to the Octave language definition. Setting a parameter an empty definition (`""`, `[]`, `{}`, `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', 'opf', 'mps' and 'mbt'), to and from the problem descriptions (Octave-variables) that the `mosek` function can accept (refer to Figure 1).

When exporting to a file, some of these formats have more limitations than others. The 'lp' format does not support conic constraints, parameter settings or initial solutions, while the 'mps' format just does not support initial solutions. Other than that, all formats handles the full problem description. Note that for formats supporting parameter settings, either none or the entire list of all parameters in the MOSEK optimization library are exported, meaning that importing parameters to Octave can be lead to a very large problem description. If what you want is to save the constructed Octave-variables exactly as they are, you should instead use the `save` command being part of the Octave language definition.

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 support modeling formats, the 'mps' format is used by default. Notice, 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 `edit` command, the file is displayed in the Octave-console.

Figure 22: Exporting a Linear Optimization Problem

```
clear -v lo1;
lo1.sense = "max";
lo1.c = [3 1 5 1];
lo1.A = sparse([3 1 2 0;
                2 1 3 1;
                0 2 0 3]);
lo1.blc = [30 15 -Inf];
lo1.buc = [30 Inf 25];
lo1.blx = [0 0 0 0];
lo1.bux = [Inf 10 Inf Inf];

r = mosek_write(lo1, "lo1.opf")
edit("lo1.opf")
```

□

A problem description can also be imported with the `mosek_read` command taking the filepath as input. If the file extension is not recognized as one of the support 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 We will now try to import the 'lo1' optimization model that was exported in Figure 22. 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 either solutions or parameters from the file. Checking that the response code is fine, we print a message and assign the 'lo1' variable.

Figure 23: Importing a Linear Optimization Problem

```
r = mosek_read("lo1.opf", struct("usesol", false, "useparam", false));

if (r.response.code == 0)
  disp("Successfully read the optimization model");
  lo1 = r.prob;
endif
```

□

Exporting and importing is quite easy as have been shown. The option `usesol` (by default true) can be used in both `mosek_write` and `mosek_read` to indicate whether or not the solution, if any is specified, should be included in the export-import procedure. 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. Similarly, the option `useparam` (by default false) indicates whether or not to include the parameter settings. If you try to read parameters from a file that doesn't specify any, the imported problem description will simply contain the default values of the MOSEK optimization library.

▶ Exporting from the `mosek` function

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'). Notice 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' format supports solutions, and that all other modeling formats would just 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 OctMOSEK 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 1, 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[22]. In many cases the integer-valued 'LOG'-parameter provides sufficient control over the logging details and can be set as shown in Figure 20, 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 interface will terminate as quickly as possible.

`NB!` Nothing is returned if <CTRL> + <C> is pressed while optimizing..
(neither response codes nor identified solutions)

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[23]. Each criterion can be modified through a parameter (see Section 2.5 for help on this). As an example, the integer-valued parameters *SIM_MAX_ITERATIONS* and *INTPNT_MAX_ITERATIONS* respectively controls the maximum numbers of simplex and interior-point iterations, while the double-valued 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.

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

2.7.4 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 Octave-to-MOSEK Interface to join our ***central MOSEK discussion group***[24]. 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 prefered form of contact as it reaches a broader user community which may be able to help or provide feedback to your posts.

[24] Check out: groups.google.com/group/mosek

3 Developers Corner

3.1 Introduction

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

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. Two important references proved to be useful in the development of this interface.

- The MOSEK C API Reference[25]
- The Octave Oct-file Manual[26]

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 this Octave-package.

3.2 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[27]. This is a short list of the 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.

[25] Check out: mosek.com → Documentation → C API manual → API reference.
[26] Check out: gnu.org/software/octave → Manual → Appendix: Oct-Files.
[27] Check out: mosek.com → Optimization toolbox for MATLAB.

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. **Solving optimization tasks in parallel**: When using column generation the master problem sometimes break down into smaller independent subproblems. With modern multi-core computer architectures it would be relevant to solve multiple independent problems in parallel with one call to the *mosek* function. This is not yet supported by the interface.

4. **Custom Octave-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 would be useful to make this possibility available from Octave, such that a callback option pointing to an Octave-function could be specified.

5. **Requesting more information**: The MOSEK optimization library actually contains far more data on the problem and optimization procedure than can currently be reached through the interface. This might be interesting to look into.

A Input and Output Argument 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 structure 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
`..blc`	Constraint lower bounds
`..buc`	Constraint upper bounds
`..blx`	Variable lower bounds
`..bux`	Variable upper bounds
`..cones`	Conic constraints
`....{i}.type`	Cone type
`....{i}.sub`	Cone variable indexes
`..intsub`	Integer variable indexes
`..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 ***blc*** and ***buc*** describes the linear constraints of the problem, while variable bounds are given by ***blx*** and ***bux***. 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 2.3.1.

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 arguments ***iparam***, ***dparam*** and ***sparam*** are used to specify lists of integer-, double- and string-valued 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 structure. 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
`..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 2.5.

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*, 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 ***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 keys
`......skc`	Constraint bound 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

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 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).

▶ `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.

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

Outputs a model of an optimization problem in any standard modelling fileformat (e.g. lp, opf, mps, mbt, etc.), controlled by a set of options. The modelling 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 structure describing the optimization problem using the same fields as for the `mosek` function on page 52.

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 structure. 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

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 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

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.

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

Interprets a model from any standard modelling 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 structure. 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

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 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

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 structure describing the imported optimization problem. It has the same fields as the input variable 'problem' for the `mosek` function on page 52.

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[28]. Transformations do not have to be hard, and some non-linear constraints easily take the shape of a quadratic cone (see Section 2.3.2 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.

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

□

[28] 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.

$$\begin{array}{rcl} s^{2^n} & \leq & 2^{n2^{n-1}} \left[y_1 \, y_2 \, \cdots \, y_{2^n} \right] \\ s^{2^n} & \leq & 2^{(n-1)2^{n-1}} \left[(2y_1 y_2) \, (2y_3 y_4) \, \cdots \, (2y_{(2^n-1)} y_{2^n}) \right] \\ s^{2^n} & \leq & 2^{(n-1)2^{n-1}} \left[x_1^2 \, x_2^2 \, \cdots \, x_{2^{n-1}}^2 \right] \\ s^{2^{(n-1)}} & \leq & 2^{(n-1)2^{n-2}} \left[x_1 \, x_2 \, \cdots \, x_{2^{n-1}} \right] \end{array} \tag{B.4}$$

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.

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

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.