SDPPACK
USER'S GUIDE

VERSION 0.8 BETA

F. ALIZADEH, J.–P. HAEBERLY, M. V. NAYAKKANKUPPAM, AND M. L. OVERTON

ABSTRACT. This report describes SDPpack, a package of Matlab files designed to solve semidefinite programs (SDP). SDP is a generalization of linear programming to the space of block diagonal, symmetric, positive semidefinite matrices. The main routine implements a primal–dual Mehrotra predictor–corrector scheme based on the XZ+ZX search direction. We also provide certain specialized routines, one to solve SDP's with only diagonal constraints, and one to compute the Lovász θ function of a graph, using the XZ search direction. Routines are also provided to determine whether an SDP is primal or dual degenerate, and to compute the condition number of an SDP. The code optionally uses 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/phd_students/madhu/sdppack/sdppack.html
NYU Computer Science Dept Technical Report 734, March 1997.

Contents

1. Introduction

Given a positive integer vector $[n_1, \dots, n_p]$, with $n = \sum_{i=1}^{p} n_i$, let $\mathbb{B}$ denote the space of all real, symmetric, $n \times n$ block diagonal matrices whose i^{th} diagonal block is of size n_i. The inner product on this space is

$$A \bullet B = \operatorname{tr} AB = \sum_{i,j} A_{ij} B_{ij}.$$

By $X \succeq 0$, where $X \in \mathbb{B}$, we mean that X is positive semidefinite, i.e. all its diagonal blocks are positive semidefinite. Consider the semidefinite program (SDP)

$$\begin{aligned} P: \quad & \min \quad C \bullet X \\ & \text{s.t.} \quad A_k \bullet X = b_k, \quad k = 1, \dots, m; \qquad X \succeq 0 \end{aligned}$$

where C and A_k, $k = 1, \dots, m$ are all fixed matrices in $\mathbb{B}$, and the unknown variable X also lies in $\mathbb{B}$. The dual program is

$$\begin{aligned} D: \quad & \max \quad b^T y \\ & \text{s.t.} \quad Z + \sum_{k=1}^{m} y_k A_k = C; \qquad Z \succeq 0 \end{aligned}$$

where the dual slack matrix Z also lies in $\mathbb{B}$. In the special case $n_i = 1$, $i = 1, \ldots, p$, the SDP reduces to a linear program. It is assumed that the matrices A_k, $k = 1, \ldots, m$, are linearly independent.

We shall use the notation

$$\begin{aligned} \texttt{pinfeas} &= \left(\sum_{k=1}^{m} (A_k \bullet X - b_k)^2 \right)^{1/2} \\ \texttt{dinfeas} &= \left\| C - Z - \sum_{k=1}^{m} y_k A_k \right\|_F \\ \texttt{dgap} &= X \bullet Z \end{aligned}$$

where $\|\cdot\|_F$ denotes the Frobenius matrix norm. Assuming a Slater condition, i.e. the existence of a strictly feasible primal or dual point of SDP, it is well known that the optimality conditions of SDP may be expressed by the equations $\texttt{pinfeas} = 0$, $\texttt{dinfeas} = 0$, and $\texttt{dgap} = 0$ (together with the semidefinite conditions $X \succeq 0$ and $Z \succeq 0$).

2. Obtaining and Installing SDPpack

The current release of SDPpack is Version 0.8 beta, and is designed to work with Matlab[1] Version 4.2c.1. Users who have already upgraded to Matlab 5.0 can also use the current release of SDPpack. However, several optimizations specific to Matlab 5.0's new features are not available in the current release of SDPpack; these will appear in the next release. The current release of the package can be obtained from the **SDPpack home page** on the World–Wide Web:

`http://www.cs.nyu.edu/phd_students/madhu/sdppack/sdppack.html`

This URL 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.

Once you have obtained SDPpack, use the following instructions to install the package. In what follows, % denotes the shell prompt.

UNIX Platforms:

% gunzip sdppack–v0.8.tar.gz
% tar –xovf sdppack–v0.8.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 `testing` (containing testing routines that create random SDP's), `mex4` (containing C sources to generate MEX files for Matlab 4 — UNIX version only), `mex5` (containing C sources to generate MEX

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

files for Matlab 5 — UNIX platforms only), `convert` (containing routines which can convert data from the format used by SDPpack to those of some other popular codes available now) and `doc` (containing this document). Although this itself is a working installation of the software, the package works much faster with compiled MEX files (binaries). There are three MEX files: `svec.mex***`, `smat.mex***` and `evsumdiv.mex***`, where `***` is a string that depends on your architecture. Compiled MEX files are already available for several platforms (SunOS–4/Sparc, SunOS–5/Sparc, AIX/RS6000, IRIX–5.3/R4000, IRIX-6.2/R8000, Windows NT/95) from the SDPpack home page (see this page on the web for an up to date list of available binaries). The table below summarizes the status of available binaries. Download these files in the `sdppack` directory, so that they reside along

TABLE 1. MEX files available for SDPpack Version 0.8

Operating System	Architecture	Matlab 4.2c.1	Matlab 5.0
IRIX–6.2	R8000/R10000	Yes	Yes
IRIX–5.3	R3000/R4000	Yes	Yes
SunOS–4.4.1	Sparc	Yes	Available soon
SunOS–5.5.1	Sparc	Yes	Available soon
AIX	RS6000	Yes	Available soon
Windows NT/95	PC x86	No	Yes
MacOS (System 7)	Macintosh	No	Available soon

with the corresponding `m`–files: `svec.m`, `smat.m` and `evsumdiv.m`. If you cannot find compiled MEX files for your platform and for the Matlab version you use, then you will need to compile them yourself. On UNIX platforms, Matlab 4.0 users must use the C sources in `mex4`, while Matlab 5.0 users must use the C sources in `mex5`. To compile the MEX files, do the following:

```
% cmex -O2 svec.c        (Matlab 5.0 users should replace cmex by mex)
% cmex -O2 smat.c
% cmex -O2 evsumdiv.c
```

The MEX files thus created must be moved to the `sdppack` directory, so they reside with the corresponding `m` files. Depending on the C compiler you use, the switches in the command line for `cmex` could vary; consult the manual for your C compiler. The following sections describe how to use the package, giving an overview of the main routines in the package. You may wish to add the `testing` subdirectory to the Matlab search path, if you plan on using those routines (see Section 5). More information about any specific routine can be obtained by typing `help routine_name` from within Matlab. Appendix A describes an ASCII storage format for SDP'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 sdp.m and the function fsdp.m

The Matlab routines `sdp.m` and `fsdp.m` solve block diagonal SDP's using a primal–dual Mehrotra predictor–corrector scheme based on the XZ+ZX search direction [1], or AHO direction as it has been referenced in the literature[2]. The simplest option for the user is to call the script `sdp.m`, which automatically calls the Matlab function `fsdp.m`. Additional scripts are provided to help the user set up the data, define necessary parameters, and initialize the variables (as described shortly). The user who requires a function interface should bypass `sdp.m` and call `fsdp.m` directly. In either case there are five steps to be followed:

- set up the data defining the SDP (the use of `makeA.m` or `import.m` simplifies this process)
- initialize the required parameters (the routine `setpars.m` sets all parameters to their default values)
- provide initial values for the variables X, y, and Z (the routine `initvars.m` provides default settings)
- call either the script `sdp.m` or the function `fsdp.m` to solve the SDP
- interpret the output

We now describe each of these steps in detail.

3.1. Preparing the data.

The SDP is defined by the following data:

`A`: a matrix with m rows and q columns, where

$$q = \sum_{i=1}^{p} \frac{n_i(n_i+1)}{2}.$$

The kth row of `A` holds the symmetric block diagonal matrix A_k stored as a vector of length q

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

`C`: the block diagonal matrix C defining the primal objective function

`blk`: a vector whose length is the number p of blocks and whose entries are the block sizes n_i, $1 \le i \le p$

The user has a choice of four ways to set up the data. In all cases, if a matrix has more than one block, it *must* be stored in sparse format.[3] If it has only a single block, then it can be stored either in sparse or in full format, at the user's discretion. However, in the case of a single block, it is recommended that the initial `X` and `Z` be provided as full matrices, as the solutions will most likely be full, even if the data itself is sparse.

1. Constructing `A` directly using the function `svec.m`, which converts block diagonal matrices in $\mathbb{B}$ to column vectors of length q. The function `smat.m` restores the symmetric matrix from such a vector. These routines are invoked by

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

By default, the matrix passed to `svec.m` is assumed to have dense blocks. If the user wishes to take advantage of the sparsity in the blocks of this matrix,

[2]Preliminary versions of this software have been distributed privately for some time. The current version offers improved efficiency and stability and supersedes any prior release.

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

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 $v =$ `svec(M,blk)` and $w =$ `svec(N,blk)` for matrices $M, N \in \mathbb{B}$, then

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

The user must also construct `b`, `C` and `blk`.

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

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

   It requires that the block diagonal matrices $A_1, \ldots, A_m$ be stored in variables named `A1,A2,A3,...`. Thus if $m = 100$, the matrix A_{100} must be stored in a variable `A100`.[4] The user must also construct `b`, `C` and `blk`.

   When `makeA.m` is called by the user, the 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 parameter `sparseblks` may be passed to the routine `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

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

   The data must be stored in a special compact format[5] that is described in Appendix A. 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 (LMI problems from control theory and problems from truss topology design) used in Appendix C, for which `mat` files are available from the SDPpack home page.

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

`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 before calling the solver results in just data validation alone.

`tau`: (Default value = 0.999)
The fraction of the step to the boundary (of the positive semidefinite cone) taken by the algorithm. This choice leads to fast convergence and is generally

[4] Arrays of matrices are not permitted in Matlab 4. They will be utilized in a Matlab 5 version.
[5] This format is based on one provided to us by A. Nemirovskii.

reliable, but may occasionally lead to failures due to short steps (see below). In many cases, the quantity `dgap` $=$ `X` $\bullet$ `Z` is reduced by approximately a factor of $1/(1-\texttt{tau})$ per iteration in the last few iterations.

`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 the infeasibility or `dgap` is large, it is recommended to try restarting with either a reduced value of `tau` or with `X` and `Z` set to larger multiples of the identity. This is done automatically when the driver script `sdp.m` is used, but is *not* done if the the driver script is bypassed with a direct call to the function `fsdp.m`.

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

$$\texttt{pinfeas} + \texttt{dinfeas} + \texttt{dgap} < \texttt{abstol}.$$

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

$$\texttt{pinfeas} + \texttt{dinfeas} + \texttt{dgap} < \texttt{reltol} \times (\|X\|_F + \|Z\|_F)\,.$$

`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.

`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 `dgap` is less than the previous value of `dgap` divided by `gapprogtol`, then the progress is considered "sufficient". This check is performed only when `dgap` has been reduced below `abstol`.

`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 `dgap` 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 parameters attempt to achieve a judicious balance between feasibility and complementarity by trading the former in return for the latter.

`bndtol`: (Default value $= 10^{8}$)
Tolerance on the norm of the solution; if $\|X\|_F$ or $\|Z\|_F$ becomes greater than `bndtol`, the algorithm terminates. Unbounded primal (dual) feasible iterates suggest that the dual (primal) program may be infeasible.

`prtlevel`: (Default value $= 1$)
Determines print level; setting this to 0 produces no output from `fsdp.m`, except warnings that Matlab 4 insists on displaying that ill–conditioned systems

are being solved.[6] This is normal near the solution. Setting `prtlevel` to 1 produces one line of output per iteration (iteration number, primal and dual step lengths, primal and dual infeasibilities, $X \bullet Z$, primal and dual objective values). Upon termination, summary information is provided by the script `sdp.m` regardless of the value of `prtlevel`.

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

3.3. **Initializing the variables.** `X`, `y` and `Z` store the variables X, y and Z. They must be initialized with suitable starting values. Both `X` and `Z` must be strictly positive definite. Typically `y` is set to zero and `X` and `Z` to multiples of the identity matrix. If the block diagonal matrices `X` and `Z` have more than one block, they must necessarily 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 script `initvars.m` may be used to initialize `y` to zero and `X`, `Z` to `scalefac` times the identity matrix, where `scalefac` is a scalar set by the user. The proper choice of `scalefac` is highly problem dependent. The default value is 100, but it may be necessary to set `scalefac` to a larger value. On the other hand, `scalefac` = 1 is often satisfactory, and results in a smaller number of iterations.

3.4. **Invoking `sdp.m` or `fsdp.m`.** After preparing the data, initializing the variables and setting the parameters, the user may simply type

```
sdp
```

to solve the SDP. Alternatively, the user who requires a function interface should type

```
[X,y,Z,iter,gapval,feasval,objval,iter,termflag] = ...
  fsdp(A,b,C,blk,X,y,Z,maxit,tau,steptol,abstol,reltol,...
    gapprogtol,feasprogtol,bndtol,prtlevel,validate);
```

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

`X, y, Z`: The final values of the variables X, y and Z. If `X` and `Z` have multiple blocks, they are stored using Matlab's sparse format. If they have only one block, then they are always full. `X` and `Z` are numerically positive semidefinite in the sense that Matlab's `chol` function does not encounter negative pivots when applied to them.

`iter`: The number of iterations taken by the algorithm.

`gapval`: A vector of length `iter` + 1, with entries equal to the value of `dgap` ($X \bullet Z$) as a function of the iteration count. The first entry in the `gapval` array is the value of `dgap` corresponding to the initial point provided.

`objval`: A matrix with two columns and `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.

[6] These messages can be suppressed in Matlab 5.0 by typing `warning off`.

feasval: A matrix with two columns and $\mathtt{iter}+1$ rows, whose entries are the values of **pinfeas** and **dinfeas** in the first and the second columns respectively, as a function of the iteration count.

termflag: This is the termination flag returned by `fsdp.m`, with the following meaning:

termflag $= 0$**:** Successful termination: both the absolute and relative tolerances were satisfied.

termflag $= 1$**:** The new primal iterate `X` is numerically indefinite or singular, i.e. `chol(X)` encountered a negative or zero pivot. This would not occur in exact arithmetic. The algorithm terminates, returning the current iterate `X` (which is positive semidefinite). This is normal, and usually means the problem is essentially solved but the termination criteria were too stringent. (If `iter` $= 0$, this means that the initial `X` provided was not positive semidefinite. This is easily remedied by adding a positive multiple of the identity to `X`.)

termflag $= 2$**:** The new dual iterate `Z` is numerically indefinite or singular, i.e. `chol(Z)` encountered a negative or zero pivot. This would not occur in exact arithmetic. The algorithm terminates, returning the current iterate `Z` (which is positive semidefinite). This termination flag is also used if `chol` finds that the new iterate `Z` is numerically positive semidefinite, but the routine `blkeig` finds that it has a zero (or negative) eigenvalue, in which case the algorithm returns the new iterate, and then terminates.[7] Both cases are normal, and they usually mean that the problem is essentially solved but that the termination criteria were too stringent. (If `iter` $= 0$, this means that the initial `Z` provided was not positive semidefinite. This is easily remedied by adding a positive multiple of the identity to `Z`.)

termflag $= 3$**:** Termination occurred because the Schur complement was numerically singular, i.e. the Matlab routine `lu` generates a zero pivot, making the XZ+ZX direction undefined. The most likely explanation is that the matrix `A` was rank deficient. The routine `preproc.m` can help in detecting inconsistent constraints and in the elimination of redundant constraints (see Section 5). 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 parameters `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 the infeasibility or `dgap` is large, it is recommended to try restarting with either a reduced value of `tau` or with `X` and `Z` set to larger multiples of the identity, or both. This is done automatically when the driver script `sdp.m` is used, but is the user's responsibility when the driver script is bypassed by a direct call to the function `fsdp.m`. (If `iter` $= 0$, then the initial guesses `X` and `Z` were most

[7] It is possible that `blkeig` returns nonnegative eigenvalues, while the Matlab built–in function `eig` itself does not, or vice–versa. This has to do with the fact that `blkeig` computes eigenvalues one block at a time, and hence is usually slightly more accurate.

likely too close to the boundary of the positive semidefinite cone. Adding a positive multiple of the identity to `X` and/or `Z` will rectify this.)

`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 `fsdp.m` was called with an incorrect number of arguments. If `validate` = 1, this could additionally mean that the initial `X` or `Z` did not conform to the specified block structure.

`termflag` = −2: Termination occurred because $\|X\|_F$ exceeded `bndtol`, indicating possible dual infeasibility.

`termflag` = −1: Termination because $\|Z\|_F$ exceeds `bndtol`, indicating possible primal infeasibility.

We encourage the reader to consult Appendix B which contains a sample Matlab session illustrating the use of the `sdp` script.

4. Specialized Routines

Specialized routines are available for two problem classes: (i) problems with diagonally constrained variables, and (ii) the Lovász θ function of a graph.

4.1. Diagonally constrained problems.

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 [2, 3] (sometimes called the KSH/HRVW/M direction in the literature.) 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 parameters, initializing the variables, invoking the solver, and interpreting the output. These are almost identical to the description in Section 3, except for the following important differences:

- The k–th constraint matrix A_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` does not have to be stored explicitly. The user must provide the cost matrix `C` and the primal constraint right-hand side `b`.
- There is a special initialization routine called `dsetpars.m` which sets the parameters to default values. The default value for `reltol` is larger than that used by `setpars.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 `dinitvars.m` which initializes the variables. It expects that the variables `C` 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 `dsetpars.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 `fsdp.m` to solve the

problem using the XZ+ZX method, after constructing the matrix `A` explicitly. 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. Hence, these specialized solvers have been purposefully designed to work with a single block only. Consequently, they do not use the `validate` parameter, but automatically make a few simple consistency checks on the data.
- The user who wants a function interface can bypass the script `dsdp.m` by typing:

$$[\mathtt{X}, \mathtt{y}, \mathtt{Z}, \mathtt{iter}, \mathtt{gapval}, \mathtt{feasval}, \mathtt{objval}, \mathtt{iter}, \mathtt{termflag}] = \ldots$$
$$\mathtt{fdsdp}(\mathtt{C}, \mathtt{b}, \mathtt{X}, \mathtt{y}, \mathtt{Z}, \mathtt{maxit}, \mathtt{tau}, \mathtt{steptol}, \mathtt{abstol}, \mathtt{reltol}, \ldots$$
$$\mathtt{gapprogtol}, \mathtt{feasprogtol}, \mathtt{bndtol}, \mathtt{prtlevel});$$

- 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 for the XZ method, was numerically indefinite or singular, *i.e.* Matlab's `chol` routine failed or generated a zero diagonal element.

4.2. **Lovász θ function.** A specialized solver to compute the Lovász θ function of a graph is also available. As in the diagonally constrained case, such an SDP is solved much more efficiently by the XZ method than the XZ+ZX method. 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:

- The k–th constraint matrix A_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 $A_m = I$, with $b = e_m$. The user must provide a matrix `G` (an adjacency list with as many rows as there are edges and 2 columns, each row of this matrix defining an edge) and a weight vector `w`, with one component for each vertex of the graph. The cost matrix C is defined by $C_{ij} = -\sqrt{w_i w_j}$. (The optimal value of this SDP is actually minus the value of the θ function of the graph.)
- The routine `lsetpars.m` sets parameters just as `dsetpars.m` does.
- The initialization routine for the variables is called `linitvars.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 `lsetpars.m`). Otherwise, it calls `fsdp.m` to solve the problem by the XZ+ZX method, after constructing the matrices `A`, `b` and `C`.
- The problems that fall into this class are graph problems that usually require the graph to be connected. Hence, these specialized solvers 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:

$$[\mathtt{X},\mathtt{y},\mathtt{Z},\mathtt{iter},\mathtt{gapval},\mathtt{feasval},\mathtt{objval},\mathtt{iter},\mathtt{termflag}] = \ldots$$
$$\mathtt{flsdp}(\mathtt{G},\mathtt{w},\mathtt{X},\mathtt{y},\mathtt{Z},\mathtt{maxit},\mathtt{tau},\mathtt{steptol},\mathtt{abstol},\mathtt{reltol},\ldots$$
$$\mathtt{gapprogtol},\mathtt{feasprogtol},\mathtt{bndtol},\mathtt{prtlevel},\mathtt{validate});$$

- 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 for the XZ method, was numerically indefinite or singular, *i.e.* Matlab's `chol` routine failed or generated a zero diagonal element.

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

5. Support Routines

SDPpack's interface to ASCII data is provided via the two routines `export.m` and `import.m`. The calling sequence for `export.m` is

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

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 `A` is sparse, then the format in Table 3 is used, otherwise that in Table 2 is used. If the data was successfully exported, `export.m` returns 0, otherwise 1. The calling sequence for `import.m` is

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

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

The three routines `plotgap.m`, `plotfeas.m` and `plotobj.m` plot the gap, the primal and dual infeasibilities, and the primal and dual objectives respectively, as a function of the iteration count. Several other miscellaneous features are available via the auxiliary routines described below. More information is available by typing `help routine_name` from within Matlab.

`blkeig.m`: This routine computes the eigenvalues of a symmetric block diagonal matrix, by computing the eigenvalues blockwise. The calling sequence is

```
lam = blkeig(X,blk)
```

For example, (strict) complementarity is easily checked by typing

$$[\mathtt{sort}(\mathtt{blkeig}(\mathtt{X},\mathtt{blk})) \qquad -\mathtt{sort}(\mathtt{blkeig}(-\mathtt{Z},\mathtt{blk}))]$$

where `X` and `Z` are the computed solutions of an SDP. (The sorting operation does not preserve the block structure.)

`primalcond.m`: Given the constraint matrix of an SDP (`A`), the block structure (`blk`) and a primal feasible point `X`, this routine can be used to test for primal degeneracy. The calling sequence is:

```
[cndprimal,D] = primalcond(A,blk,X,eigtol)
```

where `eigtol` is a tolerance used in computing the rank of `X`. A large value of `cndprimal` is a strong indication that the problem is primal degenerate [4] (type `help primalcond` for the definition).

`dualcond.m`: Given the constraint matrix of an SDP (`A`), the block structure (`blk`) and a dual feasible `Z`, this routine tests for dual degeneracy. The calling sequence is:

```
[cnddual, B] = dualcond(A,blk,Z,eigtol)
```

where `eigtol` is a tolerance used in computing the rank of `Z`. A large value of `cnddual` is a strong indication that the problem is dual degenerate [4] (type `help dualcond` for the definition).

If `X` (`Z`) passed to `primalcond.m` (`dualcond.m`) is the solution of an SDP solved with the default parameter values in `setpars.m`, and if `eigtol` $= 10^{-06}$, then a value exceeding, say 10^{10}, for `cndprimal` (`cnddual`) is indicative of primal (dual) degeneracy. Primal (dual) nondegeneracy implies the uniqueness of dual (primal) solutions. The converse is true if strict complementarity holds [4].

`sdpcond.m`: Given the data of an SDP 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 SDP [5]. The calling sequence is:

```
[cndjac,dgap,pinfeas,dinfeas,blockmat] = sdpcond(A,b,C,blk,X,y,Z)
```

A large value of `cndjac` is a strong indication of degeneracy (primal, dual or both), or that the solution violates strict complementarity. This routine takes a long time to execute compared to `primalcond.m` and `dualcond.m`, but the advantage over `primalcond.m` and `dualcond.m` is that no tolerance is required.

To use these routines to examine the degeneracy or conditioning properties of a Lovász θ function problem or of a diagonally constrainted SDP, the user must ensure that the data `A`, `b` and `C` have been constructed. This can easily be done by calling the appropriate script (`lsdp.m` or `dsdp.m`) with `useXZ` $= 0$ and `maxit` $= 0$. Upon termination of the script, these variables will be defined in the Matlab workspace.

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

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

```
K = skron(M,N,blk)
```

This routine is called only by `sdpcond.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:

```
[Anew,bnew,flag] = preproc(A,b,rkthresh)
```

where `rkthresh` is a small threshold (*e.g.* 10^{-6}) used to determine the rank of `A`.

`makeA.m`: This script assumes that there are variables `m`, `blk` and `A1` through `A`m available in the Matlab workspace, and creates the constraint matrix `A` (see Section 3.1).

The package also provides routines to create random test problems, *i.e.* block diagonal SDP's, diagonally constrained problems, Lovász θ problems. There is also a routine to create SDP's with solutions of prescribed rank. These routines are discussed below.

rndinf.m: This script assumes that `blk` and `m` are available in the Matlab environment, and generates a random, block diagonal primal and dual feasible SDP with block structure `blk` and `m` primal constraints. The script `initvars.m` must be called to initialize the variables, which are generally not feasible.

diagcstr.m: This script assumes that `n` is available in the the Matlab workspace, and generates a random diagonally constrained problem. The k–th constraint matrix A_k is assumed to be $e_k e_k^T$, where e_k is the k–th unit vector (a vector with all zeros, except a 1 in the k–th position). The routine generates `b` and `C` randomly. This problem can be solved with the specialized routines `dsdp.m` and `fdsdp.m` (see Section 4), which do not require `A` to be explicitly stored.

thetarnd.m: This script assumes that `n` and `dsty` are available in the Matlab workspace, and sets up an SDP to compute the Lovász θ function of a random graph with n vertices and expected edge density approximately `dsty`. The vertices are given random weights. A warning is printed if the graph is disconnected. The specialized routine `lsdp` can be used to solve this problem (see Section 4).

makesdp.m: This script assumes that `m`, `blk`, `r` and `s` are available in the Matlab workspace, and creates an SDP with a primal solution `X` having rank `r`, and a dual solution `Z` having rank `s`. This routine does not have the provision to handle multiple blocks, so here, `blk` is just a single number. This is particularly useful for creating degenerate test problems, or problems with a non–strictly complementary solution.

nosfeas.m: This script assumes that the block structure vector (`blk`) and the number of constraints (`m`) are available in the Matlab workspace, and creates an SDP which has *no strictly* feasible primal solution.

In addition to these routines, the `convert` subdirectory contains scripts that will convert SDP data to a format recognized by some other popular codes. In particular, there are scripts `to_sp`, `to_lmitlbx` and `to_sdpa` which convert SDP data in the Matlab workspace to formats recognized by SP [6], Matlab's LMI Toolbox, and SDPA [7] respectively. Typing `help routine_name` within Matlab provides some more information on the correspondence between SDPpack variables and those used by the other codes. This is merely to encourage users to try other codes on the benchmark problems in Appendix C (available from the SDPpack web page). These routines may not be supported in future releases.

6. Software Support and Future Work

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 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.

The next release of SDPpack, which will take advantage of several special features of Matlab 5.0, is due for release soon. A fast C version based on LAPACK has already been written, and will be available soon. News and information about

SDPpack (including new releases) will be communicated via the Interior–Point Mailing List (see http://www.mcs.anl.gov/home/otc/InteriorPoint/).

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. In revision for SIAM Journal on Optimization, 1997.

[2] 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.

[3] 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.

[4] F. Alizadeh, J.-P. Haeberly, and M. L. Overton. Complementarity and nondegeneracy in semidefinite programming. *Mathematical Programming*, 1997. To appear.

[5] M. V. Nayakkankuppam and M. L. Overton. Conditioning of semidefinite programs. Technical report, Courant Institute of Mathematical Sciences, New York University, March 1997. URL: http://www.cs.nyu.edu/phd_students/madhu/sdp/papers.html.

[6] S. Boyd and L. Vandenberghe. *SP: Software for semidefinite programming (User's Guide)*, beta version edition, November 1994. URL: http://www-isl.stanford.edu/ boyd/SP.html.

[7] K. Fujisawa, M. Kojima, and K. Nakata. *SDPA: Semidefinite programming algorithm*. Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, revised edition, August 1996.

Appendix A. An efficient storage scheme for SDP

The following scheme for storing block diagonal SDP's in ASCII format is based one communicated to us by A. Nemirovskii. We essentially have two cases: (i) the blocks are dense, and (ii) the blocks are sparse. The main difference between these two cases is the way in which block diagonal matrices are represented. For (i), the entries in the upper triangular part of the matrix are provided row–wise, whereas for (ii), we record the the number of nonzero entries for each block, and the row and column numbers of each entry in this block. In either case, we store one number per line.

Table 2. ASCII format for SDP's with dense blocks

Line #	Description
1	m – the number of constraints
2	l – the number of blocks
3	1 – denotes dense blocks
4 to $p+3$	$blk(i), 1 \leq i \leq p$ – the sizes of the blocks
$p+4$ to $p+3+m$	b – one entry per line
$p+4+m$ to $p+3+m+blk(1)*(blk(1)+1)/2$	upper triangle of block 1 of C row–wise
$\vdots$	$\vdots$
$\cdots$	upper triangle of block p of C row–wise
$\cdots$	similar section for A_1
$\vdots$	$\vdots$
$\cdots$	similar section for A_m

TABLE 3. ASCII format for SDP's with sparse blocks

Line #	Description
1	m – the number of constraints
2	l – the number of blocks
3	0 – denotes sparse blocks
4 to $p+3$	$blk(i), 1 \leq i \leq p$ – the sizes of the blocks
$p+4$ to $p+3+m$	b – one entry per line
$p+4+m$	N – the number of nonzero entries in the upper of triangle C
$p+5+m$ to $p+4+m+4*N$	the nonzero entries of C, each entry in the following format: < row #> < column #> < entry> ⋮
$\cdots$	similar section with data for A_1
⋮	⋮
$\cdots$	similar section with data for A_m

APPENDIX B. EXAMPLES

This appendix illustrates the use of the main routines in SDPpack by sample Matlab sessions. In the examples below, `>>` denotes the Matlab prompt. In all cases, Matlab was invoked from the `sdppack/` directory. Several annoying warning messages from Matlab 4 about solving ill–conditioned systems have been edited out. This sample session does not make use of the MEX files; Appendix C shows benchmarks using the MEX files.

B.1. **A randomly generated problem.**

```
>>
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Example of a randomly generated problem
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> path(path, 'testing');
>> format short e
>>
>> blk = [50 5 5 5 5 20];
>> m = 75;
>> rndinf         % generate random feasible problem with INFEASIBLE
>>                % initial points
>> setpars
>> scalefac = 1;  % X0 = Z0 = I fine for random problems
>> initvars
>> sdp
```

```
tau =   0.9990,     scalefac =       1

iter   p_step      d_step     p_infeas    d_infeas      X.Z         pobj        dobj
  0   0.000e+00   0.000e+00   1.405e+02   1.894e+03   9.000e+01   1.691e+01   0.000e+00
  1   9.787e-01   9.924e-01   2.989e+00   1.438e+01   8.926e+00   5.454e+02   5.447e+02
  2   8.311e-01   7.137e-01   5.049e-01   4.116e+00   2.745e+00   5.499e+02   5.481e+02
  3   7.393e-01   8.693e-01   1.316e-01   5.381e-01   1.395e+00   5.504e+02   5.494e+02
  4   8.680e-01   9.131e-01   1.737e-02   4.675e-02   1.850e-01   5.498e+02   5.496e+02
  5   1.000e+00   1.000e+00   1.072e-11   6.634e-13   1.603e-02   5.497e+02   5.497e+02
  6   9.702e-01   9.750e-01   1.080e-12   6.331e-13   4.790e-04   5.497e+02   5.497e+02
  7   1.000e+00   1.000e+00   4.659e-11   6.265e-13   4.710e-05   5.497e+02   5.497e+02
  8   9.990e-01   9.990e-01   1.248e-12   6.509e-13   4.713e-08   5.497e+02   5.497e+02
  9   9.990e-01   9.990e-01   1.049e-12   6.560e-13   4.714e-11   5.497e+02   5.497e+02
fsdp: stop since error reduced to desired value

sdp: elapsed time               = 206.51526 seconds
sdp: elapsed cpu time           = 161.41000 seconds
sdp: flops                      =   1.06780e+09
sdp: Number of iterations       =    9
sdp: final value of X.Z         =   4.714e-11
sdp: final primal infeasibility =   1.049e-12
sdp: final dual infeasibility   =   6.560e-13
sdp: primal objective value     =   5.4965327462494383e+02
sdp: dual objective value       =   5.4965327462489518e+02
>>
>> % note the successive reductions of X.Z by factors
>> % of 1000 in final iterations (this is because tau = 0.999)
>>
>> primalcond(A,blk,X,1.0e-06); % confirms that primal nondegenerate
primalcond =   5.965e+00
>>                              % (as expected since randomly generated)
>>
>> dualcond(A,blk,Z,1.0e-06);   % confirms that dual nondegenerate
dualcond =   6.398e+00
>>                              % (as expected since randomly generated)
```

B.2. **Artificially generated problems.**

```
>>
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Examples of artificially generated problems
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> blk = [20 10 5];
>> m = 20;
>> nosfeas        % generate problem with no strictly feasible primal point
>> setpars
>> scalefac = 1;  % X0 = Z0 = I fine for random problems
```

```
>> initvars
>> prtlevel = 0;  % disable printing of iterates
>> sdp

tau =   0.9990,     scalefac =       1

sdp: elapsed time              =  13.92766 seconds
sdp: elapsed cpu time          =  13.54000 seconds
sdp: flops                     =   5.47805e+07
sdp: Number of iterations      =   20
sdp: final value of X.Z        =   3.829e-08
sdp: final primal infeasibility =   1.300e-12
sdp: final dual infeasibility  =   7.451e-09
sdp: primal objective value    =  -5.9560845828366151e+00
sdp: dual objective value      =  -5.9560846207436846e+00
```

```
>> blk = 10;
>> m = 50;
>> r = 2;         % choose primal solution rank and
>> s = 7;         % dual solution rank in advance
>> makesdp        % generated so solution is primal degenerate

 makesdp: strict complementarity violated

 makesdp: primal nondegeneracy violated
>>
>> initvars
>> prtlevel = 0;
>> sdp

tau =   0.9990,     scalefac =       1


sdp: elapsed time              =   1.30246 seconds
sdp: elapsed cpu time          =   1.26000 seconds
sdp: flops                     =   6.23466e+06
sdp: Number of iterations      =    6
sdp: final value of X.Z        =   1.129e-11
sdp: final primal infeasibility =   2.350e-11
sdp: final dual infeasibility  =   6.537e-14
sdp: primal objective value    =   1.2626520565210694e+01
sdp: dual objective value      =   1.2626520565144881e+01
>>
>> [sort(blkeig(X,blk))  -sort(blkeig(-Z,blk))]   % sorted eigenvalues

ans =

   2.0190e-14   3.0311e+00
```

```
   4.2094e-13   1.7728e+00
   8.8589e-13   1.2308e+00
   1.0784e-12   1.0751e+00
   1.1413e-12   9.7008e-01
   1.4416e-12   7.9179e-01
   1.5300e-12   7.6035e-01
   2.2363e-12   6.2020e-01
   5.1329e-02   3.1152e-11
   6.7871e-01   1.8835e-12

>>
>> % note that convergence took place to a strictly complementary solution
>>
>> primalcond(A,blk,X,1.0e-06); % confirms that solution is primal degenerate
primalcond =         Inf
>>
>> dualcond(A,blk,Z,1.0e-06);   % check if dual degenerate
dualcond =   2.931e+00
>>
>> sdpcond(A,b,C,blk,X,y,Z);    % confirms that SDP condition number is infinite,

sdpcond: gap                      =   1.129e-11
sdpcond: primal infeasibility     =   2.350e-11
sdpcond: dual infeasibility       =   6.537e-14
sdpcond: cond estimate of 3x3 block matrix =   2.220e+15
>>                                % since SDP is degenerate
```

B.3. **A problem with no strictly complementary solution.**

```
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Example from AHO1 where no strictly
>> % complementary solution exists
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> C = [0 0 0; 0 0 0; 0 0 1];
>> A1 = [1 0 0; 0 0 0; 0 0 0];
>> A2 = [0 0 1; 0 1 0; 1 0 0];
>> A3 = [0 1 0; 1 0 0; 0 0 1];
>> b = [1 0 0]';
>> blk = 3;
>> m = 3;
>> makeA
>>
>> initvars
>> sdp

tau =   0.9990,     scalefac =        1


sdp: elapsed time              =   0.39385 seconds
```

```
sdp: elapsed cpu time           =   0.40000 seconds
sdp: flops                      =   5.72750e+04
sdp: Number of iterations       =   12
sdp: final value of X.Z         =   2.312e-09
sdp: final primal infeasibility =   2.220e-16
sdp: final dual infeasibility   =   5.752e-17
sdp: primal objective value     =   1.1559195968314993e-09
sdp: dual objective value       =  -1.1559195968284586e-09
>>
>> [sort(blkeig(X,blk))  -sort(blkeig(-Z,blk))]

ans =

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

>>
>> % the true solutions X and Z each have rank 1,
>> % but observe how the eigenvalues of the computed
>> % solution are much less accurate than for problems with
>> % strictly complementary solutions, and are not indicative
>> % of the true ranks.
>>
>> primalcond(A,blk,X,1.0e-03);   % confirms that solution is primal NONdegenerate
primalcond =   1.000e+00
>> dualcond(A,blk,Z,1.0e-03);     % confirms that solution is dual NONdegenerate
dualcond =   1.414e+00
>> sdpcond(A,b,C,blk,X,y,Z);      % condition number is infinite, since SC failed

sdpcond: gap                        =   5.000e-08
sdpcond: primal infeasibility       =   1.188e-14
sdpcond: dual infeasibility         =   3.456e-18
sdpcond: cond estimate of 3x3 block matrix =   4.124e+04
>> % cond estimate is not large because eigenvalues which should
>> % be zero are not very small
```

B.4. **A diagonally constrained problem.** This section illustrates the use of `diagcstr.m` and `dsdp.m` to generate and solve diagonally constrained problems.

```
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % A diagonally constrained problem
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> n = 50;         % n = m = 50
>> diagcstr        % random C and b, but A_k = e_k e_k^T
>> dsetpars        % set useXZ = 1, tau = .99
>> scalefac = 1;  % X0 = Z0 = I fine for random problems
```

```
>> dinitvars
>> 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   4.135e+00   5.768e+01   5.000e+01   3.281e+00   0.000e+00
  1   1.337e-02   1.000e+00   4.080e+00   0.000e+00   1.686e+03  -1.597e+03  -1.606e+03
  2   7.917e-01   9.188e-01   8.497e-01   3.158e-14   3.500e+02  -1.077e+03  -1.075e+03
  3   7.632e-01   1.000e+00   2.012e-01   3.408e-14   1.894e+02  -9.608e+02  -1.059e+03
  4   7.614e-01   1.000e+00   4.801e-02   2.010e-14   5.199e+01  -1.028e+03  -1.058e+03
  5   1.000e+00   7.896e-01   1.464e-14   3.256e-14   1.472e+01  -1.041e+03  -1.056e+03
  6   7.997e-01   1.000e+00   1.042e-14   0.000e+00   3.150e+00  -1.053e+03  -1.056e+03
  7   1.000e+00   5.721e-01   2.092e-13   2.010e-14   1.213e+00  -1.055e+03  -1.056e+03
  8   7.319e-01   1.000e+00   8.455e-14   1.421e-14   3.547e-01  -1.056e+03  -1.056e+03
  9   1.000e+00   5.730e-01   2.643e-13   0.000e+00   1.381e-01  -1.056e+03  -1.056e+03
 10   9.643e-01   1.000e+00   1.827e-12   1.421e-14   3.300e-02  -1.056e+03  -1.056e+03
 11   9.812e-01   1.000e+00   1.197e-13   3.553e-15   1.627e-03  -1.056e+03  -1.056e+03
 12   9.440e-01   1.000e+00   2.579e-13   1.465e-14   9.273e-05  -1.056e+03  -1.056e+03
 13   9.164e-01   1.000e+00   3.404e-12   1.421e-14   8.667e-06  -1.056e+03  -1.056e+03
 14   1.000e+00   1.000e+00   1.344e-11   3.553e-15   2.334e-07  -1.056e+03  -1.056e+03
 15   5.189e-01   3.699e-01   1.046e-10   0.000e+00   1.206e-07  -1.056e+03  -1.056e+03
 16   1.554e-07   1.434e-05   1.039e-10   1.465e-14   1.202e-07  -1.056e+03  -1.056e+03
fdsdp: stop since steps are too short

dsdp: elapsed time                =   3.32932 seconds
dsdp: elapsed cpu time            =   3.19000 seconds
dsdp: flops                       =   7.31872e+07
dsdp: Number of iterations        =   16
dsdp: final value of X.Z          =   1.202e-07
dsdp: final primal infeasibility =   1.039e-10
dsdp: final dual infeasibility    =   1.465e-14
dsdp: primal objective value      =  -1.0559774619074481e+03
dsdp: dual objective value        =  -1.0559774620433770e+03
```

```
>> setpars        % sets useXZ = 0, tau = .999
>> scalefac = 1;  % X0 = Z0 = I fine for random problems
>> dinitvars
>> dsdp           % since useXZ = 0, general-purpose XZ+ZX code used

dsdp: using XZ+ZX method...


tau =   0.9990,     scalefac =       1
```

```
iter   p_step      d_step     p_infeas    d_infeas      X.Z         pobj        dobj
  0   0.000e+00   0.000e+00   4.135e+00   5.768e+01   5.000e+01   3.281e+00   0.000e+00
  1   1.349e-02   1.000e+00   4.079e+00   0.000e+00   1.671e+03  -1.611e+03  -1.606e+03
  2   6.448e-01   1.000e+00   1.449e+00   3.843e-14   5.957e+02  -1.109e+03  -1.122e+03
  3   8.000e-01   9.591e-01   2.898e-01   1.628e-14   1.410e+02  -1.040e+03  -1.059e+03
  4   8.515e-01   8.056e-01   4.305e-02   2.010e-14   7.150e+01  -1.004e+03  -1.056e+03
  5   1.000e+00   1.000e+00   1.696e-12   2.842e-14   2.556e+01  -1.031e+03  -1.056e+03
  6   9.985e-01   1.000e+00   2.903e-15   1.421e-14   3.886e-02  -1.056e+03  -1.056e+03
  7   9.990e-01   9.990e-01   3.981e-15   1.465e-14   3.886e-05  -1.056e+03  -1.056e+03
  8   9.990e-01   9.990e-01   7.745e-16   3.553e-15   3.886e-08  -1.056e+03  -1.056e+03
  9   9.989e-01   9.990e-01   7.946e-16   3.553e-15   4.200e-11  -1.056e+03  -1.056e+03
fsdp: stop since limiting accuracy reached (smallest eigenvalue of Z =  -5.453e-14)

dsdp: elapsed time               =  22.59205 seconds
dsdp: elapsed cpu time           =  13.13000 seconds
dsdp: flops                      =   5.35133e+08
dsdp: Number of iterations       =     9
dsdp: final value of X.Z         =   4.200e-11
dsdp: final primal infeasibility =   7.946e-16
dsdp: final dual infeasibility   =   3.553e-15
dsdp: primal objective value     =  -1.0559774620403921e+03
dsdp: dual objective value       =  -1.0559774620404330e+03
>>
>> % notice that specialized XZ method is faster but less
>> % accurate than XZ+ZX method
```

B.5. **A Lovász θ function problem.**

```
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % A Lovasz theta function problem
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> %
>> n = 30;        % number of vertices
>> dsty = 0.2;    % edge density is 20%
>> thetarnd       % random graph with random weights
>>
>> lsetpars       % set useXZ = 1, tau = .99
>> scalefac = 1;  % X0 = Z0 = I fine for random problems
>> validate = 1;  % check connectivity
>> linitvars
>> 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   1.821e+01   3.000e+01  -1.640e+01   0.000e+00
  1   4.392e-02   5.159e-02   2.773e+01   1.727e+01   3.436e+01  -1.150e+02  -4.542e-01
  2   1.491e-02   1.000e+00   2.731e+01   1.159e-14   3.992e+02  -1.076e+02  -1.790e+01
  3   1.000e+00   1.000e+00   4.775e-15   5.700e-15   1.240e+01  -4.591e+00  -1.699e+01
  4   1.000e+00   8.510e-01   2.260e-16   6.862e-15   1.719e+00  -5.909e+00  -7.627e+00
  5   6.299e-01   1.000e+00   3.197e-16   2.152e-15   6.784e-01  -6.752e+00  -7.431e+00
  6   7.758e-01   9.734e-01   3.486e-16   2.583e-15   1.905e-01  -7.115e+00  -7.305e+00
  7   7.659e-01   1.000e+00   1.108e-15   3.490e-15   8.253e-02  -7.221e+00  -7.304e+00
  8   9.001e-01   1.000e+00   3.232e-15   1.899e-15   1.723e-02  -7.284e+00  -7.302e+00
  9   9.767e-01   9.953e-01   6.683e-16   1.890e-15   5.154e-04  -7.300e+00  -7.300e+00
 10   9.880e-01   1.000e+00   3.142e-14   2.346e-15   1.962e-05  -7.300e+00  -7.300e+00
 11   9.558e-01   1.000e+00   2.754e-14   2.140e-15   1.776e-06  -7.300e+00  -7.300e+00
 12   9.368e-01   1.000e+00   4.841e-13   2.329e-15   1.117e-07  -7.300e+00  -7.300e+00
 13   1.000e+00   1.000e+00   4.403e-13   2.901e-15   1.215e-08  -7.300e+00  -7.300e+00
Stop since new point is substantially worse than current iterate
      X.Z =   3.672e-10
      pri_infeas =   4.895e-12
      dual_infeas =   2.557e-15

lsdp: elapsed time                 =   7.77212 seconds
lsdp: elapsed cpu time             =   7.58000 seconds
lsdp: flops                        =   1.96740e+07
lsdp: Number of iterations         =   13
lsdp: final value of X.Z           =   1.215e-08
lsdp: final primal infeasibility   =   4.403e-13
lsdp: final dual infeasibility     =   2.901e-15
lsdp: primal objective value       =  -7.3004453174538488e+00
lsdp: dual objective value         =  -7.3004453295989862e+00
lsdp: Lovasz theta function value =   7.3004453235264180e+00
```

```
>> setpars        % sets useXZ = 0, tau = .999
>> scalefac = 1;  % X0 = Z0 = I fine for random problems
>> validate = 1;  % check connectivity
>> linitvars
>> lsdp           % since useXZ = 0, general-purpose code XZ+ZX used

lsdp: using XZ+ZX method...


tau =   0.9990,     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   1.821e+01   3.000e+01  -1.640e+01   0.000e+00
  1   4.432e-02   5.205e-02   2.771e+01   1.726e+01   3.434e+01  -1.159e+02  -4.580e-01
  2   6.103e-02   1.000e+00   2.602e+01   8.154e-15   3.015e+02  -8.796e+01  -1.441e+01
  3   1.000e+00   1.000e+00   1.059e-14   5.238e-15   9.234e+00  -4.455e+00  -1.369e+01
  4   1.000e+00   8.729e-01   3.455e-15   5.401e-15   1.129e+00  -6.274e+00  -7.403e+00
  5   8.498e-01   7.268e-01   2.676e-15   4.130e-15   2.300e-01  -7.079e+00  -7.309e+00
```

```
  6   1.000e+00   1.000e+00   9.434e-14   4.110e-15   1.352e-01  -7.180e+00  -7.315e+00
  7   9.833e-01   9.939e-01   1.582e-15   5.173e-15   2.124e-03  -7.298e+00  -7.300e+00
  8   9.990e-01   9.990e-01   6.664e-16   4.506e-15   2.142e-06  -7.300e+00  -7.300e+00
  9   9.990e-01   9.990e-01   5.556e-16   3.820e-15   2.142e-09  -7.300e+00  -7.300e+00
 10   9.990e-01   9.990e-01   2.234e-16   3.210e-15   2.169e-12  -7.300e+00  -7.300e+00
fsdp: stop since error reduced to desired value

lsdp: elapsed time               =   9.36495 seconds
lsdp: elapsed cpu time           =   9.14000 seconds
lsdp: flops                      =   2.09046e+08
lsdp: Number of iterations       =   10
lsdp: final value of X.Z         =   2.169e-12
lsdp: final primal infeasibility =   2.234e-16
lsdp: final dual infeasibility   =   3.210e-15
lsdp: primal objective value     =  -7.3004453290700102e+00
lsdp: dual objective value       =  -7.3004453290721765e+00
lsdp: Lovasz theta function value =   7.3004453290710938e+00
>>
>> % notice that specialized XZ method is faster
>> % but less accurate than XZ+ZX method
>>
>> % since useXZ = 0, A was formed; so we can now call primalcond
>> % and dualcond
>>
>> % for random weights, usually Lovasz SDP is primal degenerate
>> %
>> rank(X,1.0e-06)     % for random weights, usually rank(X) is 1

ans =

     1

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

ans =

    29

>> primalcond(A,blk,X,1.0e-06);  % usually primal degenerate
primalcond =         Inf
>> dualcond(A,blk,Z,1.0e-06);    % usually dual nondegenerate
dualcond =   1.000e+00
```

```
>> w = ones(size(w));  % change weights to all one
>> lsetpars            % set useXZ = 1, tau = .99
>> scalefac = 1;       % X0 = Z0 = I fine for random problems
>> prtlevel = 0;       % turn off detailed output
>> validate = 1;
```

```
>> linitvars
>> lsdp               % since useXZ = 1, special-purpose XZ code used

lsdp: using XZ method...


tau =   0.9900,     scalefac =       1


lsdp: elapsed time               =   8.22000 seconds
lsdp: elapsed cpu time           =   8.08000 seconds
lsdp: flops                      =   2.10767e+07
lsdp: Number of iterations       =   15
lsdp: final value of X.Z         =   3.563e-09
lsdp: final primal infeasibility =   3.244e-10
lsdp: final dual infeasibility   =   4.295e-15
lsdp: primal objective value     =  -1.1999999995647892e+01
lsdp: dual objective value       =  -1.2000000000398519e+01
lsdp: Lovasz theta function value =   1.1999999998023206e+01
```

```
>> setpars            % sets useXZ = 0, tau = .999
>> prtlevel = 0;
>> scalefac = 1;      % X0 = Z0 = I fine for random problems
>> validate = 1;      % check connectivity
>> linitvars
>> lsdp               % since useXZ = 0, general-purpose XZ+ZX code used

lsdp: using XZ+ZX method...


tau =   0.9990,     scalefac =       1

lsdp: elapsed time               =   9.36421 seconds
lsdp: elapsed cpu time           =   9.10000 seconds
lsdp: flops                      =   2.08908e+08
lsdp: Number of iterations       =   10
lsdp: final value of X.Z         =   7.552e-12
lsdp: final primal infeasibility =   2.508e-16
lsdp: final dual infeasibility   =   6.900e-15
lsdp: primal objective value     =  -1.1999999999993703e+01
lsdp: dual objective value       =  -1.2000000000001263e+01
lsdp: Lovasz theta function value =   1.1999999999997483e+01
>>
>> % notice that specialized XZ method is faster but less
>> % accurate than XZ+ZX method
>>
>> % since useXZ = 0, the matrix A was formed, so we can now call
>> % primalcond and dualcond
```

```
>>
>> rank(X,1.0e-06)  % for weights all one, usually rank(X) > 1

ans =

     3

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

ans =

    27

>> primalcond(A,blk,X,1.0e-06);  % sometimes primal nondegenerate
primalcond =   4.249e+17
>> dualcond(A,blk,Z,1.0e-06);    % sometimes dual degenerate
dualcond =   5.881e+10
```

B.6. **A sample truss problem.**

```
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Sample truss problem
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> load testing/truss/truss1     % from Nemirovskii
>> setpars                       % sets scalefac = 100
>> initvars
>> sdp

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   2.359e-12   8.789e-15   2.470e+02   2.471e+02   1.014e-01
  3   6.313e-01   1.000e+00   8.669e-13   1.194e-16   9.229e+01   9.189e+01  -4.013e-01
  4   8.211e-01   1.000e+00   1.635e-13   1.688e-16   1.668e+01   1.745e+01   7.700e-01
  5   5.064e-03   4.190e-01   1.635e-13   2.267e-15   1.004e+01   1.746e+01   7.424e+00
  6   1.000e+00   9.086e-01   2.853e-13   1.088e-15   1.254e+00   1.006e+01   8.809e+00
  7   9.886e-01   9.977e-01   2.754e-13   2.432e-15   1.919e-02   9.018e+00   8.999e+00
  8   9.990e-01   9.990e-01   1.535e-13   2.035e-15   1.948e-05   9.000e+00   9.000e+00
  9   9.990e-01   9.990e-01   9.392e-14   1.936e-15   1.948e-08   9.000e+00   9.000e+00
 10   9.990e-01   9.990e-01   3.879e-14   2.220e-15   1.971e-11   9.000e+00   9.000e+00
fsdp: stop since error reduced to desired value

sdp: elapsed time               =   1.30909 seconds
sdp: elapsed cpu time           =   1.26000 seconds
sdp: flops                      =   8.99130e+04
sdp: Number of iterations       =   10
```

```
sdp: final value of X.Z         =   1.971e-11
sdp: final primal infeasibility =   3.879e-14
sdp: final dual infeasibility   =   2.220e-15
sdp: primal objective value     =   8.9999963153051237e+00
sdp: dual objective value       =   8.9999963152853795e+00
>> primalcond(A,blk,X,1.0e-06);  % check if primal degenerate
primalcond =         Inf
>> dualcond(A,blk,Z,1.0e-06);    % check if dual degenerate
dualcond =   8.920e+00
```

B.7. **A sample LMI problem.**

```
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> % Sample LMI problem
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>>
>> load testing/lmi/hinf1        % from Gahinet
>> setpars                       % sets scalefac = 100
>> initvars
>> sdp

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   4.536e+02   3.742e+02   1.400e+05   0.000e+00   0.000e+00
  1   8.702e-01   1.000e+00   5.887e+01   3.553e-15   1.462e+04  -2.311e+00  -1.423e+02
  2   9.964e-01   1.000e+00   2.138e-01   5.040e-14   1.881e+02  -8.974e-02  -1.366e+02
  3   6.524e-01   9.997e-01   7.431e-02   2.807e-14   1.424e+01  -1.268e-01  -5.962e+00
  4   9.296e-01   8.036e-01   5.232e-03   1.974e-14   1.969e+00  -9.935e-01  -2.389e+00
  5   6.838e-01   1.000e+00   1.654e-03   2.169e-14   1.164e+00  -1.614e+00  -2.462e+00
  6   9.780e-01   9.855e-01   3.633e-05   3.224e-14   2.058e-02  -2.029e+00  -2.043e+00
  7   8.210e-01   1.000e+00   6.502e-06   4.463e-14   4.563e-03  -2.032e+00  -2.035e+00
  8   7.948e-01   1.000e+00   1.334e-06   8.362e-14   1.583e-03  -2.033e+00  -2.033e+00
  9   7.530e-01   1.000e+00   3.291e-07   1.996e-13   9.761e-04  -2.032e+00  -2.033e+00
 10   1.465e-02   7.657e-01   3.243e-07   3.405e-13   9.435e-04  -2.032e+00  -2.033e+00
 11   1.473e-01   1.567e-01   8.776e-07   3.623e-13   8.924e-04  -2.032e+00  -2.033e+00
 12   8.588e-01   8.535e-01   2.237e-07   3.358e-13   1.017e-04  -2.033e+00  -2.033e+00
 13   8.159e-01   1.000e+00   2.989e-06   4.530e-13   2.650e-05  -2.033e+00  -2.033e+00
 14   5.789e-01   3.171e-01   1.110e-06   5.029e-13   2.717e-05  -2.033e+00  -2.033e+00
 15   5.643e-03   5.534e-02   1.107e-06   6.097e-13   3.097e-05  -2.033e+00  -2.033e+00
 16   3.933e-01   5.692e-01   7.331e-07   1.436e-12   1.992e-05  -2.033e+00  -2.033e+00
 17   4.891e-01   7.193e-01   3.623e-07   8.719e-13   1.067e-05  -2.033e+00  -2.033e+00
 18   1.000e+00   1.000e+00   2.734e-07   1.033e-12   8.867e-06  -2.033e+00  -2.033e+00
 19   7.159e-01   1.000e+00   1.412e-07   1.974e-12   2.106e-06  -2.033e+00  -2.033e+00
 20   1.000e+00   1.000e+00   6.915e-08   2.002e-12   2.749e-08  -2.033e+00  -2.033e+00
 21   9.990e-01   9.990e-01   2.089e-07   1.097e-12   3.033e-11  -2.033e+00  -2.033e+00
 22   1.000e+00   1.000e+00   2.279e-07   1.079e-12   4.434e-12  -2.033e+00  -2.033e+00
fsdp: stop since limiting accuracy reached.
(new X is indefinite, hence rejected)
```

```
sdp: elapsed time               =   3.43612 seconds
sdp: elapsed cpu time           =   3.36000 seconds
sdp: flops                      =   1.84914e+06
sdp: Number of iterations       =   22
sdp: final value of X.Z         =   4.434e-12
sdp: final primal infeasibility =   2.279e-07
sdp: final dual infeasibility   =   1.079e-12
sdp: primal objective value     =  -2.0326830740965565e+00
sdp: dual objective value       =  -2.0326415068447088e+00
>> primalcond(A,blk,X,1.0e-06);  % check if primal degenerate
primalcond =   1.699e+15
>> dualcond(A,blk,Z,1.0e-06);    % check if dual degenerate
dualcond =   9.738e+00
```

Appendix C. Benchmarks

This appendix provides benchmarks of SDPpack Version 0.8 BETA on some randomly generated test problems, a set of 16 LMI problems[8] from control applications, and a set of 8 problems[9] from truss topology design. The LMI and the truss design problems are available as `mat` files from the SDPpack home page.

The truss design problems were solved with `sdp.m` with the default values of parameters set using `setpars.m`:

- `prtlevel` = 1
- `validate` = 0
- `maxit` = 100
- `tau` = 0.999
- `scalefac` = 100.0
- `autorestart` = 1
- `reltol` = 10^{-11}
- `abstol` = 10^{-8}
- `steptol` = 10^{-8}
- `gapprogtol` = 100.0
- `feasprogtol` = 5.0
- `bndtol` = 10^{8}

The LMI problems were solved with the same parameter settings above, but with `scalefac` = 1000.0. The benchmarks were conducted on an SGI workstation with a MIPS R10000 processor, a MIPS R10010 floating point unit and 192 MB of main memory. In the tables, `pinfeas`, `dinfeas` and `dgap` are shown on a log scale ($\log_{10}$). Each * next to `termflag` indicates a restart.

[8] These were provided to us by P. Gahinet.

[9] These were provided to us by A. Nemirovskii.

TABLE 4. Randomly generated problems: (1) SDP with `blk` = [20 20] and $m = 40$ solved with XZ+ZX, (2) diagonally constrained SDP with $n = 20$ solved with XZ, (3) same problem as in (2) solved with XZ+ZX, (4) Lovász θ function with $n = 10$ and `dsty` = 0.2 solved with XZ, and (5) same problem as in (4) solved with XZ+ZX.

Problem	`iter`	`pinfeas`	`dinfeas`	`dgap`	CPU secs	`termflag`
1	10	-11	-13	-12	6.6e+00	0
2	20	-09	-14	-06	2.3e+00	5
3	8	-16	-15	-11	3.3e+00	0
4	12	-14	-15	-07	1.7e+00	4
5	9	-14	-15	-10	8.9e-01	0

TABLE 5. Randomly generated problems: (1) SDP with `blk` = [40 40] and $m = 80$ solved with XZ+ZX, (2) diagonally constrained SDP with $n = 40$ solved with XZ, (3) same problem as in (2) solved with XZ+ZX, (4) Lovász θ function with $n = 20$ and `dsty` = 0.2 solved with XZ, and (5) same problem as in (4) solved with XZ+ZX.

Problem	`iter`	`pinfeas`	`dinfeas`	`dgap`	CPU secs	`termflag`
1	10	-11	-13	-11	7.4e+01	0
2	20	-10	-14	-07	1.2e+01	5
3	10	-15	-15	-09	4.4e+01	0
4	16	-07	-15	-08	2.5e+01	3
5	17	-13	-15	-12	2.0e+01	0

TABLE 6. Randomly generated problems: (1) SDP with `blk` = [60 60] and $m = 120$ solved with XZ+ZX, (2) diagonally constrained SDP with $n = 60$ solved with XZ, (3) same problem as in (2) solved with XZ+ZX, (4) Lovász θ function with $n = 30$ and `dsty` = 0.2 solved with XZ, and (5) same problem as in (4) solved with XZ+ZX.

Problem	`iter`	`pinfeas`	`dinfeas`	`dgap`	CPU secs	`termflag`
1	9	-11	-12	-12	2.9e+02	0
2	24	-10	-Inf	-06	4.2e+01	5
3	12	-15	-14	-08	2.6e+02	1
4	20	-12	-15	-07	1.6e+02	4
5	13	-16	-15	-10	1.0e+02	0

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`

TABLE 7. Benchmarks on problems from truss topology design

P	n	m	#it	pinfeas	dinfeas	dgap	$C \bullet X$	$b^T y$	CPU secs	termflag
1	13	6	10	–14	–15	–11	9.00e+00	9.00e+00	3.1e–01	0
2	133	58	12	–12	–14	–11	1.23e+02	1.23e+02	3.8e+00	2
3	31	27	15	–13	–15	–08	9.11e+00	9.11e+00	1.4e+00	4
4	19	12	11	–13	–15	–11	9.01e+00	9.01e+00	4.7e–01	0
5	331	208	15	–10	–14	–09	1.32e+02	1.32e+02	7.2e+01	0
6	451	172	39	–06	–13	–08	9.01e+02	9.01e+02	6.4e+01	5
7	301	86	45	–06	–13	–10	9.00e+02	9.00e+02	3.0e+01	2
8	628	496	18	–10	–14	–11	1.33e+02	1.33e+02	1.1e+03	2

TABLE 8. Benchmarks on LMI problems from control applications

P	n	m	#it	pinfeas	dinfeas	dgap	$C \bullet X$	$b^T y$	CPU secs	termflag
1	14	13	13	–08	–12	–08	–2.03e+00	–2.03e+00	6.9e–01	4
2	16	13	14	–11	–11	–08	–1.09e+01	–1.09e+01	7.6e–01	1*
3	16	13	15	–07	–11	–08	–5.69e+01	–5.69e+01	8.9e–01	2
4	16	13	15	–07	–10	–10	–2.74e+02	–2.74e+02	8.4e–01	1
5	16	13	16	–04	–11	–08	–3.62e+02	–3.62e+02	8.9e–01	2
6	16	13	35	–03	–11	–05	–4.48e+02	–4.48e+02	1.8e+00	2**
7	16	13	11	–04	–11	–08	–3.90e+02	–3.90e+02	6.3e–01	4
8	16	13	14	–05	–12	–07	–1.16e+02	–1.16e+02	7.3e–01	1
9	16	13	17	–06	–13	–10	–2.36e+02	–2.36e+02	9.8e–01	1
10	18	21	26	–07	–07	–05	–1.08e+02	–1.08e+02	2.0e+00	–1
11	22	31	26	–06	–07	–06	–6.59e+01	–6.59e+01	3.3e+00	2
12	24	43	25	–08	–08	–01	–1.99e-01	–1.77e-01	4.4e+00	–1
13	30	57	19	–05	–09	–02	–4.43e+01	–4.43e+01	5.9e+00	2
14	34	73	28	–08	–09	–04	–1.29e+01	–1.29e+01	1.5e+01	5*
15	37	91	19	–06	–08	–02	–2.39e+01	–2.39e+01	2.1e+01	2
37	16	13	17	–06	–13	–10	–2.36e+02	–2.36e+02	9.9e–01	1