Users Guide to the

Octave-to-MOSEK Interface

Made by:
Henrik Alsing Friberg, MOSEK ApS
hap@mosek.com

April 4, 2011

Contents

1 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 oct-files. The interface is simple to learn and utilize despite its many features, and thus provides a perfect tool for research and educational projects[1]. Notice however that the project is not part of the MOSEK product line, does not reflect the full capabilities of the MOSEK Optimization library, and is not guaranteed to stay updated nor backward compatible. It merely provides the basic functionality for optimization, on which the user is welcome to extend upon.

2 Setting up the interface

In the project folder you will find the *mosek.cc* source file containing the C++ code of the Octave-to-MOSEK interface. Before using the interface you will have to compile this source file to make the corresponding oct-file. This is done from Octave using the *mkoctfile* command as shown below (use '-lmosek64' with 64-bit versions of MOSEK).

Figure 1: Compiling the source file

```
clear -f mosek;
mkoctfile mosek.cc ...
    '-I <INSTALL_DIR>/mosek/<VERSION>/tools/platform/<PLATFORM>/h/' ...
    '-L <INSTALL_DIR>/mosek/<VERSION>/tools/platform/<PLATFORM>/bin/' ...
    '-lmosek';
```

The first line clears any previous defined function named *mosek* from the caches of Octave so as to prepare for the new definition. Next, the source is compiled here assuming that the current directory contains *mosek.cc*. The interface also has to be linked with the MOSEK, and thus the placeholder *<INSTALL_DIR>* is defined as the directory to which the MOSEK was installed, *<VERSION>* is the version of MOSEK installed, and *<PLATFORM>* is the system platform identifier. If the license file is located within the mosek directory you may find it helpful to look at the path specified by the environment variable *MOSEKLM_LICENSE_FILE* (e.g. using the *getenv* command from Octave). A successful installation of MOSEK would have defined such a variable definition. More help can be found at *http://www.mosek.com*.

When the source file is compiled, the output will be a file called *mosek.oct*. As long as this file is visible to Octave from the load path or current working directive, it will be accessible through its function name *mosek*.

[1]A MOSEK optimization package is required - check *mosek.com* for trial and free academic licenses.

3 A guided tour

3.1 Help: How do I use the interface?

This Octave-to-MOSEK interface includes only one function, namely *mosek*, able to handle a broad range of convex optimization problems using one of the optimizers from MOSEK. This general purpose function takes a structure as input containing the problem and interface configuration, and returns another structure containing details of the solution and optimization process. Opening Octave and typing the command:

help mosek

gives you the list of recognized input arguments along with an outline of the output structure as shown in Figure 2. This list will also be shown if you fail to call *mosek* correctly. Notice that if the *mosek* function is not recognized in Octave, or if you are not able to see the help information, you should go back to Section 2 and check your installation.

Notice that many of the input and output arguments are probably not relevant for your project and so will not be explained here, but instead elaborated in the sections to which they belong. A brief summary and references to more information can however be found in Appendix A.

Figure 2: Help information (input and output arguments)

```
r = mosek(problem)
------------------------------------------------------------
problem                   STRUCTURE
..sense                   STRING
..c                       COLUMN VECTOR
..c0                      SCALAR          (OPTIONAL)
..A                       SPARSE MATRIX
..blc                     COLUMN VECTOR
..buc                     COLUMN VECTOR
..blx                     COLUMN VECTOR
..bux                     COLUMN VECTOR
..cones                   CELL ARRAY      (OPTIONAL)
....{i}.type              STRING
....{i}.sub               COLUMN VECTOR
..intsub                  COLUMN VECTOR   (OPTIONAL)
..iparam/dparam/sparam    STRUCTURE       (OPTIONAL)
....<MSK_PARAM>           STRING/SCALAR   (OPTIONAL)
..sol                     STRUCTURE       (OPTIONAL)
....itr/bas/int           STRUCTURE       (OPTIONAL)
..verbose                 SCALAR          (OPTIONAL)
..writebefore             FILEPATH        (OPTIONAL)
..writeafter              FILEPATH        (OPTIONAL)

r                         STRUCTURE
..sol                     STRUCTURE
....itr/bas/int           STRUCTURE       (SOLVER DEPENDENT)
......solsta              STRING
......prosta              STRING
......skx                 CELL ARRAY
......skc                 CELL ARRAY
......xx                  COLUMN VECTOR
......xc                  COLUMN VECTOR
......slc                 COLUMN VECTOR   (NOT IN int)
......suc                 COLUMN VECTOR   (NOT IN int)
......slx                 COLUMN VECTOR   (NOT IN int)
......sux                 COLUMN VECTOR   (NOT IN int)
......snx                 COLUMN VECTOR   (NOT IN int OR bas)
```

3.2 Linear Programming

3.2.1 Solving LP problems

Linear optimization problems[2] 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.

$$\begin{array}{lcccccl} \text{minimize} & & & c^T x + c_0 & & & \\ \text{subject to} & l^c & \leq & Ax & \leq & u^c \, , & \\ & l^x & \leq & x & \leq & u^x \, . & \end{array} \tag{3.1}$$

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:

$$\begin{array}{lcccccccccc} \text{maximize} & 3x_1 & + & 1x_2 & + & 5x_3 & + & 1x_4 & & \\ \text{subject to} & 3x_1 & + & 1x_2 & + & 2x_3 & & & = & 30 \\ & 2x_1 & + & 1x_2 & + & 3x_3 & + & 1x_4 & \geq & 15 \\ & & & 2x_2 & & & + & 3x_4 & \leq & 25 \end{array} \tag{3.2}$$

having the bounds

$$\begin{array}{ccccc} 0 & \leq & x_1 & \leq & \infty \\ 0 & \leq & x_2 & \leq & 10 \\ 0 & \leq & x_3 & \leq & \infty \\ 0 & \leq & x_4 & \leq & \infty \end{array} \tag{3.3}$$

This is easily programmed in Octave as shown in Figure 3. The first line clears any previous definitions of the variable *lo1*, preparing for the new problem definition. The problem is then defined and finally solved on the last line.

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

Figure 3: 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 2 on setting up the interface.

□

From this example the input arguments for the linear program (3.1) follows easily (refer to the definitions of input arguments in Figure 2, Section 3.1).

Objective The string ***sense*** is the objective goal and could be either "minimize", "min", "maximize" or "max". The dense column 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 (3.1), that will be assumed zero if not specified.

Constraint Matrix The sparse matrix ***A*** is the constraint matrix of the problem with the coefficients written rowwise. Notice that for larger problems it may be more convenient to define an empty sparse matrix and specify the nonzero elements one at a time $A(i, j) = a_{ij}$, rather than writing out the full matrix as done in the 'lo1' example.

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 column vectors. These are equivalent to the bounds of problem (3.1), namely l^c, u^c, l^x and u^x.

▶ Errors and warnings

If the *mosek* function is executed with a problem definition 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 definition 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, or if the interface has to transform 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 occured and later summarized just before the interface returns.

`NB!` A faulty problem definition may solve smoothly ... with half of it misintepreted.
(always check for warnings in the log)

▶ 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.

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 4 and Figure 5 tthe 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 4: Primal Solution I (lo1)

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

 skc = {
   [1,1] = EQ
   [2,1] = SB
   [3,1] = UL
 }

 skx = {
   [1,1] = LL
   [2,1] = LL
   [3,1] = SB
   [4,1] = SB
 }

 xc =
   30.000
   53.333
   25.000

 xx =
   1.0441e-08
   2.8561e-08
   1.5000e+01
   8.3333e+00

 ...
}
```

Figure 5: Primal Solution II (lo1)

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

 skc = {
   [1,1] = EQ
   [2,1] = BS
   [3,1] = UL
 }

 skx = {
   [1,1] = LL
   [2,1] = LL
   [3,1] = BS
   [4,1] = 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 here considering the 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 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[3].

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

The solution also contain 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 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 3.2.2.

3.2.2 Duality

The dual problem corresponding to the primal problem (3.1) defined in Section 3.2.1, is shown below in (3.4). 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

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c + (u^c)^T s_u^c + (l^x)^T s_l^x + (u^x)^T s_u^x + c_0 \\ \text{subject to} & \quad A^T(s_l^c - s_u^c) + s_l^x - s_u^x \;=\; c\,, \\ & \qquad\quad s_l^c, s_u^c, s_l^x, s_u^x \geq 0\,. \end{array} \tag{3.4}$$

Recall that equality constraints had to be specified using two inequalities (with $l^c = u^c$) by definition of the primal problem (3.1). 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 $s_l^c - s_u^c$. It is not always recommended to do so however, as it is 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 solutions to the 'lo1' example that was previously omitted, are now shown in Figure 6 and 7. 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 (3.2.1), 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 6: Dual Solution I (lo1)

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

 ...

 y =
   2.5000e+00
   1.5575e-09
   3.3333e-01

 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 7: Dual Solution II (lo1)

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

 ...

 y =
   2.50000
  -0.00000
   0.33333

 slc =
  -0
  -0
  -0

 suc =
  -2.50000
  -0.00000
  -0.33333

 slx =
  -4.50000
  -2.16667
  -0.00000
  -0.00000

 sux =
  -0.0000e+00
  -0.0000e+00
  -0.0000e+00
  -1.1102e-16
}
```

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.

□

3.2.3 Switching Optimizer

The integer parameter[4] *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 an optimizer on its own. Currently MOSEK always selects the interior-point algorithm for linear programming problems and this also performs especially well for large optimization problems, but for small to medium sized problems the simplex method for example might be beneficial to switch over to.

To solve a linear programming problem using another user-specified optimizer, the *OPTIMIZER* parameter can be set to one of the following reference-strings[5]:

`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 3.2.1 using the primal simplex method. This is specified by setting the integer parameter to the reference-string "OPTIMIZER_PRIMAL_SIMPLEX" as shown in Figure 8, adding a single line to the 'lo1' definition from earlier.

[4]Check out Section 3.6 for more details on parameter settings.
[5]More values for the *MSK_IPAR_OPTIMIZER* parameter available at:
mosek.com → Documentation → Optimization tools manual → Parameter reference.

Figure 8: 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.param.IPAR_OPTIMIZER = "OPTIMIZER_PRIMAL_SIMPLEX";
r = mosek(lo1);
```

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 9.

Figure 9: 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.
...
```

□

3.2.4 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 *$sol*, and do not consider the solution and problem statuses within. These may however be specified anyway for the convenience of the user, and warnings will only be given if no useful information could be given to the MOSEK optimizer when *$sol* has a non-empty definition and hot-start is 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

$$\begin{array}{lrcrcrcrcrcl} \text{maximize} & 3x_1 & + & 1x_2 & + & 5x_3 & + & 1x_4 & - & 1x_5 & & \\ \text{subject to} & 3x_1 & + & 1x_2 & + & 2x_3 & & & - & 2x_5 & = & 30 \\ & 2x_1 & + & 1x_2 & + & 3x_3 & + & 1x_4 & - & 10x_5 & \geq & 15 \\ & & & 2x_2 & & & + & 3x_4 & + & 1x_5 & \leq & 25 \end{array} \tag{3.5}$$

having the bounds

$$\begin{array}{ccccc} 0 & \leq & x_1 & \leq & \infty \\ 0 & \leq & x_2 & \leq & 10 \\ 0 & \leq & x_3 & \leq & \infty \\ 0 & \leq & x_4 & \leq & \infty \\ 0 & \leq & x_5 & \leq & \infty \end{array} \tag{3.6}$$

which is equal to the 'lo1' problem (3.2.1), except that a new variable x_5 has been added. To hot-start from the solution of Figure 3, we can expand the problem definition and basic solution to include x_5 as shown.

Figure 10: Hot-starting when a new variable is added

```
% Solve the 'lo1' problem (not shown here)

% Append the 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;

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

% Extend the variable basis guess
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.param.IPAR_OPTIMIZER = "OPTIMIZER_PRIMAL_SIMPLEX";
lo1.sol.bas = bas;
r2 = 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 3), we fix the variable $x_4 = 2$, and hot-start using

Figure 11: Hot-starting when a variable has been fixed

```
% Solve the 'lo1' problem (not shown here)

% Fix a 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.param.IPAR_OPTIMIZER = "OPTIMIZER_PRIMAL_SIMPLEX";
lo1.sol.bas = bas;
r2 = 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 3), we add the constraint $x_1 + x_2 \geq 2$, and hot-start using

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

```
% Solve the 'lo1' problem (not shown here)

% Append the constraint to the problem
lo1.A(end+1,:) = sparse([1,1,0,0]);
lo1.blc(end+1) = 2;
lo1.buc(end+1) = inf;

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

% Extend the constraint basis guess
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.param.IPAR_OPTIMIZER = "OPTIMIZER_PRIMAL_SIMPLEX";
lo1.sol.bas = bas;
r2 = 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 10.

Figure 13: Hot-starting using an initial guess

```
% Define the 'lo1' problem (not shown here)

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

% Guess on the basis solution status keys
clear -v bas;
bas.skc = stkeys([5,1,4]);
bas.skx = stkeys([3,3,1,1]);

% Hot-start the simplex optimizer
lo1.param.IPAR_OPTIMIZER = "OPTIMIZER_PRIMAL_SIMPLEX";
lo1.sol.bas = bas;
r2 = mosek(lo1);

% Indexes of status keys can be found using, e.g.
%  find(strcmp(stkeys, 'LL'), 1)
% and index vectors can be recovered by, e.g.
%  cellfun(@(sk) find(strcmp(stkeys,sk),1), r2.sol.bas.skc)
```

3.3 Conic Quadratic Programming

3.3.1 Solving CQP problems

Conic programming[6] is a generalization of linear programming that also encapsulate the complexity of quadratic and all-quadratic programming. Conic problems posses the properties of strong duality and can be written as shown below. This is the primal problem.

$$\begin{array}{lcccc} \text{minimize} & & c^T x + c_0 & & \\ \text{subject to} \;\; l^c & \leq & Ax & \leq & u^c \\ \phantom{\text{subject to}} \;\; l^x & \leq & x & \leq & u^x \\ & & x \in \mathcal{C} & & \end{array} \tag{3.7}$$

The convex cone $\mathcal{C}$ can 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 (3.7).

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

$$\mathcal{C}_t = \left\{ x \in \mathbb{R}^{n_t} \; : \; x_1 \geq \sqrt{\sum_{j=2}^{n_t} x_j^2} \right\} \tag{3.8}$$

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

$$\mathcal{C}_t = \left\{ x \in \mathbb{R}^{n_t} \; : \; 2x_1 x_2 \geq \sum_{j=3}^{n_t} x_j^2, \; x_1 \geq 0, \; x_2 \geq 0 \right\} \tag{3.9}$$

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

[6] 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, nonnegative variables and two cones:

$$\begin{array}{lrcl} \text{minimize} & x_5 + x_6 & & \\ \text{subject to} & x_1 + x_2 + x_3 + x_4 & = & 1 \\ & x_1, x_2, x_3, x_4 & \geq & 0 \\ & x_5 \geq \sqrt{x_1^2 + x_3^2} & & \\ & x_6 \geq \sqrt{x_2^2 + x_4^2} & & \end{array} \tag{3.10}$$

The two cones are of the quadratic cone type (CT_QUAD), and the subindexes of the variables follows 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*ones(6,1);
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 (3.7) follows easily (refer to Figure 2, Section 3.1). The objective function, the linear constraints and variable bounds should all be specified as for linear programs (see Section 3.2). The only addition is the cones which should be added to the cell array ***cones*** of structure-typed elements.

Each of the structures in *cones* should specify the cone type as the string ***type***, either being quadratic "CT_QUAD" or rotated quadratic "CT_RQUAD". They should also contain the column vector ***sub***, specifying the subset of variables belonging to the cone. The i'th element of *sub* will be the index of the variable referred by x_i in the cone definitions (3.8) and (3.9). As an examples, the rotated quadratic cone with subindexes {4,6,2,3} would define the cone

$$\mathcal{C}_t = \left\{ x \in \mathbb{R}^4 \ : \ 2x_4x_6 \geq x_2^2 + x_3^2, \ x_4 \geq 0, \ x_6 \geq 0 \right\} \tag{3.11}$$

▶ Errors and warnings

If the *mosek* function is executed with a problem definition 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 definition 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 part of the problem definition was not recognized or otherwise ignored. Errors will always interrupt the optimization process whereas warnings will not. Since warnings can hold valuable information and may be important to notice, they are first printed in the log as soon as they occur and then later summarized just before the interface returns.

`NB!` A faulty problem definition may solve smoothly ... with half of it ignored.
(always check for warnings in the log)

▶ 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 = {
   [1,1] = EQ
 }

 skx = {
   [1,1] = SB
   [2,1] = SB
   [3,1] = SB
   [4,1] = SB
   [5,1] = SB
   [6,1] = SB
 }

 xc = 1

 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 $\boldsymbol{xx}$) and the activity of each constraint, $\boldsymbol{xc}$, defined by $x^c = Ax$. From the solution status (called $\boldsymbol{solsta}$) 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 the problem is either ill-posed, infeasible or unbounded (dual infeasible). This can be read from the problem status (called $\boldsymbol{prosta}$). The solution and problem status are based on certificates found by the MOSEK Optimization library[7].

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

The solution also contain status keys for both variables and constraints (called $\boldsymbol{skx}$ and $\boldsymbol{skc}$). 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 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 3.3.2.

3.3.2 Duality

The dual problem corresponding to the primal conic quadratic problem (3.7) defined in Section 3.3.1, is shown below in (3.12). Notice that the coefficients of the dual problem is the same as those used in the primal problem. Matrix A for example is still the constraint matrix of the primal problem, merely transposed in the dual problem formulation.

The dual problem is very similar to that of linear programming (Section 3.2.2), except for the added variable s_n^x belonging to the dual cone of $\mathcal{C}$ given by $\mathcal{C}^*$.

$$\begin{array}{ll} \text{maximize} & (l^c)^T s_l^c + (u^c)^T s_u^c + (l^x)^T s_l^x + (u^x)^T s_u^x + c_0 \\ \text{subject to} & \begin{array}{rcl} A^T y + s_l^x - s_u^x + s_n^x & = & c \\ -y + s_l^c - s_u^c & = & 0 \\ s_l^c, s_u^c, s_l^x, s_u^x \geq 0 & & \\ s_n^x \in \mathcal{C}^* & & \end{array} \end{array} \tag{3.12}$$

The dual cone $\mathcal{C}^*$ always has the same number of variables as its primal $\mathcal{C}$, and is in fact easily expressed for the three convex cone types supported by MOSEK. For the $\mathbb{R}$-cone its dual is the single point in space with dual variables s_n^x all zero, such that (3.12) coincides with the dual linear program (3.4), 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, are now shown in Figure 16. The dual variables y, slc, suc, slx, sux and snx corresponds naturally to y, 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 (3.3.1), the first and only constraint is an equality constraint why its dual variable can read from the y. Notice also that since none of the primal variables have upper bounds, the corresponding dual variables sux are all fixed to zeros. And since all primal variables belong to different cones than the $\mathbb{R}$-cone, none of the snx variables are forced to zero and can take values from the larger conic subspaces.

Figure 16: Dual Solution (cqo1)

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

 ...

 y   = 0.70711
 slc = 0.70711
 suc = 0

 slx =
   9.7732e-10
   9.7732e-10
   9.7732e-10
   9.7732e-10
   0.0000e+00
   0.0000e+00

 sux =
   0
   0
   0
   0
   0
   0

 snx =
  -0.70711
  -0.70711
  -0.70711
  -0.70711
   1.00000
   1.00000
}
```

□

3.4 Quadratic Programming

3.4.1 Solving QP problems

Any convex quadratic problem can be stated on the form

$$\begin{array}{lll} \text{minimize} & 0.5\|Fx\|^2 + c^T x & \\ \text{subject to} & 0.5\|Gx\|^2 + a^T x & \leq \quad b \end{array} \tag{3.13}$$

where F and G are matrices and c and a are column vectors. Here a convex quadratic term $x^T Qx$ would be written $\|Fx\|^2$ where $F^T F = Q$ is the Cholesky factorization. For simplicity 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 (3.13) can be reformulated as

$$\begin{array}{lrcl} \text{minimize} & 0.5\|t\|^2 + c^T x & & \\ \text{subject to} & 0.5\|z\|^2 + a^T x & \leq & b \\ & Fx - t & = & 0 \\ & Gx - z & = & 0 \end{array} \tag{3.14}$$

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.

$$\begin{array}{lcccl} \text{minimize} & v + c^T x & & & \\ \text{subject to} & p + a^T x & = & b & \\ & Fx - t & = & 0 & \\ & Gx - z & = & 0 & \\ & w & = & 1 & \\ & q & = & 1 & \\ & \|t\|^2 & \leq & 2vw & v, w \geq 0 \\ & \|z\|^2 & \leq & 2pq & p, q \geq 0 \end{array} \tag{3.15}$$

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 3.3, 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.

3.5 Mixed Integer Programming

3.5.1 Solving MIP 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[8].

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 dependently on the strength of the continuous relaxation to your problem formulation - that is, how far from the optimal mixed integer solution your problem formulation is when solved without the integrality requirement.

Another 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.

The 'milo1' Example (Part 1 of 2) The following is an example of a mixed integer linear optimization problem, with two inequalities and two non-negative integer variables:

$$\begin{array}{lrcl} \text{maximize} & x_1 + 0.64x_2 & & \\ \text{subject to} & 50x_1 + 31x_2 & \leq & 250 \\ & 3x_1 - 2x_2 & \geq & -4 \\ & x_1, x_2 \geq 0 & & \text{and integer} \end{array} \tag{3.16}$$

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.

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

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

  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 2). The column vector *intsub* should simply contain indexes to the subset of variables for which integrality is required. For instance if x should be a binary $\{0,1\}$-variable, its index in the problem formulation should be added to *intsub*, and its bounds $0 \leq x \leq 1$ should be specified explicitly.

If executed correctly you should be able to see the log of the interface and optimization process printed to your screen. The output structure shown in Figure 18, will only include an integer solution 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[9].

3.5.2 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

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

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 this, that can be solved to optimality without branching, it does not really make sense to hot-start. This example however shows how one could do it.

Figure 19: Hot-starting from initial guess (milo1)

```
% Define the 'milo1' problem
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]';

% Hot-start the mixed integer optimizer
milo1.sol.int.xx = [5 0]';
r = mosek(milo1);
```

□

3.6 Parameters in MOSEK

3.6.1 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[10], and are all supported by this interface. Only a few is mentioned here.

Notice that the prefix "MSK_" required by the MOSEK C API is not required by this interface, though it would match the style of other documentations and interfaces better if done so.

The 'IPAR_LOG' 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 3.2.1, we can now try to silence the optimization process as shown below.

Figure 20: Suppressing the optimization log (IPAR_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.param.IPAR_LOG = 0;
r = mosek(lo1);
```

Notice that problems and warnings from the interface itself, will not be affected by this parameter and should be specified separately by the input argument *verbose*.

□

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

Adding parameters to the input arguments of a convex optimization problem is straightforward (refer to Figure 2). The structure *param* should specify a list of variables with a name matching that of a parameter, and a value being the parameter value. Some parameters will require a string value while others expect a scalar (either of integer or decimal value).

The 'IPAR_OPTIMIZER' Example This parameter controls the 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 3.2.1 again, only this time using the dual simplex method. This is specified by setting the parameter to the string value "OPTIMIZER_DUAL_SIMPLEX" as shown.

Figure 21: Selecting the dual simplex method (IPAR_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.param.IPAR_OPTIMIZER = "OPTIMIZER_DUAL_SIMPLEX";
r = mosek(lo1);
```

□

The interface is very tolerant when it comes to parameters and will only halt the execution with an error message if you try to set a parameter to a value that cannot be recognized as a valid one. In case the parameter value is empty, e.g. an empty array or empty string, the parameter is just ignored and takes the default value while the user is notified by a warning. A warning will also be given if the name of a parameter could not be recognized as a valid one, and the execution continues ignoring the erroneous parameter.

3.7 Logfiles and data streams

3.8 Frequently Asked Questions

3.8.1 How to make MOSEK silent?

The verbosity of the interface (how many details that are printed), can be regulated by the input argument *verbose*. The default value is 10, but as of now there only exists three message priorities: Errors=1, Warnings=2 and Info=3. Setting *verbose* to the value of 1, will for example only allow the interface to speak when errors are encountered. Notice that the output stream from MOSEK have the same priority as interface errors.

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

The verbosity of the MOSEK optimization process, can be regulated by a large range of parameters[11]. In many cases the *IPAR_LOG* parameter provides sufficient control over the logging details and can be set as shown in Figure 20, Section 3.6.

3.8.2 How to change the termination criteria?

The list of termination criteria is long and span over absolute and relative tolerances, maximum iterations, time limits and objective bounds[12]. Each criterion can be modified through a parameter (see Section 3.6 on how to specify a parameter).

As an example, *SIM_MAX_ITERATIONS* and *IPAR_INTPNT_MAX_ITERATIONS* respectively control the maximum numbers of simplex and interior-point iterations, while *DPAR_OPTIMIZER_MAX_TIME* sets the overall time limit in seconds.

3.8.3 How to obtain the result of an external model file?

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

4 Developers Corner

4.1 Introduction

In the project folder you can find the *mosek.cc* source file containing the C++ code of the Octave-to-MOSEK interface. 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 log 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[13]
- The Octave Oct-file Manual[14]

The code has been released under the ??FREE?? license, and you are welcome to extend upon this release and modify the interface to your personal needs.

4.2 Limitations and Missing Features

The interface currently provides the features needed for basic research and educational purposes, but has many shortcomings when comparing to the Matlab-to-MOSEK interface[15]. This is a short list of the missing features.

1. **Direct input of Quadratic Programs** At the moment the user will manually have to 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 support the direct input of Quadratic programs, so this should only require a small amount of coding.

2. **Direct input of any problem that can 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.

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

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. **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

Here follows a brief description of where the user can find more information on the individual input and output arguments.

The argument ***sense*** is the goal of the optimization and should indicate whether we wish to maximize or minimize the objective function given by $f(x) = c^T x$, where $\boldsymbol{c}$ is the objective coefficients. 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 3.2).

The argument ***cones*** is used for Conic Programming and has been described in Section 3.3. The issue regarding the transformation of other convex problems to the formulation of a Conic Program has been addressed in appendix B, and more specific for the transformation of Quadratic Programs in Section 3.4.

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 3.5.

The argument ***param*** is used to specify a list of parameters for the MOSEK Optimization library in order to aid or take control of the optimization process. This has been described in Section 3.6.

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 3.2.4 and for mixed integer programming in Section 3.5.2.

The argument ***verbose*** is used to specify the amount of logging information given by the interface. This is described in Section 3.6.

The output structure returned by the interface is capable of holding the three types of solutions listed below. The existence of a solution depends on the optimization problem and the algorithm used to solve it.

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 3.2 and 3.3.

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 3.2).

The solution ***int*** will exist whenever variables are required to take integer values and the corresponding Mixed Integer Program has been solved (see Section 3.5).

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[16]. Linear constraints can be added to a Conic Quadratic Program directly and does not have to be transformed. Transformations do not have to be hard, and some non-linear constraints easily take the shape of a quadratic cone (see Section 3.3.1 for definitions).

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. Notice that we from line five to six use the results of the $\sqrt{x}$ example above.

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

□

[16]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. Each line implies the next, and the inequality ends 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.

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

Notice also 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 (B.4) as shown below, and the variables s and y_j just need to be linearly coupled with x and t in order for the recursion to be usable.

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

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.

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

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 (B.4), where linear coupling between the variables needs to be applied. You will need to choose n such that $2^n \geq p + q$ in order to model this.

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

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.

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

Considering the objective function, minimization problems can be rewritten as shown below matching (B.11), and are therefore applicable to the isolation of monomials and the whole transformation used above. In this way the objective function also takes 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

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

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

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

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

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

which is equivalent to the quadratic conic optimization problem

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