SOCP

Software for Second-order Cone Programming

User's Guide

Beta Version April 1997

Miguel Sousa Lobo mlobo@isl.stanford.edu
Lieven Vandenberghe vandenbe@isl.stanford.edu
Stephen Boyd boyd@isl.stanford.edu

1 Introduction

This package contains software for solving the *second-order cone programming* (SOCP) problem

$$\begin{array}{ll} \textbf{minimize} & f^T x \\ \textbf{subject to} & \|A_i x + b_i\| \le c_i^T x + d_i \quad i = 1, \ldots, L. \end{array} \tag{1}$$

The optimization variable is the vector $x \in \mathbf{R}^m$. The problem data are $f \in \mathbf{R}^m$, and, for $i = 1, \ldots, L$, $A_i \in \mathbf{R}^{N_i \times m}$, $b_i \in \mathbf{R}^{N_i}$, $c_i \in \mathbf{R}^m$ and $d_i \in \mathbf{R}$. The norm appearing in the constraints is the Euclidean norm, *i.e.*, $\|v\| = (v^T v)^{1/2}$. If $N_i = 0$ we interpret the ith constraint as

$$0 \le c_i^T x + d_i, \tag{2}$$

i.e., a linear inequality.

We refer to [LVBL97] (distributed with this software package) for a survey of applications of second-order programming.

1.1 Overview

The package contains:

- C-source: `socp.c` and `socp.h` contain a C-function `socp` that solves the general second-order cone programming problem. See §4.
- A Matlab4 interface, which allows the user to call the code from matlab4. Compiled mex-files for some platforms are included; for other platforms you can compile the interface from its source code `socp_mex.c`. See §4.
- A Matlab routine `socp.m`. Although the mex-files can be called directly from matlab, they can also be accessed with this routine that greatly simplifies their usage. It provides default parameters, diagnostics and procedures for finding initial points. See §2.
- Matlab routine: `socp1.m`. When the problem has only one constraint, it has an analytic solution, which can be computed with this function. See §3.
- Matlab examples: we provide simple matlab routines that use `socp.m` to solve various problems (some of which are described in the survey paper [LVBL97]). See §5.
- Documentation, in postscript format: `doc.ps` (this document).
- The survey paper [LVBL97], in postscript format: `socp.ps`.

This software package is available at `http://www-isl.stanford.edu/people/boyd/group_index.html` (or via anonymous ftp from `isl.stanford.edu` in `pub/boyd/`).

1.2 How to use the package

You can use the package in different ways.

- Write a C-program that calls the function `socp` in `socp.c`, and link the code with LAPACK and BLAS. You should be able to do this on any machine with a C compiler; see §4 for more details.

- **The easiest method** (provided you have matlab4) is to use the m-file `socp.m`. `socp.m` calls the C-function via a mex-file interface. Both `socp.m` and the mex-files must be in your matlab path. See §2.

- If none of the compiled mex-files is appropriate for your machine, create a mex-file for use in matlab4 by compiling `socp_mex.c` using `cmex`, and linking it with LAPACK and BLAS (see §4 for more details).

1.3 Who can/should use the code?

The code is intended for researchers who want to apply second-order cone programming (which includes quadratically constrained quadratic programming). Our aim is to provide a program that is simple, easy to use, reasonably efficient, and useful for small and medium-sized problems (say, up to hundreds of variables).

1.4 Caveat

SOCP is a simple, general code, and does not exploit problem structure, nor sparsity.

2 Matlab function: `socp.m`

```
[x,info,z,w,hist,time] = ...
  socp(f,A,b,C,d,N,x,z,w,abs_tol,rel_tol,target,max_iter,Nu,out_mode);
```

Any of the input parameters `x, z, w, abs_tol, rel_tol, target, max_iter, Nu` and `out_mode`, and/or the output parameters `info,z,w,hist` and `time` can be omitted. They must be omitted in order from the end, *i.e.*, if a given parameter is used, all the preceeding ones must also be used. Alternatively, any of these parameters can be replaced by the matlab empty list: `[]`. In both cases, the parameter is replaced by a default value or, in the case of the initial points, an alternative procedure is used to compute them. See detailed discussion of parameters below.

The shortest calling sequence is:

```
x = socp(f,A,b,C,d,N);
```

2.1 Purpose

`socp` solves the second-order cone program

$$\begin{array}{ll} \textbf{minimize} & f^T x \\ \textbf{subject to} & \|A_i x + b_i\| \leq c_i^T x + d_i \quad i = 1, \ldots, L \end{array}$$

and its dual

$$\begin{array}{lll} \textbf{maximize} & -\sum_{i=1}^L \left(b_i^T z_i + d_i w_i \right) & \\ \textbf{subject to} & \sum_{i=1}^L \left(A_i^T z_i + c_i w_i \right) = f & \\ & \|z_i\| \leq w_i & i = 1, \ldots, L \end{array}$$

given strictly feasible primal and dual initial points.

2.2 Storage convention

The problem data $A_i \in \mathbf{R}^{N_i \times n}$, $b_i \in \mathbf{R}^{N_i}$, $c_i \in \mathbf{R}^n$, $d_i \in \mathbf{R}$, and the dual variables $z_i \in \mathbf{R}^{N_i}$, and $w_i \in \mathbf{R}$, are stored as matrices and vectors A, b, C, d, z and w, defined as follows:

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_L \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_L \end{bmatrix} \quad C = \begin{bmatrix} c_1^T \\ \vdots \\ c_L^T \end{bmatrix} \quad d = \begin{bmatrix} d_1 \\ \vdots \\ d_L \end{bmatrix} \quad z = \begin{bmatrix} z_1 \\ \vdots \\ z_L \end{bmatrix} \quad w = \begin{bmatrix} w_1 \\ \vdots \\ w_L \end{bmatrix}.$$

Note that the `socp.m` *data storage convention is different from the convention used in the C and mex functions.*

2.3 Input arguments

1. `f`: Vector of length n, specifies the primal objective.

2. `A`: Matrix of size $\sum_{i=1}^{L} N_i$ by n, as described above.

3. `b`: Vector of length $\sum_{i=1}^{L} N_i$, as described above.

4. `C`: Matrix of size L by n, as described above.

5. `d`: Vector of length L, as described above.

6. `N`: Vector of length L. The i-th element is the number of rows in A_i.

7. `x`: Vector of length n. Primal starting point. Must be strictly feasible. If `x` is omitted, a *p*hase 1 method is used to find an initial primal point (See §s-phase1).

8. `z`: Vector of length $\sum_{i=1}^{L} N_i$. Dual starting point (together with `w`).

9. `w`: Vector of length L. Dual starting point (together with `z`). `z` and `w` must be strictly dual feasible. If omitted, a big-M constraint is added to the SOCP (See §s-bigM)

10. `abs_tol`: Absolute tolerance. Default value: 10^{-6}. See discussion of convergence criteria below.

11. `rel_tol`: Relative tolerance. Default value: 10^{-4}. See discussion of convergence criteria below.

12. `target`: Target value, only used if `rel_tol`< 0.0. Default value: 0.0. See discussion of convergence criteria below.

13. `max_iter`: Maximum number of iterations. Default value: 100.

14. `Nu`: The parameter ν that controls the rate of convergence. $\nu \geq 1.0$. Default value: 10.

15. `out_mode`: Specifies what should be output in `hist`; 0– nothing (default value); 1– the duality gap, for the initial point and after each iteration; 2– the duality gap *and* the deviation from centrality, for the initial point and after each iteration. (For more on duality gap and deviation from centrality, see [LVBL97] and [VB96].)

2.4 Output arguments

1. `x`: Vector of length n. The last primal iterate x.
2. `info`: String; possible values: `'absolute accuracy reached'`, `'relative accuracy reached'`, `'target value reached'`, `'target value is unachievable'`, `'maximum iterations exceeded'`, `'error'`.
3. `z`, and
4. `w`: Vectors of length $\sum_{i=1}^{L} N_i$ and of length L. The last dual iterate z, w.
5. `hist`: See `out_mode`.
6. `time`: Vector with 3 numbers; performance statistics: `time(1)` is user time (cpu time used), `time(2)` is system time (time spent in system calls), `time(3)` is total number of iterations performed.

2.5 Convergence criteria

`socp` computes an *interval* in which the optimal value has been located. The upper bound is the primal objective $p = f^T x$ evaluated at the primal feasible solution stored in `x`; the lower bound is the dual objective $d = -\sum_{i=1}^{L}\left(b_i^T z_i + d_i w_i\right)$ evaluated at the dual feasible solution stored in `z`.

The length of this interval $(p - d)$ is called the *duality gap* associated with x and z (see [LVBL97]).

The program exits normally under five possible conditions:

- *The maximum number of iterations is exceeded.*
- *The absolute tolerance is reached:*

$$p - d \leq \texttt{abs\_tol}.$$

- *The relative tolerance is reached.* The primal objective p and the dual objective d are both positive and

$$\frac{p-d}{d} \leq \texttt{rel\_tol},$$

or the primal and dual objective are both negative and

$$\frac{p-d}{-p} \leq \texttt{rel\_tol}.$$

- *The target value is reached.* `rel_tol` is negative and the primal objective p is less than `target`.

- *The target value is unachievable.* `rel_tol` is negative and the dual objective d is greater than `target`.

The target value stopping condition is useful for feasibility problems: `socp` stops when it has either found a point with objective less than the given `target`, or has found a dual point with objective value greater than `target` (which proves the optimal objective is greater than `target`). The target value stopping condition is enabled by passing a negative relative tolerance argument, *e.g.*, `rel_tol` $= -1.0$.

2.6 Caveats

- The columns of $\begin{bmatrix} A \\ C \end{bmatrix}$ should be *linearly independent*. If this is not the case, either the problem is not dual feasible (which means the primal is unbounded) or it can be reduced to another problem with fewer variables.

  This reduction is done by default in `socp`, but requires some computational effort. If, due to the nature of your problem, you know that the matrix is full-rank, you can skip the verification and size reduction procedure by editing the appropriate line in the file `socp.m` from

  ```
  check_rank=1;
  ```

  to

  ```
  check_rank=0;
  ```

- If $\begin{bmatrix} A \\ C \end{bmatrix}$ is such that (even after the transformation above) its size is close to square, some numerical difficulties may arise. In this case you should consider using the dual problem as the primal.

- The optimization procedure uses *strictly* feasible primal and dual points, which means that the standard trick to add equality constraints (writing $Ax = b$ as the two vector inequalities $Ax - b \geq 0$, $b - Ax \geq 0$) will not work. You have to explicitly eliminate equality constraints.

2.7 Initial dual point: *Big-M*

Note: If you just want to use `socp.m` *and are not interested in how it works, you can skip this and the next sections.*

When the dual variables `z` and `w` are not specified in the call to `socp.m`, a *b*ig-M procedure is used. The original problem is extended to include a bound on the primal variable. The dual of the modified problem is such that a strictly feasible dual point can always be easily computed.

However, the solution to the original problem may lie outside of the added bound. In this case, the optimization would converge to a point on the boundary of the set defined by the new bound, and not to the desired solution. This problem is circunvented by increasing the value of the bound repeatedly, to ensure it is not active in the solution.

The modified second-order cone program solved by `socp.m` is

$$\begin{array}{ll} \textbf{minimize} & f^T x \\ \textbf{subject to} & \|A_i x + b_i\| \le c_i^T x + d_i \quad i = 1, \ldots, L \\ & \sum_{i=1}^L (c_i^T x + d_i) \le M \end{array}$$

and its dual is

$$\begin{array}{lll} \textbf{maximize} & -\sum_{i=1}^L \left(b_i^T z_i + d_i(w_i - v) \right) - Mv & \\ \textbf{subject to} & \sum_{i=1}^L \left(A_i^T z_i + c_i(w_i - v) \right) = f & \\ & \|z_i\| \le w_i & i = 1, \ldots, L \\ & v \ge 0. & \end{array}$$

A strictly feasible primal initial point is assumed.

For $\left[\begin{array}{c} A \\ C \end{array}\right]$ with independent columns, all the primal variables x_i are bounded by the additional constraint. In this case, the extra dual variables associated with the added constraint allow for the easy computation of a strictly feasible dual point.

M is iteratively increased to keep the bound inactive. Every `BigM_iter` iterations, if

$$\texttt{M} \; > \; \texttt{BigM\_K} \sum_{i=1}^{L} (c_i^T x + d_i).$$

the bound M is changed to

$$M \; = \; \texttt{BigM\_K} \sum_{i=1}^{L} (c_i^T x + d_i).$$

The default values are `BigM_iter=2` and `BigM_K=2`.

The solution returned by `socp.m` is to the original problem. (The original convergence criteria are still strictly met, since the duality gap of the modified problem is an upper bound on the duality gap of the original problem.)

2.8 Initial primal point: *Phase* 1

When the primal variable `x` is not specified in the call to `socp.m`, a *p*hase 1 procedure is used. A *feasibility problem* is first solved, and its solution is then used as an initial point for the original problem.

The feasibility problem is

$$\begin{array}{ll} \textbf{minimize} & \alpha \\ \textbf{subject to} & \|A_i x + b_i\| \le c_i^T x + d_i + \alpha \quad i = 1, \ldots, L. \end{array}$$

A strictly feasible primal point is always trivially found by taking α large enough. As for an initial dual point, the *big-M* procedure described above is used. (In practice, `socp.m` constructs the modified problem and makes a recursive call to itself.)

The convergence criteria for this problem are specified by `target=0.0` (activated by `rel_tol=-1.0`). `socp` will stop when $\alpha < 0$; or when it can be shown that, subject to the constraints, $\min \alpha \geq 0$. That is, it will stop as soon as x becomes strictly feasible for the original problem; or as soon as it can be shown that there is no such strictly feasible point.

If a strictly feasible primal point is found, it is then used as an initial primal point to solve the original problem.

As a final note, we must say that these approaches (*big-M* and *phase* 1) are certainly not as good as using an *infeasible method*. However they have the advantage of being simple, and are effective enough for small to medium sized problems (with, say, a few hundred variables).

3 Matlab function: `socp1.m`

```
[x,info,z,w,fsbl,bndd] = socp1(f,A,b,c,d);
```

`socp1` computes an analytic solution to the single constraint second-order cone program

$$\begin{array}{ll} \textbf{minimize} & f^T x \\ \textbf{subject to} & \|Ax + b\| \leq cx + d \end{array}$$

and its dual

$$\begin{array}{ll} \textbf{maximize} & -b^T z + dw \\ \textbf{subject to} & A^T z + c^T w = f \\ & \|z\| \leq w \end{array}$$

Note that here c is a row vector. The problem is defined by $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$, $c \in \mathbf{R}^n$ and $d \in \mathbf{R}$. It's variables are $x \in \mathbf{R}^n$, $z \in \mathbf{R}^m$, and $w \in \mathbf{R}$.

3.1 Input arguments

1. `f`: Vector of length n, specifies the primal objective.
2. `A`: Matrix of size m by n.
3. `b`: Column vector of length m.
4. `c`: Row vector of length n.
5. `d`: Scalar.

3.2 Output arguments

1. `x`: Vector of length n. Primal solution x, if one could be computed.
2. `info`: String; possible values: `'solution found'`, `'problem is infeasible'`, `'problem is unbounded'`, `'error'`.
3. `z`: Vector of length m, and
4. `w`: Scalar. z and w provide a dual solution, if one could be computed.
5. `fsbl`: Scalar. 1 if the problem is feasible, 0 otherwise.
6. `bndd`: Scalar. 1 if the problem has a bounded solution, 0 otherwise.

4 C-function: `socp.c`

The main routine is a C-function in `socp.c`:

```
int socp( int L, int *N, int n, double *f, double *A, double *b,
          double *x, double *z,
          double abs_tol, double rel_tol, double target, int *iter,
          double Nu,
          int *info, int out_mode, double *hist,
          double *dblwork, int *intwork )
```

To compute the amount of workspace required (`dblwork`, `ptrwork` and `intwork`), another C-function is provided in `socp.c`. This should be called before `socp`:

```
void socp_getwork( int L, int *N, int n, int max_iter, int out_mode,
                   int *mhist, int *nhist,
                   int *ndbl, int *nint )
```

4.1 Purpose

`socp` solves the second-order cone program

$$\begin{array}{ll} \textbf{minimize} & f^T x \\ \textbf{subject to} & \|A_i x + b_i\| \le c_i^T x + d_i \quad i = 1, \ldots, L \end{array}$$

and its dual

$$\begin{array}{lll} \textbf{maximize} & -\sum_{i=1}^{L} \left(b_i^T z_i + d_i w_i \right) & \\ \textbf{subject to} & \sum_{i=1}^{L} \left(A_i^T z_i + c_i w_i \right) = f & \\ & \|z_i\| \le w_i & i = 1, \ldots, L \end{array}$$

given strictly feasible primal and dual initial points.

4.2 Storage convention

The sizes of the constraints are given in a vector N, of length L. N_i defines the dimensionality of the i-th conic constraint, *i.e.*, the dimension of the argument of the norm plus one. Hence, $A_i \in \mathbf{R}^{(N_i-1)\times n}$, $b_i \in \mathbf{R}^{(N_i-1)}$, $c_i \in \mathbf{R}^n$, $d_i \in \mathbf{R}$, $z_i \in \mathbf{R}^{(N_i-1)}$, and $w_i \in \mathbf{R}$.

For the C and mex functions, A, b, and the dual variable z, are conventioned to be

$$A = \begin{bmatrix} A_1 \\ c_1^T \\ \vdots \\ A_L \\ c_L^T \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ d_1 \\ \vdots \\ b_L \\ d_L \end{bmatrix} \quad z = \begin{bmatrix} z_1 \\ w_1 \\ \vdots \\ z_L \\ w_L \end{bmatrix}.$$

Note that this is different from the data storage convention used in `socp.m`.

4.3 Arguments for `socp`

1. `L` (integer, in): The number of conic constraints.
2. `N` (pointer to integer, in): Array of length L. The$i$-th element is the dimension of i-th constraint.
3. `n` (integer, in): The number of variables.
4. `f` (pointer to double, in): Array of length n, specifies the primal objective.
5. `A` (pointer to double, in): Array of length mn, as described above (storage is one column at a time, *i.e.*, element i, j of matrix is entry $i + m * (j - 1)$ of array).
6. `b` (pointer to double, in): Array of length m, as described above.
7. `x` (pointer to double, in/out): Array of length n. On entry, a strictly feasible primal point. On exit, last primal iterate x. Meaning depends on stopping criterion (solution to primal problem, if appropriate).
8. `z` (pointer to double, in/out): Array of length m. On entry, a strictly feasible dual point, as described above. On exit, last dual iterate z. Meaning depends on stopping criterion (solution to dual problem, if appropriate).
9. `abs_tol` (double, in): Absolute tolerance. See discussion of convergence criteria below.
10. `rel_tol` (double, in): Relative tolerance. See discussion of convergence criteria below.
11. `target` (double, in): Target value, only used if `rel_tol`< 0.0. See discussion of convergence criteria below.
12. `iter` (pointer to integer, in/out): On entry, `*iter` is the maximum number of iterations, socp is aborted if more are required for convergence. On exit, `*iter` is the number of iterations performed.
13. `Nu` (double, in): The parameter ν that controls the rate of convergence. $\nu \geq 1.0$. Recommended range: 5.0–50.0.
14. `info` (pointer to integer, out): Stopping criteria that caused exiting; 0– error; 1– absolute tolerance; 2– relative tolerance; 3– target value (was achieved or is unachievable); 4–maximum iterations.
15. `out_mode` (integer, in): Specifies what should be output in `hist`; 0– nothing; 1– duality gap, for initial point and after each iteration; 2– duality gap *and* deviation from centrality, for initial point and after each iteration.
16. `hist` (pointer to double, out): Array of length greater or equal to the product of `*mhist` by `*nhist`. See `out_mode`.

17. dblwork (pointer to double): Array of doubles for workspace. See socp_getwork for length.

18. intwork (pointer to integer): Array of integers for workspace. See socp_getwork for length.

socp returns 0 if it exited due to any of the stopping criteria, or an error code from the LAPACK routine dgelss.

4.4 Arguments for socp_getwork

1. n (integer, in): Use same value as in socp.

2. L (integer, in): Use same value as in socp.

3. N (pointer to integer, in): Use same values as in socp.

4. max_iter (integer, in): Use entry value of *iter in socp.

5. out_mode (integer, in): Use same value as in socp.

6. mhist (pointer to integer, out), and

7. nhist (pointer to integer, out): The required length of hist in number of doubles is the product of mhist by nhist. (These are included to make it easy to edit the code to send other output through hist; currently the values are always 1 and max_iter + 1, since hist returns a vector with the duality gap for the initial point and after each iteration.)

8. ndbl (pointer to integer, out): Returns required length for dblwork, in number of doubles.

9. nint (pointer to integer, out): Returns required length for intwork, in number of integers.

4.5 Convergence criteria

(This is the same as for socp.m*.)*

socp computes an *interval* in which the optimal value has been located. The upper bound is the primal objective $p = f^T x$ evaluated at the primal feasible solution stored in x; the lower bound is the dual objective $d = -\sum_{i=1}^{L}\left(b_i^T z_i + d_i w_i\right)$ evaluated at the dual feasible solution stored in z.

The length of this interval $(p - d)$ is called the *duality gap* associated with x and z (see [LVBL97]).

The program exits normally under five possible conditions:

- *The maximum number of iterations is exceeded.*

- *The absolute tolerance is reached:*

$$p - d \leq \texttt{abs\_tol}.$$

- *The relative tolerance is reached.* The primal objective p and the dual objective d are both positive and

$$\frac{p-d}{d} \leq \texttt{rel\_tol},$$

or the primal and dual objective are both negative and

$$\frac{p-d}{-p} \leq \texttt{rel\_tol}.$$

- *The target value is reached.* `rel_tol` is negative and the primal objective p is less than `target`.

- *The target value is unachievable.* `rel_tol` is negative and the dual objective d is greater than `target`.

The target value stopping condition is useful for feasibility problems: `socp` stops when it has either found a point with objective less than the given `target`, or has found a dual point with objective value greater than `target` (which proves the optimal objective is greater than `target`). The target value stopping condition is enabled by passing a negative relative tolerance argument, *e.g.*, `rel_tol` $= -1.0$.

4.6 Caveats

- Little or no effort in *parameter validation* is made in these functions, so some surprises are to be expected if they are used without care. The mex and m-files include parameter validation.

- For `socp.c`, the columns of A must be *linearly independent*. If this is not the case, either the problem is not dual feasible (which means the primal is unbounded) or it can be reduced to one with fewer variables. This reduction is done automatically in the m-files.

- If A is such that its size is close to square, some numerical difficulties may arise. In this case you should consider using the dual problem as the primal.

- The *big-M* and *phase* 1 procedures are not implemented in C; a strictly feasible primal and dual initial point must be provided.

- The optimization procedure uses *strictly* feasible primal and dual points, which means that standard trick of adding equality constraints (writing $Ax = b$ as the two vector inequalities $Ax - b \geq 0$, $b - Ax \geq 0$) will not work. You have to explicitly eliminate equality constraints.

4.7 Compiling

The source code for the function `socp` is in the files `socp.c` and `socp.h`. It is written in ansi-C with calls to LAPACK and BLAS.

LAPACK can be obtained via Netlib (`http://www.netlib.org` or anonymous ftp at `ftp.netlib.org`). A set of optimized BLAS-routines should be supplied by your computer vendor. A non-optimized version can also be obtained from Netlib.

You can also create a mex-file for use in matlab4 by compiling `socp_mex.c` using `cmex`, and linking it with LAPACK and BLAS. The details of compiling the code vary with the compiler used and other factors such as where various libraries are located. We have provided a sample Makefile in the source directory. In order to compile the mex-interface, edit this Makefile as indicated, and type `make`.

Executable mex-files are provided for some platforms.

5 Example matlab files

Most of the examples provided can be easily be understood by reading the initial comments in the files. We provide here some additional comments on some of the examples.

5.1 grasp

The files `grasp_1.m` and `grasp_robust.m` in the directory `examples/grasp` demonstrate the grasping problem described in [LVBL97] (this paper includes a diagram of the problem geometry). The solution is computed by calling the script `grasp_polyhedron.m`. This is a general script that can be used to treat other grasping problems.

- `grasp_polyhedron.m`

  `grasp_polyhedron.m` is a script that transforms a robust grasping problem into an SOCP. The objective is: find the maximum scaling factor K such that there is a grasp that stabilizes all the force and torque pairs

  $$\begin{array}{l} F_{\rm ext}^i = KF_{\rm a}^i + F_{\rm b} \\ T_{\rm ext}^i = KT_{\rm a}^i + T_{\rm b} \end{array} \qquad i = 1, \ldots, m$$

  Such a grasp will be stable for any force and torque in the convex hull of the $F_{\rm ext}^i$ and $T_{\rm ext}^i$ (*i.e.*, a polyhedron in $\mathbf{R}^6$).

  `grasp_polyhedron.m` uses as input the following variables, that must be pre-defined (n is the number of contact points, and m is the number of vertices in the force/torque polyhedron):

  1. `p` (matrix, 3 by n): position relative to center of mass for each contact point.
  2. `u` (matrix, 3 by n): direction of force for each contact point.
  3. `v` (matrix, 3 by n): inward normal to the surface for each contact point.
  4. `Fa` (matrix 3 by m): vertices of polyhedron, force component (relative to its center `Fb`).
  5. `Ta` (matrix, 3 by m): vertices of polyhedron, torque component (relative to its center `Tb`).
  6. `fmax` (scalar): upper bound on forces `f`.

  The script writes its output in the following variables:

  1. `K` (scalar): the maximum scaling factor for which there is a stable grasp.
  2. `f` (vector, length n): the grasping forces in the corresponding to the max-K grasp, for each contact point.
  3. `F` (matrix, 3 by nm): the friction forces corresponding to the max-K grasp, for each contact point, and for each external loading.

Note that the script uses temporary variables, which may overwrite variables in the workspace with the same name.

- `grasp_1.m`

  `grasp_1.m` is a script that defines the simple example described in [LVBL97], and calls `grasp_polyhedron.m`. The problem is solved for different friction coefficients μ. For each value of μ we compute the maximum torque for which there is a stable grasp, with upper bounds on the grasping forces. It produces the figure shown in the paper.

- `grasp_robust.m`

  `grasp_robust.m` is a script that demonstrates robust grasping, with the approximation of a ball of possible forces by a polyhedron with m vertices. As m increases, the polyhedron converges to the ball (with probability one), and the results of the optimization also converge.

5.2 matrix-frac

The script `mf.m`, in the directory `examples/matrix-frac`, demonstrates the matrix-fractional programming problem described in [LVBL97].

A particularity of this problem is that the solution to the original problem is obtained from the dual variable:

```
xp=zeros(p,1);
for i=1:p,
xp(i)=1/2*(z((i+1)*(n+1))-w(i+1));
end;
```

The script repeats random problems over a range of sizes, and several times for each size to obtain statistical information on the performance of the optimization method.

5.3 lp

The script `lp.m`, in `examples/lp`, generates and solves random linear programs. It has the following possibilities:

- Test a range of `Nu`; this can be controled by editing the line

  `Nu=[10 20 100];`

- Test a range of problem sizes `n`; defined by the line

  `for n=20:20:100,`

- For each value of `Nu` and `n`, repeat to obtain statistical measures, average and variance; this is defined by the line

  `for k=1:11,`.

6 Acknowledgments

Research supported in part by the Portuguese Government (under Praxis XXI), AFOSR (under F49620-95-1-0318), NSF (under ECS-9222391 and EEC-9420565), and MURI (under F49620-95-1-0525).

References

[LVBL97] M. S. Lobo, L. Vandenberghe, S. Boyd, and H. Lebret. Second-order cone programming: interior-point methods and engineering applications. *Linear Algebra and Appl.*, 1997. Submitted.

[VB96] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, March 1996.