SDPPACK
USER'S GUIDE

VERSION 0.9 BETA FOR MATLAB 5.0
JUNE 26, 1997

FARID ALIZADEH, JEAN-PIERRE A. HAEBERLY, MADHU V. NAYAKKANKUPPAM, MICHAEL L. OVERTON, AND STEFAN SCHMIETA

Abstract. This report describes SDPpack Version 0.9 Beta for Matlab 5.0. This version extends the previous release for semidefinite programming (SDP) to mixed semidefinite–quadratic–linear programs (SQLP), i.e. linear optimization problems over a product of semidefinite cones, quadratic cones and the nonnegative orthant. Together, these cones make up all possible homogeneous self-dual cones over the reals. The main routine implements a primal–dual Mehrotra predictor–corrector scheme based on the XZ+ZX search direction for SDP. More specialized routines are also available, one to solve SDP's with diagonal constraints only, and one to compute the Lovász θ function of a graph, both using the XZ search direction. Routines are also provided to determine whether an SQLP is primal or dual degenerate at its solution and whether strict complementarity holds there. Primal nondegeneracy is associated with dual uniqueness and dual nondegeneracy with primal uniqueness, though these conditions are not equivalent if strict complementarity fails to hold. A routine is also provided to compute the condition number of an SQLP. The Matlab code calls `mex` files for improved performance; binaries are available for several platforms. Benchmarks show that the codes provide highly accurate solutions to a wide variety of problems.

This work was supported in part by the National Science Foundation.

Web: http://www.cs.nyu.edu/faculty/overton/sdppack/sdppack.html
NYU Computer Science Dept Technical Report, June 1997.

Contents

1. Introduction

We treat the primal mixed semidefinite–quadratic–linear program (SQLP):

$$
\begin{array}{ll}
\min & C_S \bullet X_S + C_Q^T X_Q + C_L^T X_L \\
\text{s.t.} & (A_S)_k \bullet X_S + (A_Q)_k^T X_Q + (A_L)_k^T X_L = b_k \quad k = 1, \ldots, m \\
& X_S \succeq 0, \quad X_Q \geq_Q 0, \quad X_L \geq 0
\end{array}
$$

where X_S is a block diagonal symmetric matrix variable, with block sizes N_1, N_2, $\ldots$, N_s respectively, each greater or equal to two; X_Q is a block vector variable, with block sizes $n_1, n_2, \ldots, n_q$ respectively, each greater or equal to two; and X_L is a vector of length n_0. The quantities C_S and $(A_S)_k$, $k = 1, \ldots, m$, are block diagonal matrices. The quantities C_Q and $(A_Q)_k$, $k = 1, \ldots, m$ are block vectors, and C_L and $(A_L)_k$, $k = 1, \ldots, m$, are vectors. All vectors are column vectors. The quantity $C_S \bullet X_S$ is the trace inner product (tr $C_S X_S$), i.e. $\sum_{i,j} (C_S)_{ij} (X_S)_{ij}$.

Each of the three inequalities in this primal program has a different meaning, each corresponding to a different kind of cone:

- The first kind of inequality is the semidefinite constraint. $X_S \succeq 0$ means that the matrix X_S is positive semidefinite, or equivalently that each of its blocks is positive semidefinite.

- The second kind of inequality describes the quadratic cone[1] constraints. Writing $x = X_Q$ for brevity, with block structure

$$x = \left[(x^1)^T \ (x^2)^T \ \ldots (x^q)^T\right]^T, \tag{1}$$

  where

$$x^i = \left[x_1^i \ x_2^i \ \cdots \ x_{n_i}^i\right]^T,$$

  the constraint $X_Q \geq_Q 0$, i.e. $x \geq_Q 0$ means that, for each block i,

$$x_1^i \geq \sqrt{\sum_{j=2}^{n_i} (x_j^i)^2}. \tag{2}$$

  Any convex quadratic constraint can be converted to this form.
- The third kind of inequality is the standard one: $X_L \geq 0$ means each component of X_L is nonnegative.

Thus the feasible set is a product of semidefinite, quadratic and nonnegative orthant cones, intersected with m hyperplanes. It is possible that one or more of the three parts of the SQLP is not present, i.e., any of s (the number of blocks in X_S), q (the number of quadratic blocks in X_Q), or n_0 (the length of X_L) may be zero. If $q = 0$, the SQLP reduces to an ordinary SDP and if $s = 0$ the SQLP reduces to QCLP (convex quadratically constrained linear programming).

The standard form given here is a very convenient one. The dual SQLP is

$$\begin{array}{ll} \max & b^T y \\ \text{s.t.} & \sum_{k=1}^m y_k (A_S)_k + Z_S = C_S \\ & \sum_{k=1}^m y_k (A_Q)_k + Z_Q = C_Q \\ & \sum_{k=1}^m y_k (A_L)_k + Z_L = C_L \\ & Z_S \succeq 0, \quad Z_Q \geq_Q 0, \quad Z_L \geq 0. \end{array}$$

Note again the three different kinds of inequalities on the three dual slack variables. In control applications, the first of the three is usually called a *linear matrix inequality* (LMI), since it can also be written $\sum_{k=1}^m y_k (A_S)_k \preceq C_S$.

We shall use the following notation. Let

$$\texttt{pinfeas} = \left(\sum_{k=1}^m \left[b_k - (A_S)_k \bullet X_S - (A_Q)_k^T X_Q - (A_L)_k^T X_L\right]^2\right)^{1/2} \tag{3}$$

$$\texttt{dinfeas} = \left\|C_S - Z_S - \sum_{k=1}^m y_k (A_Q)_k\right\|_F + \left\|C_Q - Z_Q - \sum_{k=1}^m y_k (A_S)_k\right\|_2 + \left\|C_L - Z_L - \sum_{k=1}^m y_k (A_L)_k\right\|_2 \tag{4}$$

$$\texttt{comp} = X_S \bullet Z_S + X_Q^T Z_Q + X_L^T Z_L \tag{5}$$

[1] Also known as the second-order cone, Lorentz cone or ice cream cone.

where $\|\cdot\|_F$ denotes the Frobenius matrix norm. Assuming the existence of a strictly feasible primal or dual point, it is well known that the optimality conditions may be expressed by the primal feasibility equation $\texttt{pinfeas} = 0$, the dual feasibility equation $\texttt{dinfeas} = 0$, and the complementarity condition $\texttt{comp} = 0$ (together with the inequality constraints). We will also wish to refer to the quantities

$$\texttt{normx} \;=\; \|X_S\|_F + \|X_Q\|_2 + \|X_L\|_2, \quad \texttt{normz} \;=\; \|Z_S\|_F + \|Z_Q\|_2 + \|Z_L\|_2. \tag{6}$$

2. Obtaining and Installing SDPpack

The current release of SDPpack is Version 0.9 Beta[2] and requires Matlab[3] Version 5.0. Users who have not yet upgraded to Matlab 5.0 should use SDPpack Version 0.8 Beta, which requires Matlab 4.2c.1 and solves standard SDP's only, not mixed SQLP's. Both versions of the package can be obtained from the **SDPpack home page** on the World–Wide Web:

`http://www.cs.nyu.edu/faculty/overton/sdppack/sdppack.html`

This page contains the complete distribution of the source code, compiled binaries (`mex` files) for several platforms, online documentation, and information regarding forthcoming releases, submission of bug reports, several test problems, etc.

After retrieving the files, use the following instructions to install the package.

Unix Platforms: (`%` denotes the shell prompt)

```
% gunzip sdppack-v0.9.tar.gz
% tar -xovf sdppack-v0.9.tar
```

Windows NT/95:

Move the ZIP file to the directory in which you want to install SDPpack (typically Matlab\toolbox). Unzip the file, making sure the directory structure is preserved (for example, if you use WinZip, make sure that the "Use Folder Names" checkbox is checked).

This will produce a directory called `sdppack`, which will contain the main routines of SDPpack, and the subdirectories `support` (support routines), `special` (specialized routines for certain problem classes), `doc` (this document), and `mex` (C sources to generate `mex` files for Matlab 5). Although the software does not require the use of `mex` files, it runs *much faster* with compiled `mex` files (binaries). There are seven `mex` files, with source file names `svec.c`, `smat.c`, `evsumdiv.c`, `lyapsol.c`, `arwmul.c`, `arwimul.c`, `qcschur.c`. These C source routines are in the `mex` subdirectory. The binaries (available separately from the SDPpack home page) have the name format `filename.mex***`, where `***` is a string that is architecture-dependent. The table below lists the binaries currently available; see the home page for updates.

Place the appropriate binary files in the main `sdppack` directory, so that they reside with the corresponding `m`–files. If you cannot find compiled MEX files for your platform, you will need to compile them yourself. A Unix make file is provided for this purpose. To compile the MEX files on a Unix platform, type

[2]Preliminary versions of this software have been distributed privately since August 1995. The first public release was Version 0.8 Beta, March 1997.

[3]Matlab is a registered trademark of The MathWorks Inc.

TABLE 1. MEX files available for SDPpack Version 0.9

Operating System	Architecture	Compiled mex files available?
IRIX–6.2	R8000/R10000	Yes
IRIX–6.3	R5000	Yes
IRIX–5.3	R4400	Yes
SunOS–4.1.4	Sparc	Yes
SunOS–5.5.1	Sparc	Yes
Windows NT/95	PC x86	Yes
MacOS (System 7)	Macintosh	Available soon

```
% make
```

after setting the current directory to the main `sdppack` directory. Depending on the C compiler you use, the switches in the command line for `mex` (see the file `Makefile`) could vary; consult the manual for your C compiler. (For SGI R10000 machines running the 64 bit version of Matlab 5.0, the flags `-64 -mips4` are needed to compile the `mex` files.)

After starting Matlab, type

```
setpath
```

to add all the necessary subdirectories to the Matlab path. Information about specific routines can then be obtained by typing `help routine_name` from within Matlab.

The following sections describe how to use the package, giving an overview of the main routines. Appendix A describes an ASCII storage format for SQLP's supported by SDPpack. Appendix B has several Matlab sessions illustrating how to use the main routines in the package. Appendix C benchmarks this release of SDPpack on a set of test problems.

3. THE SCRIPT `SQL.M` AND THE FUNCTION `FSQL.M`

The Matlab routines `sql.m` and `fsql.m` solve SQLP's using a primal–dual Mehrotra predictor–corrector scheme based on the XZ+ZX search direction[4] [1]. Its extension from SDP to SQLP is based on the discussion in [2] for quadratic cone constraints and will be described in more detail in a forthcoming technical report [3].

The simplest option for the user is to call the script `sql.m`, which automatically calls the Matlab function `fsql.m`. Additional scripts are provided to help the user set up the data, define necessary options, and initialize the variables (as described shortly). The user who requires a function interface should bypass `sql.m` and call `fsql.m` directly. In either case there are five steps to be followed:

- set up the data (the use of `makeA.m` or `import.m` simplifies this process)
- set the options (the routine `setopt.m` sets these to their default values)
- provide initial values for the variables (the routine `init.m` provides default settings)
- call either the script `sql.m` or the function `fsql.m` to solve the problem
- interpret the output

[4] Sometimes referenced as the AHO direction in the literature

We now describe each of these steps in detail.

3.1. **Preparing the data.** The problem is defined by the following data:

`A`: a structure[5] with three fields:

- `A.s`: a matrix with m rows and $\sum_i (N_i(N_i+1)/2)$ columns. The kth row holds the symmetric block diagonal matrix $(A_S)_k$ stored as a vector
- `A.q`: a matrix with m rows and $\sum_i n_i$ columns. The kth row holds the block vector $(A_Q)_k^T$
- `A.l`: a matrix with m rows and n_0 columns. The kth row holds the vector $(A_L)_k^T$

`b`: the vector b defining the dual objective function. Its length m is equal to the number of primal constraints

`C`: a structure with three fields, defining the primal objective function:

- `C.s`: the block diagonal matrix C_S
- `C.q`: the block vector C_Q
- `C.l`: the vector C_L

`blk`: a structure with three fields defining the block sizes:

- `blk.s`: a vector whose length is the number of blocks in the block diagonal matrix X_S and with entries set to the matrix block sizes $N_1, N_2, \ldots, N_s$. Note that these numbers must be all greater or equal to two, as any blocks with size one should be incorporated in the vector X_L
- `blk.q`: a vector whose length is the number of blocks in the block vector X_Q and with entries set to the vector block sizes $n_1, n_2, \ldots, n_q$. Note that these numbers must be all greater or equal to two, as each vector block has a special first component and at least one other component
- `blk.l`: the scalar n_0 (the length of X_L)

Important: If one or more of the three parts of the SQLP is not present, the corresponding field of `blk` may be set to 0, or to the Matlab empty matrix `[ ]`, or left unspecified. In any of these cases, the corresponding fields of `A` and `C`, if set, will be ignored.

Important: In order to be able to use the `mex` files, all block diagonal matrices *must* be stored in Matlab's sparse[6] format, unless they contain only one block.

In particular, `C.s` must be stored in sparse format, unless it has only one block. If it has only one block, it may be stored in either sparse or full format. The vectors `C.q`, `C.l` and `b` are stored in full format. The matrices `A.q` and `A.l` may be stored in either sparse or full format. There are four different ways to set up the matrix `A.s`, whose rows represent $(A_S)_k, k = 1, \ldots, m$:

1. Constructing `A.s` directly using the function `svec.m`, which converts block diagonal matrices to vector representation. The function `smat.m` restores the symmetric matrix from such a vector. These routines are invoked by

   $$\texttt{v} = \texttt{svec(M,blk.s)} \quad \text{and} \quad \texttt{M} = \texttt{smat(v,blk.s)}.$$

   The block diagonal matrix `M` passed to `svec.m` must be stored in sparse format if it has more than one block. By default, the blocks are assumed to be dense. If the user wishes to take advantage of the sparsity in the blocks of the matrix

[5] The fields of a structure may be assigned at the keyboard without initializing it otherwise, for example `A.q = [1 2; 2 2]`.

[6] type `help sparse` in Matlab for more information.

M, then a third, optional parameter sparseblks can be passed to svec.m. When sparseblks = 1, svec.m treats the blocks as sparse, and returns a sparse vector.

These routines preserve the inner product, i.e. if $v = \mathtt{svec}(\mathtt{M}, \mathtt{blk.s})$ and $w = \mathtt{svec}(\mathtt{N}, \mathtt{blk.s})$ for block diagonal matrices M, N, then

$$v^T w = M \bullet N.$$

2. Using the routine makeA.m, which calls svec.m, to construct A.s from given predefined matrices. This is invoked by

$$\mathtt{A.s} = \mathtt{makeA}(\mathtt{blk.s}, \mathtt{Amat})$$

where blk.s is as above and Amat is a one-dimensional cell array. The kth component of the cell array Amat is the matrix $(A_S)_k$. Cell arrays are indexed using braces $\{\cdot\}$. For example, to set $(A_S)_3$ assign a matrix value to Amat{3}; this must be in sparse format if it has more than one block. The individual blocks of the block diagonal data matrices are treated by default as being dense. If they are sparse, and the user wishes to take advantage of this sparsity, a third optional parameter spblocks may be passed to makeA.m, and this parameter should be set to the value 1. In this case the matrix A will be stored using the sparse matrix storage option.
3. Using the routine import.m to load *all* the data (A, b, C and blk) from a plain ASCII file. This is invoked by

$$[\mathtt{A}, \mathtt{b}, \mathtt{C}, \mathtt{blk}] = \mathtt{import}(\mathtt{filename})$$

The ASCII data in the file must be stored in a special compact format that is described in Appendix A. The file name must have a period and an extension (anything other than the standard Matlab extensions mat, mex etc.) following the period. The user–specified extension is important as Matlab treats file names without an extension as mat files. The routine export.m implements the reverse operation, saving a problem's data in an ASCII file, in a format recognized by import.m.
4. Loading a mat file defining the data A, b, C, blk, saved previously by Matlab's save command, using Matlab's load command. This option may be used to load all the examples (from control theory and truss topology design) benchmarked in Appendix C, for which mat files are available from the SDPpack home page.

3.2. **Setting the options.** Options are passed to the code by means of a structure opt. It is important to set the options correctly in order to take full advantage of the codes. Particular attention should be paid to the termination options opt.abstol, opt.reltol and opt.bndtol, for which appropriate values are quite problem dependent. All options are set to their default values by calling the routine setopt.m. The default values demand high accuracy; the number of iterations required to meet the termination criteria is reduced by requesting less accurate solutions.

opt.maxit: (Default value = 100)
The maximum number of iterations which may be taken by the algorithm. If validate = 1 (see below), then explicitly setting maxit = 0 results in data validation alone.

`opt.tau`: (Default value = 0.999)
The fraction of the step to the boundary of the feasible cone taken by the algorithm. This choice leads to fast convergence and is generally reliable, but may occasionally lead to failures due to short steps (see below). In many cases, the quantity `comp` (see (5)) is reduced by approximately a factor of $1/(1 - \texttt{tau})$ per iteration in the last few iterations.

`opt.steptol`: (Default value = 10^{-8})
Tolerance on the primal and the dual steplengths. If either one of these drops below `steptol`, the algorithm terminates. If `pinfeas`, `dinfeas` or `comp` is large, a restart is recommended, with either a reduced value of `tau`, or with `X` and `Z` set to larger initial values (see below). This is done automatically when the driver script `sql.m` is used, but is *not* done if the the driver script is bypassed with a direct call to the function `fsql.m`.

`opt.abstol`: (Default value = 10^{-8})
Absolute tolerance on the total error, imposing the condition

$$\texttt{pinfeas} + \texttt{dinfeas} + \texttt{comp} < \texttt{abstol}$$

(see (3), (4)).

`opt.reltol`: (Default value = 10^{-11})
Relative tolerance on the total error, imposing the condition

$$\texttt{pinfeas} + \texttt{dinfeas} + \texttt{comp} < \texttt{reltol} \times (\texttt{normx} + \texttt{normz})$$

(see (6)). `reltol` is usually set to a value smaller than that of `abstol`. Successful termination takes place when *both* the absolute and relative conditions are satisfied. Either one can be relaxed by making the corresponding tolerance large.

`opt.gapprogtol`: (Default value = 100)
Tolerance on progress; this parameter, in conjunction with `feasprogtol` (see below), determines when the algorithm should terminate if significant progress is not taking place. If `comp` is less than the previous value of `comp` divided by `gapprogtol`, then the progress is considered "sufficient". This check is performed only when `comp` has been reduced below 100 times the value of `abstol`.

`opt.feasprogtol`: (Default value = 5)
Tolerance on progress; this parameter, in conjunction with `gapprogtol` (see above), determines when the algorithm should terminate if significant progress is not taking place. If the new `pinfeas` is less than `feasprogtol` times the previous `pinfeas`, or the new `dinfeas` is less than `feasprogtol` times the previous `dinfeas`, then the loss of feasibility, if any, is considered "tolerable". Termination occurs if the loss of feasibility was not "tolerable" and the reduction in `comp` was not "sufficient" to justify this loss of feasibility. In short, for the default values, these conditions mean that we are not willing to let the algorithm continue if the primal or dual infeasibility worsened by a factor of 5 or more, unless the gap improved by a factor of at least 100. These options attempt to achieve a judicious balance between feasibility and complementarity by trading the former in return for the latter.

`opt.bndtol`: (Default value = 10^{8})
Tolerance on the norm of the solution; if any of the three terms in `normx` or in `normz` (see (6)) becomes greater than `opt.bndtol`, the algorithm terminates.

Unbounded primal (dual) feasible iterates suggest that the dual (primal) program may be infeasible.

`opt.prtlevel`: (Default value = 1)
Determines print level; setting this to 0 produces no output from `fsql.m`.[7] Setting `opt.prtlevel` to 1 produces one line of output per iteration (iteration number, primal and dual step lengths, primal and dual infeasibilities `pinfeas` and `dinfeas`, the inner product `comp`, and primal and dual objective values). Upon termination, summary information is provided by the script `sql.m` regardless of the value of `opt.prtlevel`.

`opt.validate`: (Default value = 0)
By default, several minor consistency checks on the dimension of the data are performed. Additionally, if `validate` = 1, `fsql.m` makes a check to ensure the initial `X.s` and `Z.s` conform to the block diagonal structure specified.

`opt.uselyapsol`: (Default value = -1)
Indicates whether or not to use the `lyapsol` `mex` file. This affects the speed of solving the Lyapunov systems required to construct the semidefinite component of the Schur complement. Using `lyapsol` is usually advantageous, but is not in the cases where X_S has only one block, or its blocks are all relatively small. When the default value -1 is used, the code automatically decides whether or not to use this mex file; the user may override this by specifying a value of 0 (do not use it) or 1 (use it).

3.3. **Initializing the variables.** The user must provide initial values for the variables. This may be done by calling the script `init.m`, which sets default values as follows, assuming that a scale factor `scalefac` is available in the workspace. The choice of `scalefac` is discussed further below.

- `X`, `Z`: structures, each with three fields:
  - `X.s`, `Z.s` : the block diagonal matrix variables X_S and Z_S, which must be positive definite. The default rule used by `init.m` is to set both to `scalefac` times the identity matrix.
  - `X.q`, `Z.q`: the block vectors X_Q and Z_Q, whose blocks must lie in the interior of their quadratic cones (see (2)). The default rule used by `init.m` is to set the first entry in each block to `scalefac` and the rest to zero.
  - `X.l`, `Z.l`: the vectors X_L and Z_L, which must be positive. The default rule used by `init.m` is to set all components to `scalefac`.
- `y`: The dual variable y. Any initial values may be used; `init.m` sets this to zero.

If the block diagonal matrices `X.s` and `Z.s` have more than one block, they *must* be stored as sparse matrices. If they contain a single block, it is recommended that they be provided in full format, as the solutions will most likely be full. The vectors are normally stored using full format.

The proper choice of `scalefac` is highly problem dependent. The routine `setopt.m` sets `scalefac` to a default value of 100, but it may be necessary to change this to a larger value. On the other hand, often `scalefac` = 1 is satisfactory and results in a smaller number of iterations.

[7] Matlab warnings that ill-conditioned systems are being solved may be suppressed by typing `warning off`. Such ill-conditioning is normal near the solution.

3.4. **Invoking `sql.m` or `fsql.m`.** After preparing the data, initializing the variables and setting the options, the user may simply type

```
sql
```

to solve the problem. Alternatively, the user who requires a function interface should use the function call[8]

$$[\mathtt{X}, \mathtt{y}, \mathtt{Z}, \mathtt{iter}, \mathtt{compval}, \mathtt{feasval}, \mathtt{objval}, \mathtt{termflag}] = \mathtt{fsql}(\mathtt{A}, \mathtt{b}, \mathtt{C}, \mathtt{blk}, \mathtt{X}, \mathtt{y}, \mathtt{Z}, \mathtt{opt})$$

3.5. **Interpreting the output.** The following output parameters are provided by both `sql.m` (as variables in the Matlab workspace) and `fsql.m` (as return values):

- **`X`, `y`, `Z`:** The final values of the variables. As on input, `X` and `Z` are each structures with three fields, `X.s`, `X.q` and `X.l` and `Z.s`, `Z.q` and `Z.l`, respectively. If the matrices `X.s` and `Z.s` have multiple blocks, they are stored using Matlab's sparse format. If they have only one block, then they are always full. As long as `termflag` is not equal to one (see below), `X.s` and `Z.s` are numerically positive definite in the sense that Matlab's Cholesky function `chol` does not encounter zero or negative pivots when applied to them; likewise the blocks of the vectors `X.q` and `Z.q` strictly satisfy the quadratic cone constraints and the vectors `X.l` and `Z.l` are strictly positive. If `termflag` equals one, at least one component of the computed solution is (numerically) either outside the cone or on its boundary.
- **`iter`:** The number of iterations taken by the algorithm.
- **`compval`:** A vector of length $\mathtt{iter} + 1$, with entries equal to the value of `comp` (see (5)) as a function of the iteration count. The first entry in the `compval` array is the value of `comp` corresponding to the initial point provided.
- **`objval`:** A matrix with two columns and $\mathtt{iter} + 1$ rows, whose entries are the values of the primal and the dual objectives in the first and the second columns respectively, as a function of the iteration count.
- **`feasval`:** A matrix with two columns and $\mathtt{iter} + 1$ rows, whose entries are the values of `pinfeas` and `dinfeas` (see (3) and (4)) in the first and the second columns respectively, as a function of the iteration count.
- **`termflag`:** An integer informing the user why `fsql.m` terminated, with the following meanings:
  - **$\mathtt{termflag} = 0$:** Successful termination: both the absolute and relative tolerances were satisfied.
  - **$\mathtt{termflag} = 1$:** The iterate `X` or `Z` generated by the algorithm is numerically outside the cone or on its boundary. This would not occur in exact arithmetic, and indicates that the computation has reached its limiting accuracy. This is normal, and usually means that the problem is essentially solved but the termination criteria were too stringent. If $\mathtt{prtlevel} > 0$ (see below), further information is printed. Specifically, if Matlab's Cholesky routine `chol` determines that either `X.s` or `Z.s` (or both) is not positive definite, the smallest eigenvalue of `X.s` or `Z.s` (or both) is computed (by `blkeig`: see Section 7) and printed. Likewise if `X.q`, `X.l`, `Z.q` or `Z.l` lies outside the cone or on its boundary, that information is printed. If necessary, this information can be used to shift the

[8] The user need not specify all output parameters when invoking a Matlab function: any number of leading output parameters may be requested. If only the right-hand side is specified, the function returns only the first output parameter.

solution inside the cone or onto its boundary.[9] (If `iter` $= 0$, the initial `X` or `Z` provided is not in the interior of the cone as required.)

`termflag` $= 2$: This occurs if the eigenvalue routine `blkeig` (see Section 7) determines that `Z.s` has a zero (or negative) eigenvalue, even though `chol` reported that `Z.s` is positive definite. This indicates that the algorithm has reached its limiting accuracy. The minimum eigenvalue is printed if `prtlevel` > 0. (The eigenvalues of `Z.s` (but not `X.s`) are required to compute the search direction.)

`termflag` $= 3$: Termination occurred because the Schur complement was numerically singular, i.e. the Matlab routine `lu` generated a zero pivot, making the search direction undefined. The most likely explanation is that the matrix [`A.s` `A.q` `A.l`] is rank deficient. The routine `preproc.m` can help to detect inconsistent constraints and eliminate redundant constraints (see Section 7). Otherwise, this is a rare situation which might occur close to the solution, if the termination criteria are too stringent.

`termflag` $= 4$: Termination occurred because the progress made by the algorithm was no longer significant. See the description of the input options `gapprogtol` and `feasprogtol`. This indicates that the termination criteria may be too stringent. (If `iter` $= 0$, then the initial point was probably already too close to the boundary.)

`termflag` $= 5$: Termination occurred because either the primal or the dual steplength became too small. If `pinfeas`, `dinfeas` or `comp` is large, a restart is recommended, with either a reduced value of `opt.tau`, or with `X` and `Z` reset using a larger value of `scalefac`, or both. Such a restart is done automatically when the driver script `sql.m` is used, but is the user's responsibility when the driver script is bypassed by a direct call to the function `fsql.m`. (If `iter` $= 0$, then the initial guesses `X` and `Z` were most likely too close to the boundary of the cone.)

`termflag` $= 6$: Termination occurred because the maximum number of iterations was reached. (If `maxit` $= 0$, then the data passed the validation test.)

`termflag` $= 7$: Termination occurred because data failed validation checks. This means that some input argument was not of the correct dimension or `fsql.m` was called with an incorrect number of arguments. If `validate` $= 1$, this could alternatively mean that the initial `X` or `Z` did not conform to the specified block structure.

`termflag` $= -2$: Termination occurred because at least one of the components of `normx` (see (6)) exceeded `bndtol`, indicating possible dual infeasibility.

`termflag` $= -1$: Termination occurred because at least one of the components of `normz` (see (6)) exceeded `bndtol`, indicating possible primal infeasibility.

The reader is encouraged to consult Appendix B which contains a sample Matlab session illustrating the use of the `sql` script.

[9] Unfortunately, the output of Matlab's `chol` function does not distinguish between the singular and indefinite cases. Hence the need to rely on an eigenvalue routine to make this distinction. Using `blkeig` is preferable to calling the Matlab built-in function `eig` directly, since it computes eigenvalues one block at a time.

4. Problems in Nonstandard Form

Often, problems arise in nonstandard form, but in many cases they are easily converted to the SQLP form. For example, if one wishes to place nonnegativity bound constraints on some or all of the components of X_S, one introduces a primal constraint and a new component of X_L for each such bound, *e.g.* constraining $(X_S)_{12} = (X_L)_1 \geq 0$. The disadvantage of this approach is the increase in dimension size, but the advantage is its convenience. As another example, if one has upper and lower bounds on a variable, one uses two inequality constraints: again, this approach has disadvantages, but has the virtue of simplicity. A less obvious example is how to handle mixed inequalities and equalities. For example, suppose one has the two constraints

$$\sum_{k=1}^{m} y_k F_k = F_0, \qquad \sum_{k=1}^{m} y_k G_k \preceq G_0,$$

where $\{F_k\}$ and $\{G_k\}$ are symmetric matrices. The first constraint is called an LME (linear matrix equality) and the second an LMI. One introduces one dual slack matrix for the LMI and two for the LME, giving

$$\begin{aligned} &\sum_{k=1}^{m} y_k F_k + Z_1 = F_0, \qquad &&\sum_{k=1}^{m} y_k(-F_k) + Z_2 = -F_0, \\ &\sum_{k=1}^{m} y_k G_k + Z_3 = G_0, \qquad &&Z_1 \succeq 0, Z_2 \succeq 0, Z_3 \succeq 0 \end{aligned}$$

which is then in the required dual block semidefinite form. The disadvantage of this approach is that the corresponding primal solution set is not bounded; this sometimes leads to numerical difficulties, but is worth trying.

5. Specialized Routines

Specialized routines are available for two problem classes: (i) SDP's with diagonally constrained variables, and (ii) the Lovász θ function of a graph. These routines may be found in the `special` subdirectory.[10]

5.1. **Diagonally constrained SDP's.**

For the case of diagonally constrained problems (for example, MAX–CUT relaxations), the Schur complement equations can be formed and solved very efficiently using the XZ search direction[11] [4, 5]. The specialized routines `dsdp.m` and `fdsdp.m` take advantage of this. As before, the five steps involved in setting up and solving a problem are: preparing the data, setting the options, initializing the variables, invoking the solver, and interpreting the output. These are similar to those described in Section 3, except for the following important differences:

- These routines solve a specialized SDP, not an SQLP. The quadratic and linear parts of the SQLP are assumed to be empty.
- The k–th constraint matrix $(A_S)_k$ is assumed to be $e_k e_k^T$, where e_k is the k–th unit vector (a vector of all zeros, except a 1 at the k–th position), so the matrix A_S *should not* be stored explicitly. The user must provide the

[10] As long as `setpath.m` has been called, Matlab will be able to find routines in the various subdirectories.

[11] Sometimes called the HRVW/KSH/M direction in the literature

cost matrix C_S (stored in the field `C.s`) and the primal constraint right-hand side b, stored in `b`. The fields `C.q` and `C.l` are assumed to be empty, and are ignored. No structure `A` is needed.

- There is a special initialization routine called `dsetopt.m` which sets the options to default values. The default value for `reltol` is larger than that used by `setopt.m`, since the XZ method generally cannot achieve the same high accuracy as the XZ+ZX method. Also the default value for `tau` is 0.99 (instead of 0.999) since the XZ method performs poorly with values of `tau` close to one.
- There is a special initialization routine called `dinit.m` which initializes `X.s`, `y`, `Z.s`. It expects that `C.s` and `b` are available in the Matlab workspace.
- The script `dsdp.m` uses an additional parameter called `useXZ`, which when set to 1 (this is the default set by `dsetopt.m`) solves the problem by the specialized code `fdsdp.m` using the XZ method. When `useXZ` is set to 0, the specialized code is not used, but instead `dsdp.m` calls `fsql.m` to solve the problem using the XZ+ZX method, after constructing the matrix A_S explicitly and setting the structure `A` accordingly. The latter usually provides more accurate solutions, but at substantially increased computation time.
- The problems that fall into this class are graph problems that usually require the graph to be connected, and hence will usually be applied to problems with a single block.[12] They do not use the `validate` parameter, *i.e.* no check is made to ensure block structure compatibility. Nevertheless, a few simple consistency checks are always made on the data.
- The user who wants a function interface can bypass the script `dsdp.m` by typing

$$[\mathtt{X.s}, \mathtt{y}, \mathtt{Z.s}, \mathtt{iter}, \mathtt{compval}, \mathtt{feasval}, \mathtt{objval}, \mathtt{iter}, \mathtt{termflag}] = \mathtt{fdsdp}(\mathtt{C.s}, \mathtt{b}, \mathtt{X.s}, \mathtt{y}, \mathtt{Z.s}, \mathtt{opt})$$

- When the output parameter `termflag` has the value 3, the meaning is slightly different from the XZ+ZX case. Here, `termflag` = 3 means that the Schur complement matrix, which is symmetric and mathematically positive definite for the XZ method, was numerically indefinite or singular, *i.e.* Matlab's `chol` routine failed or generated a zero diagonal element. Also, the termination code `termflag` = 2 is not used.

5.2. **Lovász θ function.** A specialized solver to compute the Lovász θ function of an undirected graph is also available. As in the diagonally constrained case, such an SDP may be solved more efficiently by the XZ method than the XZ+ZX method, as long as the number of edges in the graph is not too large. The driver script is `lsdp.m` and the specialized function is `flsdp.m`. The five steps involved in setting up and solving a problem are again similar to Section 3, except for the following important differences:

- These routines solve a specialized SDP, not an SQLP. The quadratic and linear parts of the SQLP are assumed to be empty.
- The k–th constraint matrix $(A_S)_k$, $k = 1, \ldots, m-1$, is $e_i e_j^T + e_j e_i^T$, where (i, j) is the kth edge in the graph, and $(A_S)_m = I$, with $b = e_m$. These should not be constructed explicitly. The user must instead provide an edge list `G` (a matrix with as many rows as there are edges and 2 columns, with each row of this matrix defining an edge of the graph (listing its two vertices)) and

[12] However, `dsdp` and `fdsdp` will function with multiple blocks.

a weight list `w` (a vector with one component for each vertex of the graph, specifying the weight for that vertex). The cost matrix C_S is defined by $(C_S)_{ij} = -\sqrt{w_i w_j}$ (and is not constructed explicitly). The optimal value of this SDP is actually the negative of the value of the θ function of the graph.
- The routine `lsetopt.m` sets the options in the structure `opt` just as `dsetopt.m` does.
- The initialization routine for the variables is called `linit.m`, which expects the variables `G` and `w` to be available in the Matlab workspace.
- Like `dsdp.m`, the script `lsdp.m` requires the parameter `useXZ` to be available, and calls the specialized solver `flsdp.m` only if `useXZ` equals 1 (the default value set by `lsetopt.m`). Otherwise, it calls `fsql.m` to solve the problem by the XZ+ZX method, after constructing the matrices A_S and C_S and the vector b and setting the structures `A` and `C` and the vector `b` accordingly.
- The problems that fall into this class are graph problems that usually require the graph to be connected. Hence, this specialized solver have been purposefully designed to work with a single block only. If `validate` is set to 1, `flsdp.m` checks to see if the graph is connected and prints a warning if it is not.
- The user who wants a function interface can bypass the script by typing

$$[\texttt{X.s}, \texttt{y}, \texttt{Z.s}, \texttt{iter}, \texttt{compval}, \texttt{feasval}, \texttt{objval}, \texttt{iter}, \texttt{termflag}] = \texttt{flsdp}(\texttt{G}, \texttt{w}, \texttt{X.s}, \texttt{y}, \texttt{Z.s}, \texttt{opt});$$

- When the output parameter `termflag` has the value 3, the meaning is slightly different from the XZ+ZX case. Here, $\texttt{termflag} = 3$ means that the Schur complement matrix, which is symmetric and mathematically positive definite for the XZ method, was numerically indefinite or singular, *i.e.* Matlab's `chol` routine failed or generated a zero diagonal element. Also, the termination code $\texttt{termflag} = 2$ is not used.

Appendix B contains sample Matlab sessions that illustrate the use of `dsdp.m` and `lsdp.m` on these special types of problems.

6. Complementarity and Nondegeneracy

We provide routines to check the strict complementarity and the primal and dual nondegeneracy conditions at the computed solution of an SQLP, and to estimate its condition number. These routines may be found in the `support` subdirectory.

6.1. **Strict complementarity.** In order to define the strict complementarity condition for given (X_S, X_Q, X_L) and (Z_S, Z_Q, Z_L) satisfying the cone inequalities, we must consider the three components separately:

The semidefinite part: The matrices X_S, Z_S satisfy the complementarity condition if $X_S \bullet Z_S = 0$, *i.e.*, the matrix product $X_S Z_S = 0$, and the strict complementarity condition if, in addition, $X_S + Z_S$ is strictly positive definite. Alternatively, let $\lambda_1 \geq \cdots \geq \lambda_n$ denote the eigenvalues of X_S and $\omega_1 \leq \cdots \leq \omega_n$ denote the eigenvalues of Z_S. Then X_S, Z_S satisfy the complementarity condition if, for all i, the product $\lambda_i \omega_i$ is zero, and the strict complementarity condition if, for all i, exactly one of the pair λ_i and ω_i is zero.

The quadratic part: Using the notation (1)–(2), again writing $x = X_Q$ for brevity, and likewise $z = Z_Q$: x, z satisfy the complementarity condition if

$x^T z = 0$, and the strict complementarity condition if, in addition, for each block i, exactly one of the pair

$$x_1^i \qquad \texttt{and} \qquad (z_1^i)^2 - \sum_{j=2}^{n_i} (z_j^i)^2) \tag{7}$$

is zero, and exactly one of the pair

$$z_1^i \qquad \texttt{and} \qquad (x_1^i)^2 - \sum_{j=2}^{n_i} (x_j^i)^2) \tag{8}$$

is zero. Consequently, for each block, either the vectors x^i and z^i are both nonzero boundary points, or one is zero and the other is in the interior of the quadratic cone.

The linear part: X_L, Z_L satisfy the complementarity condition if $X_L^T Z_L$ is zero, and the strict complementarity condition if, for each i, exactly one of the pair $(X_L)_i$ and $(Z_L)_i$ is zero.

Generally, `fsql.m` returns computed solutions approximately satisfying strict complementarity when these exist. The routine which verifies strict complementarity is

`scomp.m`: Verify strict complementarity at a computed solution. The calling sequence is

$$[\texttt{strictc}, \texttt{v}] = \texttt{scomp}(\texttt{X}, \texttt{Z}, \texttt{blk}, \texttt{tol});$$

where `X`, `Z` and `blk` are structures as before, and `tol` is a tolerance This routine first checks the global complementarity condition `comp` $<$ `tol`. If violated, it quits immediately with `strictc` $= -1$. Otherwise, it checks strict complementarity block by block, using the square root of `tol` to determine whether the relevant quantities are sufficiently close to zero. The output vectors `v.s` and `v.q` and scalar `v.l` indicate the result for each block, with a value of 1 if strict complementarity holds in that block, and 0 if it is violated. The summary flag `strictc` is then set to the minimum value of `v.s`, `v.q` and `v.l`.

The SDPpack routines are generally able to solve problems for which strict complementarity fails to hold, but the convergence rate is markedly slower, and generally less accuracy is achieved. Note the use of the square root of the tolerance in `scomp.m`, since if strict complementarity is violated, usually quantities which are mathematically zero are not reduced to very small values. The reason for the square root is as follows. Suppose that an SDP where strict complementarity fails to hold is approximately solved, obtaining computed solutions X_S and Z_S with $X_S \bullet Z_S = \kappa$. This implies that the pairwise computed eigenvalue products satisfy $\lambda_i \omega_i \leq \kappa$. For an index i for which strict complementarity holds, one computed eigenvalue of the pair is typically small (about the same order of magnitude as κ) and the other is not. For an index for which strict complementarity fails, both computed eigenvalues in the pair typically have order of magnitude only $\kappa^{1/2}$, even though both are zero at the true solution.

If `scomp` returns `strictc` $= 0$, check the output parameter `v` to identify which blocks are relevant, and use the routines `blkeig` (for the semidefinite part) and `qcpos` (for the quadratic part) to investigate further (see below). Consider also experimenting with different values for `tol`.

An example of a simple SDP for which strict complementarity fails is given in the sample Matlab session in Appendix B. The `ladder2` Steiner tree example in Appendix C is an example of a QCLP for which strict complementarity fails.

6.2. **Primal and dual nondegeneracy.** Primal and dual nondegeneracy conditions are defined for SDP's in [6]. It is important to realize that these conditions are defined *with respect to the specified block structure* (see [7]), so that they reduce to standard LP conditions in the special case that all blocks are one. The extensions of these concepts to SQLP's will be discussed in [3]. See also the comments in the codes for the definitions. The most important of these can be accessed using Matlab's `help` command. The routines are:

`pcond.m`: Given the constraints of an SQLP, its block structure and a primal feasible point, this routine tests for primal degeneracy. The calling sequence is:

```
[cndp,Dsize,D] = pcond(A,blk,X,tol)
```

where `A`, `blk`, `X` are structures as earlier and `tol` is a tolerance used to decide whether components of `X` are or are not on the boundary of their cones. In exact arithmetic, the routine would return the value `cndp` = `inf` $(+\infty)$ if and only if the problem is primal degenerate at `X`. A large value of `cndp` is a strong indication that the problem is primal degenerate. Type `help pcond` for the definition, as well as an explanation of the output parameters `Dsize` and `D`.

`dcond.m`: Given the constraint matrix of an SQLP, the block structure and a dual feasible point, this routine tests for dual degeneracy. The calling sequence is:

```
[cndd,Bsize,B] = dcond(A,blk,Z,tol)
```

where `A`, `blk`, `X` are structures as earlier and `tol` is a tolerance used to decide whether components of `Z` are or are not on the boundary of their cones. In exact arithmetic, the routine would return the value `cndd` = `inf` $(+\infty)$ if and only if the problem is dual degenerate at `Z`. A large value of `cndd` is a strong indication that the problem is dual degenerate. Type `help dcond` for the definition, as well as an explanation of the output parameters `Bsize` and `B`.

As a rough guide, if `X` (`Z`) passed to `pcond.m` (`dcond.m`) is the solution of an SQLP solved with the default parameter values in `setopt.m`, and if `tol` $= 10^{-6}$, then a value exceeding, say 10^8, for `cndp` (`cndd`) is indicative of primal (dual) degeneracy. Avoid setting `tol` too small, especially if strict complementarity does not hold, in which case try values larger than 10^{-6}.

Primal (dual) nondegeneracy implies the uniqueness of dual (primal) solutions. The converse is true if strict complementarity holds [6].

Before calling `pcond.m` or `dcond.m` for a diagonally constrained SDP or a Lovász θ problem, the user must ensure that `A.s`, `b` and `C.s` have been constructed. This can easily be done by calling the appropriate script (`dsdp.m` or `lsdp.m`) with `useXZ` $= 0$ and `opt.maxit` $= 0$. Upon termination of the script, the variables will be defined in the Matlab workspace.

6.3. **The condition number.** The condition number of an SDP is defined in [7] and its significance is discussed there. Briefly, the condition number of an SQLP is defined as the condition number of the Jacobian of the function to which Newton's method is applied in defining the XZ+ZX search direction, evaluated at the solution. This quantity is $+\infty$ if and only if the solution of the SQLP fails to satisfy the strict complementarity or primal or dual nondegeneracy conditions. The routine is

`sqlcond.m`: Given the data of an SQLP and the solutions `X` and `Z`, this routine verifies the optimality conditions and computes a lower bound (in the 1–norm) of the condition number of an SQLP. The calling sequence[13] is:

```
[cndsql,comp,pinfeas,dinfeas,blkmat] = sqlcond(A,b,C,blk,X,y,Z)
```

In exact arithmetic, the routine would return the value `cndsql` = `inf` ($+\infty$) if and only if strict complementarity or primal or dual nondegeneracy are violated. A large value of `cndsql` is a strong indication of that at least one of these three conditions fails to hold. This routine takes a long time to execute compared to `pcond.m` and `dcond.m`, and can therefore be used only for small problems, but its advantage is that no tolerance is required to determine ranks.

In order to use `sqlcond` on a diagonally constrained SDP or a Lovász θ problem, the user must ensure that `A`, `b` and `C` have been constructed, by first calling the appropriate script (`dsdp.m` or `lsdp.m`) with `useXZ` $= 0$ and `opt.maxit` $= 0$.

7. Support Routines

Here we describe a number of support routines which are available. These routines are in the subdirectory `support` except where noted. More information is available by typing `help routine_name` from within Matlab.

`export.m`, `import.m`: These routines provide SDPpack's interface with ASCII data. The calling sequence for the first of these is

$$\texttt{failed} = \texttt{export}(\texttt{fname}, \texttt{A}, \texttt{b}, \texttt{C}, \texttt{blk})$$

where `fname` is the name of the file (string) to which the data must be exported. The data will be stored in one of the formats described in Appendix A. If the data was successfully exported, `export.m` returns 0, otherwise 1. The file name `fname` must have a period and an extension (other than the standard Matlab extensions `mat`, `mex` etc.) following the period (see Section 3.1). The calling sequence for `import.m` is

$$[\texttt{A}, \texttt{b}, \texttt{C}, \texttt{blk}] = \texttt{import}(\texttt{fname})$$

where `fname` is the name of a file (string) containing an SQLP in one of the two formats described in Appendix A.

`plotcomp.m`, `plotfeas.m`, `plotobj.m`: Plot `comp`; `pinfeas`, `dinfeas`; and the primal and dual objective values, respectively, as a function of the iteration count.

`blkeig.m`: Returns the eigenvalues of a symmetric block diagonal matrix, computing them blockwise. The calling sequence is

$$[\texttt{lam}, \texttt{Q}, \texttt{indef}] = \texttt{blkeig}(\texttt{M}, \texttt{blk}, \texttt{sortflag})$$

where `M` is a block diagonal matrix (not a structure), `blk` is a vector of block sizes, and `sortflag` is an optional input argument. The vector of eigenvalues, `lam`, is sorted blockwise in ascending or descending order depending on whether `sortflag` is 1 or -1. The matrix of eigenvectors, `Q`, has its columns permuted accordingly. The output parameter `indef` is set to 1 if the input

[13] Recall that the user does not need to specify all output parameters.

matrix `X.s` is not positive definite and 0 otherwise. For example, eigenvalue complementarity may be checked by typing

$$[\texttt{blkeig(X.s,blk.s,1)} \quad \texttt{blkeig(Z.s,blk.s,}-\texttt{1)}]$$

where the structures `X` and `Z` contain computed solutions of an SQLP. The routine `blkeig.m` is in the `sdppack` directory since it is used by the main routines.

`qcfirst.m`: Extract the special components $x_1^1, \ldots, x_1^q$ and $z_1^1, \ldots, z_1^q$ from the block vectors `x` and `z` (see (2)). The calling sequence is

$$[\texttt{x0,z0}] = \texttt{qcfirst(x,z,blk)}$$

where `x` and `z` are block vectors with block sizes given by the vector `blk`, and the output vectors `x0` and `z0` have entries consisting of the first entries of each block of `x` and `z` respectively.

`qcpos.m`: Given a block vector `x`, this routine computes the vector of values given by the right-hand component of (8). The calling sequence is

$$[\texttt{dist,v}] = \texttt{qcpos(x,blk)}$$

where `x` is the block vector, `blk` is the vector of block sizes, `v` is the vector on the right-hand side of (8) and `dist` is the minimum entry in `v` (possibly negative or zero, if `x` is outside the quadratic cone or on its boundary. Using this routine together with `qcfirst.m`, one can conveniently check strict complementarity for the quadratic part of an SQLP, explicitly displaying the pairs of vectors in (7) and (8) by

```
[x0,z0] = qcfirst(X.q,Z.q,blk.q);
[distx,vx] = qcpos(X.q,blk.q);
[distz,vz] = qcpos(Z.q,blk.q);
[ x0  vz ]  % check one complementary pair (x=0, z on boundary)
[ z0  vx ]  % check other complementary pair (z=0, x on boundary)
```

This routine is in the `sdppack` directory since it is used by the main routines.

`arw.m`: This routine computes the "arrow matrix" [2] which is used to derive the search direction for the quadratic part. The calling sequence is

$$\texttt{X} = \texttt{arw(x,blk)}$$

This routine is called only by `sqlcond.m`.

`svec.m, smat.m`: These routines convert a symmetric block diagonal matrix into its vector representation and vice versa. See Section 3.1. These routines are in the `sdppack` directory.

`skron.m`: This routine computes the symmetric Kronecker product [1] of two block diagonal matrices. The calling sequence is

$$\texttt{K} = \texttt{skron(M,N,blk)}$$

This routine is called only by `sqlcond.m`.

`preproc.m`: This routine can be used to detect inconsistency of the constraints, or to identify and eliminate redundant constraints. The calling sequence is:

$$[\texttt{Anew,bnew,flag}] = \texttt{preproc(A,b,tol)}$$

where `tol` is a tolerance (*e.g.* 10^{-6}) used to determine the rank of [`A.s A.q A.l`].

`makeA.m`: This routine is used to construct the matrix `A.s`, the semidefinite part of the primal linear constraints. The calling sequence is

$$\mathtt{A.s} = \mathtt{makeA}(\mathtt{blk.s}, \mathtt{Amat})$$

where `blk.s` is as above and `Amat` is a cell array storing the block diagonal matrices $(A_S)_1, \ldots, (A_S)_m$ (see Section 3.1).

The package also provides routines to create a variety of randomly[14] generated test problems. There is also a routine to create SQLP's with solutions having prescribed properties. These routines are discussed below.

`sqlrnd.m`: This script assumes that `blk` and `m` are available in the Matlab environment, and generates a random primal and dual feasible SQLP with block structure specified by `blk` and with `m` primal constraints. The script `init.m` must be called to initialize the variables to default (and generally infeasible) values, before calling `sql.m` or `fsql.m` to solve the problem. This routine is in the main `sdppack` directory.

`drnd.m`: This script assumes that `n` is available in the the Matlab workspace, and randomly generates the vector b and matrix C_S defining a diagonally constrained SDP (see Section 5.1), stored in `b` and `C.s` respectively. The script `dinit.m` must be called to initialize the variables to default (and generally infeasible) values, before calling the specialized routines `dsdp.m` or `fdsdp.m` to solve the diagonally constrained SDP (see Section 5.1). This routine is in the `special` subdirectory.

`lrnd.m`: This script assumes that `n` and `dsty` are available in the workspace, and generates a random graph with n vertices and expected edge density approximately `dsty`, with edge list stored in `G`. The vertices are given random weights, stored in `w`. A warning is printed if the graph is disconnected. The variables `G` and `w` may then be used as input to the routines which compute a Lovász θ function. The script `linit.m` must be called to initialize the variables to default (and generally infeasible) values, before calling the specialized routines `lsdp.m` or `flsdp.m` to solve the SDP which defines the Lovász θ function (see Section 5.2). This routine is in the `special` subdirectory.

`snrnd.m`: This script assumes that integers `M`, `N` and `d` are available in the workspace, and generates a random "sum of norms" problem: the "dual" is

$$\min \sum_{i=1}^{\mathtt{N}} \|z_i\| \quad \text{s.t.} \quad B_i^T y + z_i = c_i, \quad i = 1, \ldots, \mathtt{N}$$

and the "primal" is

$$\max \sum_{i=1}^{\mathtt{N}} c_i^T x_i \quad \text{s.t.} \quad \sum_{i=1}^{\mathtt{N}} B_i x_i = 0, \quad \|x_i\| \leq 1$$

where the vectors x_i and z_i are of length `d` and the vector y is of length `M` (so the matrices B_i have size `M` by `d`). The routine converts this to a quadratically constrained linear program expressed in SQLP format, so it can then be solved by `sql.m`. This provides an interesting class of problems, since the blocks in the resulting block vector Z_Q must always be on the cone boundary at the

---

[14] All these routines generate data uniformly distributed in $[-1, 1]$. They may be modified to use the normal distribution by using `randn` instead of `rand`.

solution, *i.e.* with the right-hand component of (7) always zero. See the comments in the code for further details. This routine is in the `special` subdirectory.

`makesql.m`: This script assumes that integer `m` and structures `blk`, `rx` and `rz` are available in the Matlab workspace, and creates an SQLP having a primal and a dual solution with prescribed "rank" properties. The structure `blk` is as before. The structure `rx` has three fields, `rx.s`, `rx.q` and `rx.l`. The field `rx.s` is a vector specifying the desired rank of each block of the solution X_S. The field `rx.q` is a vector of values, giving a number 0, 1 or 2 for each block of X_Q, specifying, respectively, whether the corresponding block of the solution X_Q is to be zero, a nonzero vector on the boundary of the quadratic cone, or a vector in the interior of the quadratic cone. The scalar `rx.l` specifies the desired number of nonzero components in the solution vector X_L. The structure `rz` with fields `rz.s`, `rz.q` and `rz.l` specifies the corresponding information for the dual solution (Z_S,Z_Q,Z_L). This routine is useful for creating test problems for which the strict complementarity or primal and dual nondegeneracy conditions do not hold. However, the prescribed rank structures must not violate the complementarity conditions. See the comments in the routine for further explanation.

In addition, scripts `sql2sdp` and `sdp2sql` convert SDP's in SQLP format to the SDP format required by SDPpack Version 0.8 and vice versa, and `sql2qc` and `qc2sql` likewise convert QCLP's from SQLP format to a QCLP-only format and vice versa.

8. Software Support

Although SDPpack is provided "as is" without any warranty of software support, the authors welcome your feedback and suggestions about the package via email. *Bug reports and test problems are especially valuable to the authors.* While sending a bug report by email, please be sure to include the version of the code, your Matlab version, the details of your platform and a small example that causes the bug to appear. To facilitate this, a "Bug Report Submission Form" is available from the SDPpack web page. Test problems may be contributed either as `mat` files or as ASCII files in the format described in Appendix A, and may be emailed to the authors. UNIX users are requested to archive multiple files using `tar`, compress using `gzip` and encode using `uuencode`.

News and information about SDPpack will be communicated via the Interior–Point Mailing List (see http://www.mcs.anl.gov/home/otc/InteriorPoint/).

Warning: copying the value of a structure to another structure can, in certain circumstances, lead to incorrect results, because of a bug in Matlab 5.0 which arises when the structure includes a sparse matrix. See the routine `matbug.m` for an example of the bug. The Mathworks has informed us that this bug will be fixed in Matlab 5.1.

9. Work in Progress

A version of SDPpack written in C is in preparation. How much speedup this will gain over the present version is not yet clear, since use of mex files makes Matlab quite efficient.

Other possible algorithmic improvements have been investigated. These include:

- Variations on Mehrotra's predictor-corrector scheme, including higher-order corrections and other formulas for the choice of centering parameter. Typically these lead to a reduction in the number of iterations, but increased CPU time.
- Extension of Gondzio's multiple centrality corrections for LP to SQLP. This led to a reduction in computation time on the largest of the truss problems (`truss8`) discussed in Appendix C.
- A simplified homogeneous self-dual scheme, which in principle assists with detection of infeasibility. However, when used to solve feasible problems we found this scheme gave substantially less accurate solutions *to the original problem* than the present code, and furthermore did not in practice resolve the feasibility issue in cases where feasiblity is in doubt, for example the control LMI's in Appendix C.
- Choosing the starting points by solving two least squares problems, as is common in LP. This has not proved to be a good choice in our experience.

None of these show enough improvements to the present code to justify their inclusion in the package, but this may change in the future.

References

[1] F. Alizadeh, J.-P. Haeberly, and M. L. Overton. Primal-dual interior-point methods for semidefinite programming: convergence rates, stability, and numerical methods. Technical Report 721, NYU Computer Science Dept, 1996. To appear in SIAM Journal on Optimization.

[2] I. Adler and F. Alizadeh. Primal–Dual Interior Point Algorithms for Convex Quadratically Constrained and Semidefinite Optimization Problems. Technical Report RRR-111-95, RUTCOR, Rutgers University, 1995.

[3] F. Alizadeh, J.-P Haeberly, M. V. Nayakankuppam, M. L. Overton, and S. Schmieta. Optimization with linear, quadratic and semidefinite constraints. In preparation, 1997.

[4] C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz. An interior-point method for semidefinite programming. *SIAM Journal on Optimization*, 6:342–361, 1996.

[5] M. Kojima, M. Shida, and S. Hara. Interior-point methods for the monotone linear complementarity problem in symmetric matrices. *SIAM Journal on Optimization*, 6:86–125, 1997.

[6] F. Alizadeh, J.-P. Haeberly, and M. L. Overton. Complementarity and nondegeneracy in semidefinite programming. *Mathematical Programming*, 77:111–128, 1997.

[7] M. V. Nayakkankuppam and M. L. Overton. Conditioning of semidefinite programs. Technical report, NYU Computer Science Dept, March 1997. URL: http://www.cs.nyu.edu/phd_students/madhu/sdp/papers.html.

[8] F. Alizadeh, J.-P. Haeberly, M. V. Nayakkankuppam, and M. L. Overton. SDPpack user's guide, version 0.8 beta. Technical Report 734, NYU Computer Science Dept, March 1997. URL: http://www.cs.nyu.edu/phd_students/madhu/sdppack/sdppack.html.

Appendix A. An efficient storage scheme for SQL

In the User's Guide for SDPpack Version 0.8 Beta[8], an ASCII storage format for SDP was suggested, based on one communicated to us by A. Nemirovskii. Here, we further extend this to represent SQL problems. We essentially have two cases: the matrices are (i) dense, or (ii) sparse, and in each case, the matrix is represented in a different way. In all cases below, we store one number per line in the file (so that the file may be read efficiently using Matlab's `load` command), although the text sometimes shows several entries on the same line for clarity.

Semidefinite part: We need to store the cost matrix `C.s` as well as the constraint matrices $(A_S)_i$, $i = 1, \ldots, m$ corresponding to the rows of the matrix `A.s`. Suppose X is a block diagonal matrix with block structure $[N_1, \ldots, N_s]$.

We respresent the ith diagonal block of this matrix by $X(i)$, and the (j,k) element within the ith block as $X(i)_{jk}$.

Case (i): For the first block, we first record a 0 (denoting that the block is dense), and then the entries in the upper triangular part of the matrix are provided row–wise.

$$\begin{array}{ccc} 0 & & \\ X(1)_{11} & X(1)_{12}\ldots & X(1)_{1N_1} \\ & X(1)_{22}\ldots & X(1)_{2N_1} \\ & & \vdots \\ & \ldots & X(1)_{N_1N_1} \end{array}$$

(Similar segments for the remaining blocks $X(2),\ldots,X(s)$.)

Case (ii): For the first block, we first record a 1 (denoting that the block is sparse), then the number of nonzero entries in the upper triangular part of the block, say n. Then, for $1 \le j \le n$, we record the row number r_j, the column number c_j, and the value of the corresponding nonzero entry $X(1)_{r_jc_j}$.

$$\begin{array}{ccc} 1 & & \\ n & & \\ r_1 & c_1 & X(1)_{r_1c_1} \\ \vdots & \vdots & \vdots \\ r_n & c_n & X(1)_{r_nc_n} \end{array}$$

(Similar segments for the remaining blocks $X(2),\ldots,X(N_s)$.) The row and the column indices are into the block, and *not* into the whole matrix X.

Note: at present, the code does not permit mixing sparse and dense matrix blocks: the blocks must be either all dense or all sparse.

Quadratic part: Here, we need to store the constraint matrix `A.q`, which has m rows and $\sum_{i=1}^{q} n_i$ columns.

Case (i): We first record a 0 if the constraint matrix is dense. Then, the elements of the constraint matrix are given column–wise.

Case (ii): We first record a 1 if the constraint matrix is sparse, then the number of nonzero entries in the matrix. Then, for each nonzero entry, we record the row number, the column number and the value of the corresponding entry.

Linear part: Here, we need to store the constraint matrix `A.l` of size $m \times n_0$.

Case (i): We first record a 0 if the constraint matrix is dense. Then, the elements of the constraint matrix are given column–wise.

Case (ii): We first record a 1 if the constraint matrix is sparse, then the number of nonzero entries in the matrix. Then, for each nonzero entry, we record the row number, the column number and the value of the corresponding entry.

In Table A, we describe the overall storage format for an SQLP; matrices for each of the three parts (semidefinite, quadratic and linear) are stored as just described. The C–style comments are not a part of the storage format.

TABLE 2. ASCII storage format for SQLP's

Line #	Description
/* Common data */	
1	m – the number of constraints
next m lines	b – one entry per line
/* SD data */	
next line	s, the number of SD blocks (0 means no SD component)
next s lines	`blk.s(i)`, $1 \leq i \leq s$ – the sizes of the semidefinite blocks
$\cdots$	the cost matrix `C.s`
$\cdots$	the matrix $(A_S)_1$
$\vdots$	$\vdots$
$\cdots$	the matrix $(A_S)_m$
/* QC data */	
next line	q, the number of QC blocks (0 means no QC component)
next q lines	`blk.q(i)`, $1 \leq i \leq q$ – the sizes of the quadratic blocks
next $\sum_{i=1}^{q}$ `blk.q(i)` lines	the cost vector `C.q`
$\cdots$	the matrix `A.q`
/* LP data */	
next line	`blk.l` (0 means no LP component)
next `blk.l` lines	the cost vector `C.l`
$\cdots$	the matrix `A.l`

APPENDIX B. EXAMPLES

This appendix illustrates the use of the main routines in SDPpack by a sample Matlab session. In the examples below, >> denotes the Matlab prompt. Matlab was invoked from the `sdppack` directory.

```
>> warning off     % eliminate ill conditioning warnings
>>
>> setpath         % set the path
>>
>> format short e
>>
>> clear all
>>
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Examples of setting up a problem using makeA
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> blk.s = [3 3];  % two semidefinite blocks of size 3 each
>> m = 2;          % two primal constraints
>> Amat{1} = speye(6);  % sparse identity matrix
>> Amat{2}(1:3,1:3) = sparse([1 2 3; 2 4 5; 3 5 6]);
>> Amat{2}(4:6,4:6) = sparse([1 0 0; 0 2 2; 0 2 3]);
>> A.s = makeA(Amat, blk.s);  % make the 2 x 12 matrix A.s
>> C.s(1:3,1:3) = sparse(ones(3,3));
```

```
>> C.s(4:6,4:6) = sparse(zeros(3,3));
>> b = [1 2]';
>> setopt;         % set the options
>> init;
>> sql;

tau =   0.9990,     scalefac =      100

iter   p_step      d_step     p_infeas    d_infeas      X . Z       pobj        dobj
  0   0.000e+00   0.000e+00   1.801e+03   2.437e+02   6.000e+04   3.000e+02   0.000e+00
  1   1.000e+00   1.000e+00   7.183e-02   2.220e-16   1.020e+02   3.349e-01  -9.934e+01
  2   1.000e+00   9.916e-01   1.337e-15   2.320e-14   8.935e-01   3.155e-01  -5.781e-01
  3   6.685e-01   1.000e+00   2.701e-15   3.928e-16   2.178e-01   1.279e-02  -2.050e-01
  4   9.699e-01   9.827e-01   3.140e-16   2.629e-16   4.009e-03   7.435e-05  -3.935e-03
  5   1.000e+00   9.992e-01   1.554e-15   1.734e-16   6.112e-06   3.387e-07  -5.773e-06
  6   9.990e-01   9.990e-01   8.882e-16   2.923e-16   6.112e-09   3.387e-10  -5.773e-09
  7   9.990e-01   9.990e-01   2.220e-16   3.351e-16   6.112e-12   3.386e-13  -5.773e-12
fsql: stop since error reduced to desired value

sql: elapsed time               =  0.971    seconds
sql: elapsed cpu time           =  0.300    seconds
sql: number of iterations       =  7
sql: final value of X . Z       =  6.112e-12
sql: final primal infeasibility =  2.220e-16
sql: final dual infeasibility   =  3.351e-16
sql: primal objective value     =  3.3864577808628837e-13
sql: dual objective value       = -5.7731216096433383e-12
>> full(X.s)       % display solution as full matrix

ans =

   9.7585e-02  -5.2413e-02  -4.5172e-02            0            0            0
  -5.2413e-02   1.1485e-01  -6.2439e-02            0            0            0
  -4.5172e-02  -6.2439e-02   1.0761e-01            0            0            0
            0            0            0   1.5025e-01            0            0
            0            0            0            0   2.3968e-01   1.0069e-01
            0            0            0            0   1.0069e-01   2.9002e-01

>> full(Z.s)       % display solution as full matrix

ans =

   1.0000e+00   1.0000e+00   1.0000e+00            0            0            0
   1.0000e+00   1.0000e+00   1.0000e+00            0            0            0
   1.0000e+00   1.0000e+00   1.0000e+00            0            0            0
            0            0            0   4.6022e-12            0            0
            0            0            0            0   5.7731e-12   2.3418e-12
            0            0            0            0   2.3418e-12   6.9440e-12
```

```
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> %  Add a quadratic block to this problem
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> blk.q = 3;
>> A.q = [1 2 3; 4 5 6];
>> C.q = [1 -1 -1]';
>> init;
>> sql;

tau =   0.9990,     scalefac =      100

iter   p_step      d_step     p_infeas    d_infeas      X . Z       pobj        dobj
  0   0.000e+00   0.000e+00   2.211e+03   3.427e+02   7.000e+04   4.000e+02   0.000e+00
  1   1.000e+00   5.838e-01   1.271e-13   1.426e+02   2.041e+04   4.047e+02  -1.802e+01
  2   1.000e+00   9.527e-01   2.970e-11   6.746e+00   1.296e+03   3.857e+02  -1.707e+00
  3   1.000e+00   1.000e+00   1.748e-09   7.581e-16   2.743e+02   2.734e+02  -9.276e-01
  4   9.953e-01   1.000e+00   8.258e-12   2.544e-16   1.279e+00   3.522e-01  -9.264e-01
  5   1.000e+00   9.029e-01   1.512e-15   9.572e-16   2.875e-01   6.095e-02  -2.266e-01
  6   1.000e+00   6.339e-01   2.614e-14   2.435e-16   6.941e-02  -5.025e-02  -1.197e-01
  7   4.536e-01   4.488e-01   3.991e-13   4.347e-16   5.265e-02  -1.841e-02  -7.106e-02
  8   9.470e-01   1.000e+00   2.162e-14   3.220e-16   2.873e-03  -6.976e-02  -7.264e-02
  9   9.990e-01   9.988e-01   1.510e-15   4.280e-16   3.227e-06  -7.053e-02  -7.053e-02
 10   9.990e-01   9.990e-01   9.155e-16   2.922e-16   3.227e-09  -7.053e-02  -7.053e-02
 11   9.990e-01   9.990e-01   8.006e-16   3.907e-16   3.228e-12  -7.053e-02  -7.053e-02
fsql: stop since error reduced to desired value

sql: elapsed time               =  0.679    seconds
sql: elapsed cpu time           =  0.390    seconds
sql: number of iterations       =  11
sql: final value of X . Z       =  3.228e-12
sql: final primal infeasibility =  8.006e-16
sql: final dual infeasibility   =  3.907e-16
sql: primal objective value     = -7.0528980380840739e-02
sql: dual objective value       = -7.0528980384069004e-02
>> [X.q Z.q]       % quadratic part of solution

ans =

   1.7078e-01   1.1404e+00
   1.2339e-01  -8.2400e-01
   1.1806e-01  -7.8841e-01

>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Example of a randomly generated problem
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> blk.s = [50 5 5 5 5 20];    % semidefinite blocks
>> blk.q = [50 100 150];        % quadratic blocks
>> blk.l = 100;                % linear block
```

```
>> m = 100;
>> sqlrnd         % generate random feasible problem
>> setopt         % set default options
>> init           % initialize variables (infeasible)
>> sql            % solve problem

tau =   0.9990,     scalefac =      100

iter   p_step      d_step     p_infeas    d_infeas      X . Z       pobj        dobj
  0   0.000e+00   0.000e+00   5.044e+04   2.099e+03   1.930e+06   3.963e+04   0.000e+00
  1   1.000e+00   6.436e-01   3.510e-11   7.481e+02   5.628e+05   1.025e+04  -1.348e+03
  2   1.000e+00   9.355e-01   3.058e-09   4.822e+01   4.299e+04   8.355e+03   2.328e+01
  3   1.000e+00   9.410e-01   1.069e-08   2.844e+00   7.400e+03   5.799e+03   1.156e+02
  4   7.786e-01   4.205e-01   2.382e-09   1.648e+00   1.956e+03   1.719e+03   1.190e+02
  5   1.000e+00   1.000e+00   3.818e-08   1.221e-13   3.625e+02   4.880e+02   1.256e+02
  6   7.567e-01   1.000e+00   9.289e-09   1.289e-13   8.980e+01   2.174e+02   1.276e+02
  7   7.252e-01   7.134e-01   2.553e-09   1.205e-13   4.403e+01   1.849e+02   1.409e+02
  8   8.715e-01   9.749e-01   3.308e-10   1.288e-13   1.853e+01   1.635e+02   1.449e+02
  9   9.287e-01   9.833e-01   2.486e-11   1.252e-13   3.447e+00   1.491e+02   1.457e+02
 10   9.789e-01   9.678e-01   2.278e-11   1.237e-13   4.773e-01   1.464e+02   1.459e+02
 11   1.000e+00   1.000e+00   1.116e-11   1.248e-13   8.423e-02   1.460e+02   1.459e+02
 12   9.474e-01   9.573e-01   3.018e-12   1.213e-13   4.430e-03   1.459e+02   1.459e+02
 13   1.000e+00   1.000e+00   1.210e-10   1.288e-13   4.988e-04   1.459e+02   1.459e+02
 14   9.980e-01   9.980e-01   1.016e-11   1.289e-13   1.021e-06   1.459e+02   1.459e+02
 15   9.990e-01   9.990e-01   2.651e-11   1.292e-13   1.021e-09   1.459e+02   1.459e+02
 16   9.989e-01   9.989e-01   3.060e-11   1.315e-13   1.118e-12   1.459e+02   1.459e+02
fsql: stop since error reduced to desired value

sql: elapsed time               =  147.092  seconds
sql: elapsed cpu time           =  113.480  seconds
sql: number of iterations       =  16
sql: final value of X . Z       =  1.118e-12
sql: final primal infeasibility =  3.060e-11
sql: final dual infeasibility   =  1.315e-13
sql: primal objective value     =  1.4592215412906933e+02
sql: dual objective value       =  1.4592215412906256e+02
>>
>> % note the successive reductions of X . Z by factors
>> % of 1000 in final iterations (this is because tau = 0.999)
>>
>> pcond(A,blk,X,1.0e-06); % confirms that primal nondegenerate
primal condition number =   3.8717e+00
>>                         % (as expected since randomly generated)
>>
>> dcond(A,blk,Z,1.0e-06); % confirms that dual nondegenerate
dual condition number =   5.693e+00
>>                         % (as expected since randomly generated)
>>
>>
```

```
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Example from AHO paper where no strictly
>> % complementary solution exists
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> clear blk
>> C.s = [0 0 0; 0 0 0; 0 0 1];
>> AA{1} = [1 0 0; 0 0 0; 0 0 0];
>> AA{2} = [0 0 1; 0 1 0; 1 0 0];
>> AA{3} = [0 1 0; 1 0 0; 0 0 1];
>> b = [1 0 0]';
>> blk.s = 3;
>> m = 3;
>> A.s = makeA(AA,blk.s);
>>
>> init
>> sql

tau =   0.9990,     scalefac =      100

iter   p_step      d_step     p_infeas    d_infeas      X . Z       pobj        dobj
  0   0.000e+00   0.000e+00   1.726e+02   1.726e+02   3.000e+04   1.000e+02   0.000e+00
  1   8.615e-01   1.000e+00   2.392e+01   0.000e+00   2.842e+03   5.815e+01  -1.201e+02
  2   9.837e-01   1.000e+00   3.887e-01   0.000e+00   1.521e+02   3.536e+00  -1.090e+02
  3   1.000e+00   1.000e+00   4.965e-16   4.885e-15   9.861e+00   2.574e+00  -7.288e+00
  4   1.000e+00   7.694e-01   1.295e-15   1.249e-15   2.285e+00   1.946e+00  -3.382e-01
  5   8.536e-01   1.000e+00   8.311e-14   9.714e-17   3.873e-01   5.488e-02  -3.324e-01
  6   2.734e-01   9.545e-01   6.046e-14   5.551e-17   5.702e-02   4.038e-02  -1.664e-02
  7   1.000e+00   1.000e+00   1.117e-14   4.510e-17   9.225e-03   4.605e-03  -4.620e-03
  8   9.296e-01   9.296e-01   9.447e-16   1.003e-16   1.538e-03   7.681e-04  -7.700e-04
  9   1.000e+00   1.000e+00   1.221e-13   2.730e-17   1.144e-03   5.718e-04  -5.719e-04
 10   9.439e-01   9.439e-01   6.217e-15   1.730e-18   6.536e-05   3.268e-05  -3.268e-05
 11   1.000e+00   1.000e+00   1.999e-15   6.325e-17   3.232e-05   1.616e-05  -1.616e-05
 12   9.271e-01   9.271e-01   1.735e-18   1.038e-16   2.431e-06   1.216e-06  -1.216e-06
 13   1.000e+00   1.000e+00   2.221e-15   1.563e-17   1.350e-06   6.748e-07  -6.748e-07
 14   9.327e-01   9.327e-01   4.441e-16   2.036e-18   9.324e-08   4.662e-08  -4.662e-08
 15   1.000e+00   1.000e+00   1.110e-15   3.456e-18   5.000e-08   2.500e-08  -2.500e-08
fsql: stop since new point is substantially worse than current iterate
      X . Z =   3.541e-09
      pri_infeas =   4.441e-16
      dual_infeas =   9.704e-17

sql: elapsed time               =  0.272    seconds
sql: elapsed cpu time           =  0.270    seconds
sql: number of iterations       =  15
sql: final value of X . Z       =  5.000e-08
sql: final primal infeasibility =  1.110e-15
sql: final dual infeasibility   =  3.456e-18
sql: primal objective value     =  2.5001652697068844e-08
```

```
sql: dual objective value       = -2.5001652697556712e-08
>>
>> [blkeig(X.s,blk.s,-1)  blkeig(Z.s,blk.s,1)]

ans =

   1.0000e+00   1.5014e-08
   1.9987e-04   9.9936e-05
   1.5014e-08   1.0000e+00

>>
>> % note that convergence is much slower than usual,
>> % and that solution is much less accurate than usual.
>>
>> % confirm that solution is primal NONdegenerate
>> %  (large tolerance since not strictly complementary)
>> pcond(A,blk,X,1.0e-3);
primal condition number =   1.0001e+00
>> % confirm that solution is dual NONdegenerate
>> dcond(A,blk,Z,1.0e-3);
dual condition number =   1.414e+00
>> % condition number is infinite, since SC failed
>> sqlcond(A,b,C,blk,X,y,Z);

sqlcond: comp                              =   5.000e-08
sqlcond: primal infeasibility              =   1.110e-15
sqlcond: dual infeasibility                =   3.456e-18
sqlcond: cond estimate of 3x3 block matrix =   4.124e+04
>>
>>
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Examples of degenerate problems
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> clear blk
>> blk.s = 10;
>> m = 50;           % m chosen to ensure primal degeneracy
>> rx.s = 2;         % choose primal solution rank and
>> rz.s = 7;         % dual solution rank in advance
>> makesql           % generated so solution is primal degenerate

makesql: strict complementarity violated for the SD part
>>
>> init
>> opt.prtlevel = 0;
>> sql

tau =   0.9990,     scalefac =      100
```

```
sql: elapsed time               =  0.460    seconds
sql: elapsed cpu time           =  0.460    seconds
sql: number of iterations       =  6
sql: final value of X . Z       =  2.723e-10
sql: final primal infeasibility =  1.853e-12
sql: final dual infeasibility   =  2.084e-13
sql: primal objective value     =  2.8543802322637415e+00
sql: dual objective value       =  2.8543802321029546e+00
>>
>> [blkeig(X.s,blk.s,-1)  blkeig(Z.s,blk.s,1)]   % sorted eigenvalues

ans =

   1.0687e+00   4.8157e-11
   6.4574e-01   1.3909e-10
   2.8679e-13   7.6364e+01
   2.0865e-13   8.0091e+01
   1.7295e-13   8.4225e+01
   1.5601e-13   9.1698e+01
   1.0169e-13   1.0549e+02
   9.2401e-14   1.1717e+02
   6.1256e-14   1.2753e+02
   4.5897e-14   1.8355e+02

>>
>> % note that convergence took place to a strictly complementary solution
>>
>> pcond(A,blk,X,1.0e-06); % confirms that solution is primal degenerate
primal condition number =          Inf
>>
>> dcond(A,blk,Z,1.0e-06);  % check if dual degenerate
dual condition number =   1.374e+00
>>
>> sqlcond(A,b,C,blk,X,y,Z);    % confirms that SQL condition number is infinite,

sqlcond: comp                              =   2.723e-10
sqlcond: primal infeasibility              =   1.853e-12
sqlcond: dual infeasibility                =   2.084e-13
sqlcond: cond estimate of 3x3 block matrix =   1.017e+19
>>                              % since SQLP is degenerate
>>
>> clear blk
>> blk.s = [5 5 7];       % a bigger problem chosen so that strict
>> blk.q = [10 20 30];    % complementarity does not hold
>> blk.l = 30;
>> rx.s = [2 3 4];
>> rx.q = [0 1 2];
>> rx.l = 20;
```

```
>> rz.s = [2 2 3];
>> rz.q = [1 1 0];
>> rz.l = 5;
>> m = 100;
>> makesql;

makesql: strict complementarity violated for the SD part

makesql: strict complementarity violated for the QC part

makesql: strict complementarity violated for the LP part
>> init;
>> sql;

tau =   0.9990,     scalefac =      100


sql: elapsed time               =  12.593   seconds
sql: elapsed cpu time           =  7.940    seconds
sql: number of iterations       =  18
sql: final value of X . Z       =  1.575e-06
sql: final primal infeasibility =  5.345e-10
sql: final dual infeasibility   =  3.706e-14
sql: primal objective value     =  5.8737338167855953e+01
sql: dual objective value       =  5.8737336592772962e+01
>>              % even the solution of the linear part is not strictly
>> [X.l Z.l]    % complementary

ans =

   1.1058e+00   2.8480e-08
   1.0227e+00   3.0825e-08
   1.9006e+00   1.6573e-08
   1.5969e+00   1.9720e-08
   1.8186e+00   1.7318e-08
   1.6924e+00   1.8608e-08
   1.0905e+00   2.8872e-08
   1.4904e+00   2.1140e-08
   1.8018e+00   1.7481e-08
   1.5935e+00   1.9767e-08
   1.1138e+00   2.8249e-08
   1.4436e+00   2.1809e-08
   1.9663e+00   1.6016e-08
   1.4636e+00   2.1523e-08
   1.4495e+00   2.1735e-08
   1.8269e+00   1.7240e-08
   1.2986e+00   2.4245e-08
   1.7099e+00   1.8415e-08
   1.6780e+00   1.8766e-08
```

```
   1.2583e+00   2.5036e-08
   4.3929e-04   1.0943e-04
   4.7011e-04   1.0190e-04
   4.2388e-04   1.1309e-04
   2.3974e-04   1.9792e-04
   3.1820e-04   1.5071e-04
   2.7166e-08   1.1595e+00
   1.7050e-08   1.8479e+00
   1.9730e-08   1.5970e+00
   1.6520e-08   1.9068e+00
   1.9520e-08   1.6144e+00

>>
>>
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % A diagonally constrained SDP
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> n = 50;        % n = m = 50
>> drnd           % random C and b, but A_k = e_k e_k^T
>> dsetopt        % set useXZ = 1, tau = .99
>> scalefac = 1;  % X0 = Z0 = I fine for random problems
>> dinit
>> dsdp           % since useXZ = 1, special-purpose XZ code used

dsdp: using XZ method...


tau =   0.9900,     scalefac =       1

iter   p_step      d_step     p_infeas    d_infeas     X . Z        pobj        dobj
  0   0.000e+00   0.000e+00   4.313e+00   2.964e+01   5.000e+01   4.187e+00   0.000e+00
  1   5.346e-02   1.000e+00   4.083e+00   6.880e-15   7.776e+02  -4.254e+01  -3.767e+02
  2   9.345e-01   8.959e-01   2.672e-01   7.692e-15   1.110e+02  -7.191e+01  -1.632e+02
  3   4.907e-01   1.000e+00   1.361e-01   4.615e-15   5.926e+01  -1.020e+02  -1.519e+02
  4   9.152e-01   1.000e+00   1.154e-02   4.529e-15   9.287e+00  -1.363e+02  -1.448e+02
  5   9.631e-01   9.604e-01   4.264e-04   2.220e-15   4.821e-01  -1.427e+02  -1.431e+02
  6   9.646e-01   9.555e-01   1.507e-05   1.884e-15   3.960e-02  -1.430e+02  -1.431e+02
  7   9.151e-01   1.000e+00   1.279e-06   1.601e-15   3.484e-03  -1.431e+02  -1.431e+02
  8   8.620e-01   1.000e+00   1.766e-07   2.738e-15   7.022e-04  -1.431e+02  -1.431e+02
  9   9.724e-01   1.000e+00   4.884e-09   2.391e-15   3.878e-05  -1.431e+02  -1.431e+02
 10   9.501e-01   9.894e-01   6.289e-10   1.332e-15   2.054e-06  -1.431e+02  -1.431e+02
Stop since new point is substantially worse than current iterate
      X . Z =   2.569e-07
      pri_infeas =   2.708e-08
      dual_infeas =   2.878e-15

dsdp: elapsed time              =  5.556    seconds
dsdp: elapsed cpu time          =  3.080    seconds
```

```
dsdp: number of iterations       =  10
dsdp: final value of X . Z       =  2.054e-06
dsdp: final primal infeasibility =  6.289e-10
dsdp: final dual infeasibility   =  1.332e-15
dsdp: primal objective value     = -1.4306128549766009e+02
dsdp: dual objective value       = -1.4306128752873551e+02
>>
>> setopt         % sets useXZ = 0, tau = .999
>> dinit
>> dsdp           % since useXZ = 0, general-purpose XZ+ZX code used

dsdp: using XZ+ZX method...


tau =   0.9990,     scalefac =      100

iter   p_step      d_step     p_infeas    d_infeas      X . Z       pobj        dobj
  0   0.000e+00   0.000e+00   7.040e+02   7.071e+02   5.000e+05   4.187e+02   0.000e+00
  1   9.981e-01   1.000e+00   1.307e+00   0.000e+00   3.115e+03  -8.649e+00  -2.207e+03
  2   9.933e-01   9.944e-01   8.804e-03   3.231e-14   1.965e+02  -1.401e+01  -2.093e+02
  3   4.877e-01   1.000e+00   4.511e-03   1.025e-14   1.101e+02  -7.106e+01  -1.806e+02
  4   9.685e-01   1.000e+00   1.423e-04   1.502e-14   3.892e+01  -1.258e+02  -1.647e+02
  5   7.764e-01   8.993e-01   3.181e-05   1.693e-14   6.357e+00  -1.379e+02  -1.443e+02
  6   1.000e+00   1.000e+00   1.265e-14   3.716e-15   1.619e+00  -1.416e+02  -1.432e+02
  7   9.110e-01   9.678e-01   2.576e-15   2.083e-15   1.482e-01  -1.429e+02  -1.431e+02
  8   1.000e+00   1.000e+00   2.718e-13   3.233e-15   5.226e-02  -1.430e+02  -1.431e+02
  9   9.987e-01   9.987e-01   3.088e-15   3.580e-15   6.569e-05  -1.431e+02  -1.431e+02
 10   9.990e-01   9.990e-01   1.170e-14   1.831e-15   6.569e-08  -1.431e+02  -1.431e+02
 11   9.990e-01   9.990e-01   1.175e-14   2.512e-15   6.612e-11  -1.431e+02  -1.431e+02
fsql: stop since error reduced to desired value

dsdp: elapsed time               =  16.932   seconds
dsdp: elapsed cpu time           =  15.000   seconds
dsdp: number of iterations       =  11
dsdp: final value of X . Z       =  6.612e-11
dsdp: final primal infeasibility =  1.175e-14
dsdp: final dual infeasibility   =  2.512e-15
dsdp: primal objective value     = -1.4306128745773665e+02
dsdp: dual objective value       = -1.4306128745780248e+02
>>
>> % notice that specialized XZ method is faster but less
>> % accurate than XZ+ZX method
>>
>>
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % A Lovasz problem
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> %
>> n = 30;           % number of vertices
```

```
>> dsty = 0.2;       % edge density is 20%
>> lrnd              % random graph with random weights
>>
>> lsetopt           % set useXZ = 1, tau = .99
>> scalefac = 1;     % X0 = Z0 = I fine for random problems
>> opt.validate = 1; % check connectivity
>> linit
>> lsdp              % since useXZ = 1, special-purpose XZ code used

lsdp: using XZ method...


tau =   0.9900,     scalefac =       1

iter   p_step      d_step     p_infeas    d_infeas     X . Z        pobj        dobj
  0   0.000e+00   0.000e+00   2.900e+01   1.617e+01   3.000e+01  -1.424e+01   0.000e+00
  1   5.544e-02   7.601e-02   2.739e+01   1.494e+01   3.420e+01  -9.122e+01  -5.246e-01
  2   2.382e-02   1.000e+00   2.674e+01   8.168e-15   2.980e+02  -8.604e+01  -1.384e+01
  3   9.867e-01   1.000e+00   3.548e-01   2.516e-15   1.276e+01  -5.056e+00  -1.315e+01
  4   9.788e-01   8.449e-01   7.535e-03   4.503e-15   1.862e+00  -4.861e+00  -6.673e+00
  5   2.080e-01   6.145e-01   5.968e-03   2.924e-15   1.565e+00  -4.994e+00  -6.521e+00
  6   4.847e-02   7.077e-01   5.678e-03   1.201e-15   1.434e+00  -5.024e+00  -6.421e+00
  7   5.645e-02   3.238e-01   5.358e-03   3.013e-15   1.337e+00  -5.044e+00  -6.347e+00
  8   1.674e-02   7.984e-02   5.268e-03   1.868e-15   1.293e+00  -5.041e+00  -6.302e+00
  9   6.391e-02   5.669e-01   4.932e-03   1.996e-15   1.237e+00  -5.073e+00  -6.279e+00
 10   9.836e-03   6.618e-02   4.883e-03   2.670e-15   1.198e+00  -5.070e+00  -6.238e+00
 11   4.878e-02   8.159e-01   4.645e-03   2.802e-15   1.224e+00  -5.094e+00  -6.289e+00
 12   1.757e-02   2.141e-01   4.563e-03   2.521e-15   1.159e+00  -5.095e+00  -6.225e+00
 13   2.718e-01   1.000e+00   3.323e-03   2.609e-15   1.034e+00  -5.245e+00  -6.258e+00
 14   1.000e+00   1.000e+00   4.589e-16   2.286e-15   3.246e-01  -5.835e+00  -6.160e+00
 15   9.318e-01   7.887e-01   4.712e-16   2.619e-15   3.868e-02  -6.053e+00  -6.091e+00
 16   7.006e-01   1.000e+00   5.176e-15   2.532e-15   1.423e-02  -6.078e+00  -6.092e+00
 17   9.823e-01   9.886e-01   3.423e-16   2.084e-15   3.803e-04  -6.089e+00  -6.089e+00
 18   9.881e-01   9.920e-01   1.443e-15   3.047e-15   4.407e-06  -6.089e+00  -6.089e+00
Stop since new point is substantially worse than current iterate
      X . Z =   8.274e-08
      pri_infeas =   4.519e-14
      dual_infeas =   2.341e-15

lsdp: elapsed time               =  14.388   seconds
lsdp: elapsed cpu time           =  7.270    seconds
lsdp: number of iterations       =  18
lsdp: final value of X . Z       =  4.407e-06
lsdp: final primal infeasibility =  1.443e-15
lsdp: final dual infeasibility   =  3.047e-15
lsdp: primal objective value     = -6.0894182157068650e+00
lsdp: dual objective value       = -6.0894226229375326e+00
lsdp: Lovasz theta function value =  6.0894204193221988e+00
>>
```

```
>> setopt           % sets useXZ = 0, tau = .999
>> linit
>> lsdp             % since useXZ = 0, general-purpose code XZ+ZX used

lsdp: using XZ+ZX method...


tau =   0.9990,     scalefac =      100

iter   p_step      d_step     p_infeas    d_infeas      X . Z       pobj        dobj
  0   0.000e+00   0.000e+00   2.999e+03   5.505e+02   3.000e+05  -1.424e+03   0.000e+00
  1   9.976e-01   1.000e+00   7.055e+00   4.041e-16   7.914e+02  -1.781e+01  -1.005e+02
  2   1.000e+00   1.000e+00   1.003e-15   8.658e-16   7.540e+01  -2.330e+00  -7.773e+01
  3   1.000e+00   9.348e-01   2.322e-16   2.228e-14   4.923e+00  -2.646e+00  -7.568e+00
  4   7.859e-01   9.626e-01   6.440e-16   4.593e-15   1.175e+00  -5.323e+00  -6.499e+00
  5   1.000e+00   1.000e+00   5.525e-15   3.058e-15   3.543e-01  -5.832e+00  -6.187e+00
  6   8.798e-01   8.434e-01   8.686e-16   3.681e-15   5.603e-02  -6.036e+00  -6.092e+00
  7   1.000e+00   1.000e+00   6.409e-15   3.610e-15   5.198e-03  -6.084e+00  -6.090e+00
  8   9.980e-01   9.984e-01   8.893e-16   3.470e-15   1.023e-05  -6.089e+00  -6.089e+00
  9   9.990e-01   9.990e-01   1.905e-17   4.081e-15   1.023e-08  -6.089e+00  -6.089e+00
 10   9.990e-01   9.990e-01   2.126e-17   3.592e-15   1.024e-11  -6.089e+00  -6.089e+00
fsql: stop since error reduced to desired value

lsdp: elapsed time                = 8.923    seconds
lsdp: elapsed cpu time            = 6.890    seconds
lsdp: number of iterations        = 10
lsdp: final value of X . Z        = 1.024e-11
lsdp: final primal infeasibility  = 2.126e-17
lsdp: final dual infeasibility    = 3.592e-15
lsdp: primal objective value      = -6.0894223218290779e+00
lsdp: dual objective value        = -6.0894223218393178e+00
lsdp: Lovasz theta function value =  6.0894223218341974e+00
>>
>> % notice that specialized XZ method is faster
>> % but less accurate than XZ+ZX method
>>
>> % since useXZ = 0, A.s was formed; so we can now call pcond
>> % and dcond
>>
>> % for random weights, usually Lovasz SDP is degenerate
>> %
>> rank(X.s,1.0e-06)        % for random weights, usually rank(X.s) is 1

ans =

     1

>> rank(Z.s,1.0e-06)        % for random weights, usually rank(Z.s) is n-1
```

```
ans =

    29

>> pcond(A,blk,X,1.0e-06);  % usually primal degenerate
primal condition number =          Inf
>> dcond(A,blk,Z,1.0e-06);  % usually dual nondegenerate
dual condition number =   1.000e+00
>>
>> w = ones(size(w));  % change weights to all one
>> lsetopt             % set useXZ = 1, tau = .99
>> opt.prtlevel = 0;   % turn off detailed output
>> opt.validate = 1;
>> linit
>> lsdp                % since useXZ = 1, special-purpose XZ code used

lsdp: using XZ method...


tau =   0.9900,     scalefac =      100


lsdp: elapsed time                =  7.271    seconds
lsdp: elapsed cpu time            =  4.890    seconds
lsdp: number of iterations        =  12
lsdp: final value of X . Z        =  1.481e-06
lsdp: final primal infeasibility  =  4.993e-09
lsdp: final dual infeasibility    =  4.188e-15
lsdp: primal objective value      = -1.2059530468726049e+01
lsdp: dual objective value        = -1.2059532001323507e+01
lsdp: Lovasz theta function value =  1.2059531235024778e+01
>>
>> setopt              % sets useXZ = 0, tau = .999
>> opt.prtlevel = 0;
>> linit
>> lsdp                % since useXZ = 0, general-purpose XZ+ZX code used

lsdp: using XZ+ZX method...


tau =   0.9990,     scalefac =      100


lsdp: elapsed time                =  8.710    seconds
lsdp: elapsed cpu time            =  7.740    seconds
lsdp: number of iterations        =  12
lsdp: final value of X . Z        =  6.871e-10
lsdp: final primal infeasibility  =  9.010e-14
lsdp: final dual infeasibility    =  6.881e-15
```

```
lsdp: primal objective value      = -1.2059531781985829e+01
lsdp: dual objective value        = -1.2059531782671732e+01
lsdp: Lovasz theta function value =  1.2059531782328779e+01
>>
>> % notice that specialized XZ method is faster but less
>> % accurate than XZ+ZX method
>>
>> % since useXZ = 0, the matrix A.s was formed, so we can now call
>> % pcond and dcond
>>
>> % for weights all one, usually Lovasz SDP is not degenerate
>>
>> rank(X.s,1.0e-06)  % for weights all one, usually rank(X.s) > 1

ans =

     9

>> rank(Z.s,1.0e-06)  % for weights all one, usually rank(Z.s) < n-1

ans =

    21

>> pcond(A,blk,X,1.0e-06);  % sometimes primal nondegenerate
primal condition number =   1.4494e+01
>> dcond(A,blk,Z,1.0e-06);  % sometimes dual degenerate
dual condition number =   4.092e+01
>>
>>
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Sample truss problem
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> load testdata/truss/truss1     % from Nemirovskii
>> setopt                        % sets scalefac = 100
>> init
>> sql

tau =   0.9990,     scalefac =      100

iter   p_step      d_step     p_infeas    d_infeas      X . Z       pobj        dobj
  0   0.000e+00   0.000e+00   7.803e+02   3.603e+02   1.300e+05   1.000e+02   0.000e+00
  1   1.000e+00   7.344e-01   6.356e-14   9.570e+01   1.391e+04   2.565e+02  -6.643e+01
  2   1.000e+00   1.000e+00   7.105e-13   4.600e-16   2.470e+02   2.471e+02   1.014e-01
  3   6.313e-01   1.000e+00   2.558e-13   7.850e-17   9.229e+01   9.189e+01  -4.013e-01
  4   8.211e-01   1.000e+00   5.357e-14   6.206e-17   1.668e+01   1.745e+01   7.700e-01
  5   5.064e-03   4.190e-01   5.353e-14   9.930e-16   1.004e+01   1.746e+01   7.424e+00
  6   1.000e+00   9.086e-01   2.365e-13   1.662e-15   1.254e+00   1.006e+01   8.809e+00
```

```
  7   9.886e-01   9.977e-01   7.944e-14   2.267e-15   1.919e-02   9.018e+00   8.999e+00
  8   9.990e-01   9.990e-01   8.889e-14   8.951e-16   1.948e-05   9.000e+00   9.000e+00
  9   9.990e-01   9.990e-01   2.337e-14   1.332e-15   1.948e-08   9.000e+00   9.000e+00
 10   9.990e-01   9.990e-01   2.368e-14   1.986e-15   1.949e-11   9.000e+00   9.000e+00
fsql: stop since error reduced to desired value

sql: elapsed time               =  0.994    seconds
sql: elapsed cpu time           =  0.420    seconds
sql: number of iterations       =  10
sql: final value of X . Z       =  1.949e-11
sql: final primal infeasibility =  2.368e-14
sql: final dual infeasibility   =  1.986e-15
sql: primal objective value     =  8.9999963153049691e+00
sql: dual objective value       =  8.9999963152853759e+00
>> pcond(A,blk,X,1.0e-06);       % check if primal degenerate
primal condition number =          Inf
>> dcond(A,blk,Z,1.0e-06);       % check if dual degenerate
dual condition number =   8.920e+00
>>
>>
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Sample LMI problem
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> load testdata/lmi/hinf1        % from Gahinet
>> setopt                        % sets scalefac = 100
>> scalefac = 1000;              % better choice for these
>> init
>> sql

tau =   0.9990,     scalefac =     1000

iter   p_step      d_step      p_infeas    d_infeas      X . Z       pobj        dobj
  0   0.000e+00   0.000e+00   4.542e+03   3.742e+03   1.400e+07   0.000e+00   0.000e+00
  1   8.700e-01   1.000e+00   5.906e+02   6.355e-14   1.458e+06  -2.306e+00  -1.431e+03
  2   9.987e-01   1.000e+00   7.417e-01   4.032e-13   3.250e+03  -1.106e-02  -1.425e+03
  3   9.159e-01   1.000e+00   6.241e-02   1.995e-13   3.426e+02  -1.203e-02  -2.534e+02
  4   8.825e-01   9.935e-01   7.333e-03   3.108e-13   1.302e+01  -3.657e-02  -5.348e+00
  5   9.912e-01   7.585e-01   6.439e-05   1.352e-13   1.309e+00  -1.132e+00  -2.373e+00
  6   4.743e-01   1.000e+00   3.385e-05   5.754e-13   6.506e-01  -1.555e+00  -2.126e+00
  7   9.626e-01   1.000e+00   1.265e-06   5.312e-13   2.088e-02  -2.020e+00  -2.038e+00
  8   9.135e-01   9.854e-01   1.095e-07   5.714e-13   1.360e-03  -2.032e+00  -2.033e+00
  9   6.290e-01   1.000e+00   5.246e-07   8.049e-13   1.131e-03  -2.032e+00  -2.033e+00
 10   8.693e-01   9.966e-01   6.712e-08   1.668e-12   1.019e-04  -2.033e+00  -2.033e+00
 11   9.975e-01   1.000e+00   2.433e-07   1.531e-12   5.992e-07  -2.033e+00  -2.033e+00
 12   9.990e-01   9.990e-01   6.823e-08   1.073e-12   6.020e-10  -2.033e+00  -2.033e+00
fsql: stop since limiting accuracy reached
 (smallest eigenvalue of Z.s =  -1.741e-12)
```

```
sql: elapsed time               =  1.003    seconds
sql: elapsed cpu time           =  0.710    seconds
sql: number of iterations       =  12
sql: final value of X . Z       =  6.020e-10
sql: final primal infeasibility =  6.823e-08
sql: final dual infeasibility   =  1.073e-12
sql: primal objective value     = -2.0326845732912178e+00
sql: dual objective value       = -2.0326421368134850e+00
>> pcond(A,blk,X,1.0e-06);       % check if primal degenerate
primal condition number =   5.7041e+13
>> dcond(A,blk,Z,1.0e-06);       % check if dual degenerate
dual condition number =   9.738e+00
>>
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % A Steiner tree example (minimizing a sum of 2-norms)
>> % A special QCLP, written in SQLP format
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> load testdata/steiner/ladder1   % Chung-Graham ladder
>> %
>> %  Only blk.q is nonempty
>> blk

blk =

    s: 0
    l: 0
    q: [49x1 double]

>> setopt
>> init
>> sql

tau =   0.9990,     scalefac =      100

iter   p_step      d_step     p_infeas    d_infeas      X . Z       pobj        dobj
  0   0.000e+00   0.000e+00   6.930e+02   7.009e+02   4.900e+05   0.000e+00   0.000e+00
  1   1.000e+00   1.000e+00   6.646e-15   1.079e-16   4.802e+03  -2.430e-01  -4.803e+03
  2   1.000e+00   9.927e-01   9.293e-16   2.579e-15   3.487e+01  -4.910e-01  -3.536e+01
  3   8.302e-01   9.503e-01   2.513e-15   2.369e-15   4.777e+00  -2.048e+01  -2.526e+01
  4   9.119e-01   9.978e-01   4.454e-14   2.709e-15   5.564e-01  -2.318e+01  -2.373e+01
  5   1.000e+00   1.000e+00   1.522e-13   4.469e-15   1.071e-01  -2.348e+01  -2.359e+01
  6   8.876e-01   8.595e-01   4.918e-14   2.957e-15   1.467e-02  -2.353e+01  -2.354e+01
  7   1.000e+00   1.000e+00   5.491e-12   3.257e-15   2.940e-03  -2.353e+01  -2.354e+01
  8   9.953e-01   9.983e-01   9.722e-14   3.450e-15   9.223e-06  -2.353e+01  -2.353e+01
  9   9.990e-01   9.990e-01   7.714e-13   2.302e-15   9.223e-09  -2.353e+01  -2.353e+01
 10   9.990e-01   9.990e-01   6.046e-13   2.578e-15   9.247e-12  -2.353e+01  -2.353e+01
fsql: stop since error reduced to desired value
```

```
sql: elapsed time               =  18.389   seconds
sql: elapsed cpu time           =  12.580   seconds
sql: number of iterations       =  10
sql: final value of X . Z       =  9.247e-12
sql: final primal infeasibility =  6.046e-13
sql: final dual infeasibility   =  2.578e-15
sql: primal objective value     = -2.3534397500876146e+01
sql: dual objective value       = -2.3534397500886332e+01
>> %  verify that solution is strictly complementary
>> scomp(X,Z,blk,1e-5)

ans =

     1

>>
>> load testdata/steiner/ladder2   % a different ladder
>> %
>> %
>> %  Only blk.q is nonempty
>> blk

blk =

    s: 0
    l: 0
    q: [49x1 double]

>> setopt
>> init
>> sql

tau =   0.9990,     scalefac =      100

iter   p_step      d_step     p_infeas    d_infeas      X . Z       pobj        dobj
  0   0.000e+00   0.000e+00   6.930e+02   7.009e+02   4.900e+05   0.000e+00   0.000e+00
  1   1.000e+00   1.000e+00   6.646e-15   9.310e-17   4.802e+03  -2.427e-01  -4.803e+03
  2   1.000e+00   9.927e-01   8.237e-16   1.864e-15   3.490e+01  -4.902e-01  -3.539e+01
  3   8.279e-01   9.385e-01   2.288e-15   3.655e-15   4.956e+00  -2.040e+01  -2.536e+01
  4   9.305e-01   9.819e-01   1.014e-13   3.410e-15   4.794e-01  -2.320e+01  -2.368e+01
  5   1.000e+00   1.000e+00   2.687e-13   4.627e-15   8.107e-02  -2.343e+01  -2.351e+01
  6   8.911e-01   9.001e-01   7.657e-14   3.897e-15   8.711e-03  -2.346e+01  -2.347e+01
  7   9.966e-01   9.955e-01   1.462e-12   3.440e-15   1.773e-03  -2.347e+01  -2.347e+01
  8   1.000e+00   1.000e+00   1.543e-11   3.314e-15   4.066e-04  -2.347e+01  -2.347e+01
  9   8.499e-01   8.496e-01   2.404e-12   4.447e-15   6.396e-05  -2.347e+01  -2.347e+01
 10   1.000e+00   1.000e+00   4.716e-11   4.000e-15   1.527e-05  -2.347e+01  -2.347e+01
 11   8.631e-01   8.630e-01   7.198e-12   3.821e-15   2.177e-06  -2.347e+01  -2.347e+01
fsql: stop since new point is substantially worse than current iterate
      X . Z =   5.038e-07
```

```
      pri_infeas =   2.882e-10
      dual_infeas =   2.794e-15

sql: elapsed time               =  19.491   seconds
sql: elapsed cpu time           =  14.100   seconds
sql: number of iterations       =  11
sql: final value of X . Z       =  2.177e-06
sql: final primal infeasibility =  7.198e-12
sql: final dual infeasibility   =  3.821e-15
sql: primal objective value     = -2.3467622601298221e+01
sql: dual objective value       = -2.3467624778208947e+01
>> %  in this case, solution is NOT strictly complementary
>> [s,v] = scomp(X,Z,blk,1e-5)

s =

     0


v =

    s: []
    q: [1x49 double]
    l: []

>> [x0,z0] = qcfirst(X.q,Z.q,blk.q);
>> [distx,vx] = qcpos(X.q,blk.q);
>> [ z0  vx ]  % for some indices, z=0 and x is on boundary

ans =

   7.5904e-01   5.8516e-08
   3.7407e-01   1.1873e-07
   1.5181e-01   2.9256e-07
   6.0723e-01   7.3145e-08
   3.0362e-01   1.4629e-07
   4.5542e-01   9.7526e-08
   4.5542e-01   9.7527e-08
   3.0362e-01   1.4628e-07
   6.0723e-01   7.3146e-08
   1.5181e-01   2.9254e-07
   7.5904e-01   5.8517e-08
   1.4145e-07   3.1433e-01
   2.7576e-04   7.1523e-04
   5.7725e-01   5.3462e-08
   5.7725e-01   5.3463e-08
   2.7421e-04   7.1246e-04
   4.3819e-07   1.0158e-01
   6.3623e-01   6.9813e-08
```

```
   4.4721e-01   9.9313e-08
   1.8901e-01   2.3500e-07
   2.5820e-01   1.7203e-07
   2.5820e-01   1.7201e-07
   1.8901e-01   2.3497e-07
   4.4721e-01   9.9318e-08
   5.1640e-01   8.6011e-08
   6.3623e-01   6.9810e-08
   7.1836e-01   6.1830e-08
   3.7407e-01   1.1874e-07
   7.1836e-01   6.1829e-08
   3.7407e-01   1.1874e-07
   7.1836e-01   6.1829e-08
   3.7407e-01   1.1874e-07
   7.1836e-01   6.1829e-08
   3.7407e-01   1.1874e-07
   7.1836e-01   6.1830e-08
   3.7407e-01   1.1873e-07
   9.9986e-01   2.1123e-10
   5.7718e-01   5.3475e-08
   4.2292e-01   8.1495e-08
   5.7718e-01   5.3475e-08
   9.9986e-01   8.5694e-10
   5.1640e-01   8.6011e-08
   5.1640e-01   8.6009e-08
   5.1640e-01   8.6011e-08
   7.7460e-01   5.7339e-08
   6.3623e-01   6.9811e-08
   7.7460e-01   5.7341e-08
   5.1640e-01   8.6011e-08
   5.1640e-01   8.6009e-08

>> diary off
```

Appendix C. Benchmarks

This appendix provides benchmarks of SDPpack Version 0.9 BETA on some randomly generated test problems, a set of 16 LMI problems[15] from control applications (some of which apparently are infeasible), and a set of 8 problems[16] from truss topology design. The control and truss design problems are available as `mat` files from the SDPpack home page.

All problems were solved using `sql.m`. The random problems and the truss design problems used the default option values seet by `setopt.m`, i.e.:

- `opt.prtlevel` $= 1$
- `opt.validate` $= 0$
- `opt.maxit` $= 100$
- `opt.tau` $= 0.999$

---

[15]Provided to us by P. Gahinet.
[16]Provided to us by A. Nemirovskii.

- `scalefac` $= 100.0$
- `autorestart` $= 1$
- `opt.reltol` $= 10^{-11}$
- `opt.abstol` $= 10^{-8}$
- `opt.steptol` $= 10^{-8}$
- `opt.gapprogtol` $= 100.0$
- `opt.feasprogtol` $= 5.0$
- `opt.bndtol` $= 10^{8}$

The control LMI problems were solved with the same parameter settings above, but with `scalefac` $= 1000.0$. The benchmarks were conducted on a dual-processor Sparc Ultra-2 with 192 MB of main memory.

Table 3 shows a set of random problems with semidefinite blocks only. Each problem has three semidefinite blocks of equal size. The number n shown in the table is the sum of the block sizes. Thus, for example, the largest problem shown has three blocks each of size 100.

Table 4 shows a set of random problems with quadratic blocks only. Each problem has three quadratic blocks of equal size. The number n shown in the table is the sum of the block sizes. Thus, for example, the largest problem shown has three blocks each of size 250.

Tables 5 and 6 show the truss and control LMI results respectively. When the symbol $*$ appears next to the value of `termflag`, this indicates that a restart was necessary.

TABLE 3. Randomly generated with semidefinite blocks only

P	n	m	`iter`	`pinfeas`	`dinfeas`	$X \bullet Z$	$C \bullet X$	$b^T y$	CPU secs	`termflag`
1	60	60	14	4.27e-13	3.52e-14	8.86e-14	8.95e+00	8.95e+00	1.88e+01	0
2	120	120	13	2.72e-13	1.26e-13	2.43e-12	3.21e+01	3.21e+01	1.57e+02	0
3	180	180	15	5.10e-12	3.28e-13	2.60e-11	6.31e+01	6.31e+01	8.45e+02	0
4	240	240	13	2.75e-12	5.51e-13	1.54e-12	5.78e+01	5.78e+01	2.13e+03	0
5	300	300	14	1.43e-10	8.14e-13	9.74e-07	5.54e+01	5.54e+01	5.72e+03	4

TABLE 4. Randomly generated with quadratic blocks only

P	n	m	`iter`	`pinfeas`	`dinfeas`	$X \bullet Z$	$C \bullet X$	$b^T y$	CPU secs	`termflag`
1	150	50	8	4.01e-14	9.67e-15	1.25e-10	2.52e+01	2.52e+01	5.40e-01	0
2	300	100	8	2.15e-13	2.80e-14	9.35e-13	7.66e+00	7.66e+00	1.29e+00	0
3	450	150	8	5.13e-13	3.39e-13	1.54e-10	9.20e+01	9.20e+01	3.48e+00	0
4	600	200	8	1.34e-12	9.08e-14	1.57e-11	1.38e+01	1.38e+01	7.00e+00	0
5	750	250	8	8.32e-13	9.55e-13	3.34e-10	8.25e+01	8.25e+01	1.28e+01	0

Table 5. Benchmarks on problems from truss topology design

P	n	m	iter	pinfeas	dinfeas	$X \bullet Z$	$C \bullet X$	$b^T y$	CPU secs	termflag
1	13	6	10	2.37e-14	1.99e-15	1.95e-11	9.00e+00	9.00e+00	4.20e-01	0
2	133	58	12	1.71e-13	1.95e-14	2.31e-11	1.23e+02	1.23e+02	4.81e+00	0
3	31	27	15	5.99e-14	1.59e-15	1.16e-08	9.11e+00	9.11e+00	1.86e+00	4
4	19	12	11	1.06e-13	6.33e-16	3.03e-11	9.01e+00	9.01e+00	6.10e-01	0
5	331	208	15	1.33e-10	2.09e-14	2.45e-09	1.33e+02	1.33e+02	7.28e+01	0
6	451	172	28	2.10e-07	3.18e-13	2.44e-07	9.01e+02	9.01e+02	5.85e+01	4
7	301	86	29	1.78e-06	4.38e-13	9.71e-08	9.00e+02	9.00e+02	2.40e+01	4
8	628	496	18	8.97e-10	2.89e-14	2.16e-11	1.33e+02	1.33e+02	1.13e+03	0

Table 6. Benchmarks on LMI problems from control applications

P	n	m	iter	pinfeas	dinfeas	$X \bullet Z$	$C \bullet X$	$b^T y$	CPU secs	termflag
1	14	13	12	6.82e-08	1.07e-12	6.02e-10	-2.03e+00	-2.03e+00	7.60e-01	2
2	16	13	14	7.81e-09	1.12e-11	4.98e-08	-1.10e+01	-1.10e+01	9.20e-01	4
3	16	13	16	1.28e-05	4.70e-12	1.24e-10	-5.70e+01	-5.69e+01	9.90e-01	1
4	16	13	15	1.18e-06	2.94e-10	7.46e-11	-2.75e+02	-2.75e+02	9.10e-01	2
5	16	13	17	5.60e-04	4.41e-12	4.31e-09	-3.63e+02	-3.63e+02	1.04e+00	1
6	16	13	33	4.11e-02	4.66e-11	6.71e-08	-4.49e+02	-4.49e+02	2.00e+00	2
7	16	13	11	3.84e-05	2.93e-11	1.49e-07	-3.91e+02	-3.91e+02	6.90e-01	1
8	16	13	15	2.56e-05	1.15e-11	8.38e-09	-1.16e+02	-1.16e+02	9.10e-01	1
9	16	13	17	3.44e-07	9.64e-13	2.00e-11	-2.36e+02	-2.36e+02	1.05e+00	2
10	18	21	27	1.42e-05	2.84e-07	1.99e-05	-1.09e+02	-1.09e+02	2.30e+00	-1
11	22	31	23	9.24e-07	4.38e-07	5.66e-04	-6.60e+01	-6.59e+01	3.23e+00	-1
12	24	43	25	4.53e-08	1.04e-07	5.38e-01	-2.00e-01	-1.78e-01	5.31e+00	-1
13	30	57	29	2.10e-04	1.24e-09	1.76e-02	-4.44e+01	-4.44e+01	8.73e+00	2
14	34	73	30	3.74e-07	3.59e-09	4.73e-03	-1.30e+01	-1.30e+01	1.39e+01	5
15	37	91	18	7.79e-06	2.17e-08	3.33e-02	-2.39e+01	-2.40e+01	2.23e+01	-1*
37	16	13	17	3.44e-07	9.64e-13	2.00e-11	-2.36e+02	-2.36e+02	1.08e+00	2

RUTCOR, Rutgers University, New Brunswick, NJ.
E-mail address: `alizadeh@rutcor.rutgers.edu`

Department of Mathematics, Fordham University, Bronx, NY.
E-mail address: `haeberly@murray.fordham.edu`

Courant Institute of Mathematical Sciences, New York University, NY.
E-mail address: `madhu@cs.nyu.edu`

Courant Institute of Mathematical Sciences, New York University, NY.
E-mail address: `overton@cs.nyu.edu`

RUTCOR, Rutgers University, New Brunswick, NJ.
E-mail address: `schmieta@rutcor.rutgers.edu`