CSDP 5.0 User's Guide

Brian Borchers

September 1, 2005

Introduction

CSDP is a software package for solving semidefinite programming problems. The algorithm is a predictor–corrector version of the primal–dual barrier method of Helmberg, Rendl, Vanderbei, and Wolkowicz [3]. A more detailed, but now somewhat outdated description of the algorithms in CSDP can be found in [1]. CSDP is written in C for efficiency and portability. The code is designed to make use of highly optimized linear algebra routines from the LAPACK and BLAS libraries.

CSDP also has a number of features that make it very flexible. CSDP can work with general symmetric matrices or with matrices that have defined block diagonal structure. CSDP is designed to handle constraint matrices with general sparse structure. The code takes advantage of this structure in efficiently constructing the system of equations that is solved at each iteration of the algorithm.

In addition to its default termination criteria, CSDP includes a feature that allows the user to terminate the solution process after any iteration. For example, this feature can be used within a cutting plane scheme to terminate the solution process as soon as cutting planes have been identified. The CSDP library contains routines for writing SDP problems and solutions to files and reading problems and solutions from files.

A stand alone solver program is included for solving SDP problems that have been written in the SDPA sparse format [2]. An interface to MATLAB is also provided. The MATLAB routine can be used to solve problems that are in the format used by the SeDuMi [5].

This document describes how to install CSDP and how to use the stand alone solver, MATLAB interface, and library routines after CSDP has been installed.

The SDP Problem

CSDP solves semidefinite programming problems of the form

$$\begin{aligned}
\max \quad & \operatorname{tr}(CX) \\
A(X) &= a \\
X &\succeq 0
\end{aligned} \tag{1}$$

where

$$A(X) = \begin{bmatrix} \operatorname{tr}(A_1 X) \\ \operatorname{tr}(A_2 X) \\ \cdots \\ \operatorname{tr}(A_m X) \end{bmatrix}. \tag{2}$$

Here $X \succeq 0$ means that X is positive semidefinite. All of the matrices A_i, X, and C are assumed to be real and symmetric.

The dual of this SDP is

$$\begin{aligned}
\min \quad & a^T y \\
A^T(y) - C &= Z \\
Z &\succeq 0
\end{aligned} \tag{3}$$

where

$$A^T(y) = \sum_{i=1}^{m} y_i A_i. \tag{4}$$

Other semidefinite programming packages use slight variations on this primal–dual pair. For example, the primal–dual pair used in SDPA interchanges the primal and dual problems.

Users of CSDP can specify their own termination criteria. However, the default criteria are that

$$\begin{aligned}
\frac{\operatorname{tr}(XZ)}{1+|a^T y|} &< 1.0 \times 10^{-8} \\
\frac{\|A(x)-a\|_2}{1+\|a\|_2} &< 1.0 \times 10^{-8} \\
\frac{\|A^T(y)-C-Z\|_F}{1+\|C\|_F} &< 1.0 \times 10^{-8} \\
X, Z &\succeq 0.
\end{aligned}$$

Note that for feasible primal and dual solutions, $a^T y - \operatorname{tr}(CX) = \operatorname{tr}(XZ)$. Thus the first of these criteria insures that the relative duality gap is small. In practice, there are sometimes solutions which satisfy our primal and dual feasibility tolerances but have duality gaps which are not close to $\operatorname{tr}(XZ)$. In some cases, the duality gap may even become negative. Because of this ambiguity, we use the $\operatorname{tr}(XZ)$ gap instead of the difference between the objective functions. An option in the param.csdp file allows CSDP to use the difference of the primal objective functions instead of the $\operatorname{tr}(XZ)$ gap.

The matrices X and Z are considered to be positive definite when their Cholesky factorizations can be computed. In practice, this is somewhat more conservative than simply requiring all eigenvalues to be nonnegative.

The Seventh DIMACS Implementation Challenge used a slightly different set of error measures [4]. For convenience in benchmarking, CSDP includes these DIMACS error measures in its output.

To test for primal infeasibility, CSDP checks the inequality

$$\frac{-a^T y}{\|A^T(y) - Z\|_F} > 1.0 \times 10^8.$$

If CSDP detects that a problem is primal infeasible, then it will announce this in its output and return a dual solution with $a^T y = -1$, and $\|A^T(y) - Z\|$ very small. This acts as a certificate of primal infeasibility.

Similarly, CSDP tests for dual infeasibility by checking

$$\frac{\text{tr}(CX)}{\|A(X)\|_2} > 1.0 \times 10^8.$$

If CSDP detects that a problem is dual infeasible, it announces this in its output and returns a primal solution with $\text{tr}(CX) = 1$, and $\|A(X)\|$ small. This acts as a certificate of the dual infeasibility.

The tolerances for primal and dual feasibility and the relative duality gap can be changed by editing CSDP's parameter file. See the following section on using the stand alone solver for a description of this parameter file.

Using the stand alone solver

CSDP includes a program which can be used to solve SDP's that have been written in the SDPA sparse format. Usage is

```
csdp <problem file> [<final solution>] [<initial solution>]
```

where `<problem file>` is the name of a file containing the SDP problem in SDPA sparse format, `final solution` is the optional name of a file in which to save the final solution, and `initial solution` is the optional name of a file from which to take the initial solution.

The following example shows how CSDP would be used to solve a test problem.

```
>csdp theta1.dat-s
Iter:  0 Ap: 0.00e+00 Pobj:  1.4644661e+04 Ad: 0.00e+00 Dobj:  0.0000000e+00
Iter:  1 Ap: 9.31e-01 Pobj:  5.7513865e+03 Ad: 1.00e+00 Dobj:  8.0172003e+01
Iter:  2 Ap: 9.21e-01 Pobj:  2.3227402e+02 Ad: 1.00e+00 Dobj:  8.2749235e+01
Iter:  3 Ap: 9.30e-01 Pobj:  1.0521019e+01 Ad: 1.00e+00 Dobj:  8.4447722e+01
Iter:  4 Ap: 1.00e+00 Pobj:  2.5047625e+00 Ad: 1.00e+00 Dobj:  7.2126480e+01
Iter:  5 Ap: 1.00e+00 Pobj:  7.5846337e+00 Ad: 1.00e+00 Dobj:  4.2853659e+01
Iter:  6 Ap: 1.00e+00 Pobj:  1.5893126e+01 Ad: 1.00e+00 Dobj:  3.0778169e+01
Iter:  7 Ap: 1.00e+00 Pobj:  1.9887401e+01 Ad: 1.00e+00 Dobj:  2.4588662e+01
Iter:  8 Ap: 1.00e+00 Pobj:  2.1623330e+01 Ad: 1.00e+00 Dobj:  2.3465172e+01
```

```
Iter:  9 Ap: 1.00e+00 Pobj:  2.2611983e+01 Ad: 1.00e+00 Dobj:  2.3097049e+01
Iter: 10 Ap: 1.00e+00 Pobj:  2.2939498e+01 Ad: 1.00e+00 Dobj:  2.3010908e+01
Iter: 11 Ap: 1.00e+00 Pobj:  2.2996259e+01 Ad: 1.00e+00 Dobj:  2.3000637e+01
Iter: 12 Ap: 1.00e-00 Pobj:  2.2999835e+01 Ad: 1.00e+00 Dobj:  2.3000020e+01
Iter: 13 Ap: 1.00e+00 Pobj:  2.2999993e+01 Ad: 1.00e+00 Dobj:  2.2999999e+01
Iter: 14 Ap: 1.00e+00 Pobj:  2.3000000e+01 Ad: 1.00e+00 Dobj:  2.3000000e+01
Success: SDP solved
Primal objective value: 2.3000000e+01
Dual objective value: 2.3000000e+01
Relative primal infeasibility: 4.45e-19
Relative dual infeasibility: 3.93e-09
Real Relative Gap: 7.21e-09
XZ Relative Gap: 7.82e-09
DIMACS error measures: 4.45e-19 0.00e+00 1.00e-07 0.00e+00 7.21e-09 7.82e-09
```

One line of output appears for each iteration of the algorithm, giving the
iteration number, primal step size (Ap), primal objective value (Pobj), dual step
size (Ad), and dual objective value (Dobj). The last eight lines of output show
the primal and dual optimal objective values, the XZ duality gap, the actual
duality gap, the relative primal and dual infeasibility in the optimal solution.

CSDP searches for a file named "param.csdp" in the current directory. If
no such file exists, then default values for all of CSDP's parameters are used. If
there is a parameter file, then CSDP reads the parameter values from this file.
A sample file containing the default parameter values is given below.

```
axtol=1.0e-8
atytol=1.0e-8
objtol=1.0e-8
pinftol=1.0e8
dinftol=1.0e8
maxiter=100
minstepfrac=0.90
maxstepfrac=0.97
minstepp=1.0e-8
minstepd=1.0e-8
usexzgap=1
tweakgap=0
affine=0
printlevel=1
perturbobj=1
fastmode=0
```

The first three parameters, axtol, atytol, and objtol are the tolerances for
primal feasibility, dual feasibility, and relative duality gap. The parameters pin-
ftol and dinftol are tolerances used in determining primal and dual infeasibility.
The maxiter parameter is used to limit the total number of iterations that CSDP
may use. The minstepfrac and maxstepfrac parameters determine how close to

the edge of the feasible region CSDP will step. If the primal or dual step is shorter than minstepp or minstepd, then CSDP declares a line search failure. If parameter usexzgap is 0, then CSDP will use the objective function duality gap instead of the $\text{tr}(XZ)$ gap. If tweakgap is set to 1, and usexzgap is set to 0, then CSDP will attempt to "fix" negative duality gaps. If parameter affine is set to 1, then CSDP will take only primal–dual affine steps and not make use of the barrier term. This can be useful for some problems that do not have feasible solutions that are strictly in the interior of the cone of semidefinite matrices. The printlevel parameter determines how much debugging information is output. Use printlevel=0 for no output and printlevel=1 for normal output. Higher values of printlevel will generate more debugging output. The perturbobj parameter determines whether the objective function will be perturbed to help deal with problems that have unbounded optimal solution sets. The fastmode parameter determines whether or not CSDP will skip certain time consuming operations that slightly improve the accuracy of the solutions. If fastmode is set to 1, then CSDP may be much faster, but also somewhat less accurate.

Calling CSDP from MATLAB

An interface to the stand alone solver from MATLAB has been provided in CSDP 5.0. This interface accepts problems in the format used by the MATLAB package SeDuMi 1.05. The usage is

```
%
% [x,y,z,info]=csdp(At,b,c,K,pars)
%
% Uses CSDP to solve a problem in SeDuMi format.
%
% Input:
%       At, b, c, K      SDP problem in SeDuMi format.
%       pars             CSDP parameters (optional parameter.)
%
% Output:
%
%       x, y, z          solution.
%       info             CSDP return code.
%                        info=100 indicates a failure in the MATLAB
%                        interface, such as inability to write to
%                        a temporary file.
%
% Note: This interface makes use of temporary files with names given by the
% tempname function.  This will fail if there is no working temporary
% directory or there isn't enough space available in this directory.
%
% Note: This code writes its own param.csdp file in the current working
```

```
% directory.  Any param.csdp file already in the directory will be deleted.
%
% Note: It is assumed that csdp is the search path made available through
% the ``system'' or ``dos'' command.  Typically, having the csdp executable in
% current working directory will work, although some paranoid system
% administrators keep . out of the path.  In that case, you'll need to
% install csdp in one of the directories that is in the search path.
% A simple test is to run csdp from a command line prompt.
```

The following example shows the solution of a sample problem using this interface. To make the example more interesting, we've asked CSDP to find a solution with a relative duality gap smaller than 1.0e-9 instead of the usual 1.0e-8.

```
>> load control1.mat
>> whos
  Name       Size                  Bytes  Class

  At        125x21                  8488  double array (sparse)
  K           1x1                    140  struct array
  ans         2x1                     16  double array
  b          21x1                    168  double array
  c         125x1                     80  double array (sparse)

Grand total is 732 elements using 8892 bytes

>> pars.objtol=1.0e-9

pars =

    objtol: 1.0000e-09

>> [x,y,z,info]=csdp(At,b,c,K,pars);
Transposing A to match b
Number of constraints: 21
Number of SDP blocks: 2
Number of LP vars: 0
Iter:  0 Ap: 0.00e+00 Pobj:  3.6037961e+02 Ad: 0.00e+00 Dobj:  0.0000000e+00
Iter:  1 Ap: 9.56e-01 Pobj:  3.7527534e+02 Ad: 9.60e-01 Dobj:  6.4836002e+04
Iter:  2 Ap: 8.55e-01 Pobj:  4.0344779e+02 Ad: 9.67e-01 Dobj:  6.9001508e+04
Iter:  3 Ap: 8.77e-01 Pobj:  1.4924982e+02 Ad: 1.00e+00 Dobj:  6.0425319e+04
Iter:  4 Ap: 7.14e-01 Pobj:  8.2819408e+01 Ad: 1.00e+00 Dobj:  1.2926534e+04
Iter:  5 Ap: 8.23e-01 Pobj:  4.7411688e+01 Ad: 1.00e+00 Dobj:  4.9040115e+03
Iter:  6 Ap: 7.97e-01 Pobj:  2.6300212e+01 Ad: 1.00e+00 Dobj:  1.4672743e+03
Iter:  7 Ap: 7.12e-01 Pobj:  1.5215577e+01 Ad: 1.00e+00 Dobj:  4.0561826e+02
Iter:  8 Ap: 8.73e-01 Pobj:  7.5119215e+00 Ad: 1.00e+00 Dobj:  1.7418715e+02
```

```
Iter:  9 Ap: 9.87e-01 Pobj:  5.3076518e+00 Ad: 1.00e+00 Dobj:  5.2097312e+01
Iter: 10 Ap: 1.00e+00 Pobj:  7.8594672e+00 Ad: 1.00e+00 Dobj:  2.2172435e+01
Iter: 11 Ap: 8.33e-01 Pobj:  1.5671237e+01 Ad: 1.00e+00 Dobj:  2.1475840e+01
Iter: 12 Ap: 1.00e+00 Pobj:  1.7250217e+01 Ad: 1.00e+00 Dobj:  1.8082715e+01
Iter: 13 Ap: 1.00e+00 Pobj:  1.7710018e+01 Ad: 1.00e+00 Dobj:  1.7814069e+01
Iter: 14 Ap: 9.99e-01 Pobj:  1.7779600e+01 Ad: 1.00e+00 Dobj:  1.7787170e+01
Iter: 15 Ap: 1.00e+00 Pobj:  1.7783579e+01 Ad: 1.00e+00 Dobj:  1.7785175e+01
Iter: 16 Ap: 1.00e+00 Pobj:  1.7784494e+01 Ad: 1.00e+00 Dobj:  1.7784708e+01
Iter: 17 Ap: 1.00e+00 Pobj:  1.7784610e+01 Ad: 1.00e+00 Dobj:  1.7784627e+01
Iter: 18 Ap: 1.00e+00 Pobj:  1.7784626e+01 Ad: 1.00e+00 Dobj:  1.7784620e+01
Iter: 19 Ap: 1.00e-00 Pobj:  1.7784627e+01 Ad: 1.00e+00 Dobj:  1.7784627e+01
Iter: 20 Ap: 9.60e-01 Pobj:  1.7784627e+01 Ad: 9.60e-01 Dobj:  1.7784627e+01
Success: SDP solved
Primal objective value: 1.7784627e+01
Dual objective value: 1.7784627e+01
Relative primal infeasibility: 1.08e-09
Relative dual infeasibility: 3.12e-10
Real Relative Gap: -3.50e-10
XZ Relative Gap: 6.05e-11
DIMACS error measures: 1.08e-09 0.00e+00 7.02e-10 0.00e+00 -3.50e-10 6.05e-11
0.010u 0.000s 0:00.02 50.0%     0+0k 0+0io 134pf+0w
>> info


info =


     0
```

The **writesdpa** function can be used to write out a problem in SDPA sparse format.

```
%  This function takes a problem in SeDuMi MATLAB format and writes it out
%  in SDPA sparse format.
%
%  Usage:
%
%  ret=writesdpa(fname,A,b,c,K,pars)
%
%      fname           Name of SDPpack file, in quotes
%      A,b,c,K         Problem in SeDuMi form
%      pars            Optional parameters.
%                      pars.printlevel=0          No printed output
%                      pars.prinlevel=1 (default) Some printed output.
%                      pars.check=0 (default)     Do not check problem data
%                                                 for symmetry.
%                      pars.check=1               Check problem data for
```

```
%                                              symmetry.
%
%      ret               ret=0 on success, ret=1 on failure.
%
```

Problems in the SeDuMi format may involve "free" variables. A free variable
can be converted into the difference of two non–negative variables using the
convertf function.

```
%
%   [A,b,c,K]=convertf(A,b,c,K)
%
%   converts free variables in a SeDuMi problem into nonnegative LP variables.
%
```

Using the subroutine interface to CSDP

Storage Conventions

The matrices C, X, and Z are treated as block diagonal matrices. The dec-
larations in the file blockmat.h describe the block matrix data structure. The
`blockmatrix` structure contains a count of the number of blocks and a pointer
to an array of records that describe individual blocks. The individual blocks
can be matrices of size `blocksize` or diagonal matrices in which only a vector
of diagonal entries is stored.

Individual matrices within a block matrix are stored in column major order
as in Fortran. The ijtok() macro defined in index.h can be used to convert
Fortran style indices into an index into a C vector. For example, if A is stored
as a Fortran array with leading dimension n, element (i,j) of A can be accessed
within a C program as A[ijtok(i,j,n)].

The following table demonstrates how a 3 by 2 matrix would be stored under
this system.

C index	Fortran index
A[0]	A(1,1)
A[1]	A(2,1)
A[2]	A(3,1)
A[3]	A(1,2)
A[4]	A(2,2)
A[5]	A(3,2)

Vectors are stored as conventional C vectors. However, indexing always
starts with 1, so the [0] element of every vector is wasted. Most arguments
are described as being of size `n` or `m`. Since the zeroth element of the vector is
wasted, these vectors must actually be of size `n+1` or `m+1`.

The constraint matrices A_i are stored in a sparse form. The array **con-
straints** contains pointers which point to linked lists of structures, with one

structure for each block of the sparse matrix. The **sparseblock** data structures contain pointers to arrays which contain the entries and their **i** and **j** indices.

As an example of how to setup these data structures, the theta directory of the CSDP distribution contains an example program which sets up and solves an SDP to find the Lovasz ϑ number of a graph.

Storage Requirements

CSDP requires storage for a number of block diagonal matrices of the same form as X and Z, as well as storage for the Schur complement system that is Cholesky factored in each iteration. For a problem with m constraints and block diagonal matrices with blocks of size n_1, n_2, …, n_s, CSDP requires approximately

$$\text{Storage} = 8(m^2 + 11(n_1^2 + n_2^2 + \ldots + n_s^2))$$

bytes of storage. This formula includes all of the two dimensional arrays but leaves out the one dimensional vectors. This formula also excludes the storage required to store the constraint matrices, which are assumed to be sparse. In practice it is wise to allow for about 10% to 20% more storage to account for the excluded factors.

Calling The SDP Routine

A simplified interface to the SDP solver was provided in CSDP 3.x. The calling sequence for the solver remains unchanged in version 5.0, although the constraints data structure has changed. The routine has 11 parameters which include the problem data and an initial solution. The calling sequence for the sdp subroutine is:

```
int easy_sdp(n,k,C,a,constraints,constant_offset,pX,py,pZ,ppobj,pdobj)
   int n;                                   /* Dimension of X */
   int k;                                   /* # of constraints */
   struct blockmatrix C;                    /* C matrix */
   double *a;                               /* right hand side vector */
   struct constraintmatrix *constraints;    /* Constraints */
   double constant_offset;                  /* added to objective */
   struct blockmatrix *pX;                  /* X matrix */
   double **py;                             /* y vector */
   struct blockmatrix *pZ;                  /* Z matrix */
   double *ppobj;                           /* Primal objective */
   double *pdobj;                           /* Dual objective */
```

Input Parameters

1. **n.** This parameter gives the dimension of the X, C, and Z matrices.

2. **k.** This parameter gives the number of constraints.

3. `C`. This parameter gives the C matrix and implicitly defines the block structure of the block diagonal matrices.

4. `a`. This parameter gives the right hand side vector a.

5. `constraints`. This parameter specifies the problem constraints.

6. `constant_offset`. This scalar is added to the primal and dual objective values.

7. `pX`. On input, this parameter gives the initial primal solution X.

8. `py`. On input, this parameter gives the initial dual solution y.

9. `pZ`. On input, this parameter gives the initial dual solution Z.

Output Parameters

1. `pX`. On output this parameter gives the optimal primal solution X.

2. `py`. On output, this parameter gives the optimal dual solution y.

3. `pZ`. On output, this parameter gives the optimal dual solution Z.

4. `ppobj`. On output, this parameter gives the optimal primal objective value.

5. `pdobj`. On output, this parameter gives the optimal dual objective value.

Return Codes

If CSDP succeeds in solving the problem to full accuracy, the `easy_sdp` routine will return 0. Otherwise, the `easy_sdp` routine will return a nonzero return code. In many cases, CSDP will have actually found a good solution that doesn't quite satisfy one of the termination criteria. In particular, return code 3 is usually indicative of such a solution. Whenever there is a nonzero return code, you should examine the return and the solution to see what happened.

The nonzero return codes are

1. Success. The problem is primal infeasible.

2. Success. The problem is dual infeasible.

3. Partial Success. A solution has been found, but full accuracy was not achieved. One or more of primal infeasibility, dual infeasibility, or relative duality gap are larger than their tolerances, but by a factor of less than 1000.

4. Failure. Maximum iterations reached.

5. Failure. Stuck at edge of primal feasibility.

6. Failure. Stuck at edge of dual infeasibility.

7. Failure. Lack of progress.

8. Failure. X, Z, or O was singular.

9. Failure. Detected NaN or Inf values.

The User Exit Routine

By default, the `easy_sdp` routine stops when it has obtained a solution in which the relative primal and dual infeasibilities and the relative gap between the primal and dual objective values is less than 1.0×10^{-8}. There are situations in which you might want to terminate the solution process before an optimal solution has been found. For example, in a cutting plane routine, you might want to terminate the solution process as soon as a cutting plane has been found. If you would like to specify your own stopping criteria, you can implement these in a user exit routine.

At each iteration of its algorithm, CSDP calls a routine named `user_exit`. CSDP passes the problem data and current solution to this subroutine. If `user_exit` returns 0, then CSDP continues. However, if `user_exit` returns 1, then CSDP returns immediately to the calling program. The default routine supplied in the CSDP library simply returns 0. You can write your own routine and link it with your program in place of the default user exit routine.

The calling sequence for the user exit routine is

```
int user_exit(n,k,C,a,dobj,pobj,constant_offset,constraints,X,y,Z,params)
    int n;                                  /* Dimension of X */
    int k;                                  /* # of constraints */
    struct blockmatrix C;                   /* C matrix */
    double *a;                              /* right hand side */
    double dobj;                            /* dual objective  */
    double pobj;                            /* primal objective */
    double constant_offset;                 /* added to objective */
    struct constraintmatrix *constraints;   /* Constraints */
    struct blockmatrix X;                   /* primal solution */
    double *y;                              /* dual solution */
    struct blockmatrix Z;                   /* dual solution */
    struct paramstruc params;               /* parameters sdp called with */
```

Finding an Initial Solution

The CSDP library contains a routine for finding an initial solution to the SDP problem. Note that this routine allocates all storage required for the initial solution. The calling sequence for this routine is:

```
void initsoln(n,k,C,a,constraints,pX0,py0,pZ0)
    int n;                                  /* dimension of X */
```

```
   int k;                                /* # of constraints */
   struct blockmatrix C;                 /* C matrix */
   double *a;                            /* right hand side vector */
   struct constraintmatrix *constraints; /* constraints */
   struct blockmatrix *pX0;              /* Initial primal solution */
   double **py0;                         /* Initial dual solution */
   struct blockmatrix *pZ0;              /* Initial dual solution */
```

Reading and Writing Problem Data

The CSDP library contains routines for reading and writing SDP problems and solutions in SDPA format. The routine write_prob is used to write out an SDP problem in SDPA sparse format. The routine read_prob is used to read an SDP problem in from a file. The routine write_sol is used to write an SDP solution to a file. The routine read_sol is used to read a solution from a file.

The calling sequence for write_prob is

```
int write_prob(fname,n,k,C,a,constraints)
   char *fname;                           /* file to write */
   int n;                                 /* Dimension of X */
   int k;                                 /* # of constraints */
   struct blockmatrix C;                  /* The C matrix */
   double *a;                             /* The a vector */
   struct constraintmatrix *constraints;  /* the constraints */
```

The calling sequence for read_prob is:

```
int read_prob(fname,pn,pk,pC,pa,pconstraints,printlevel)
   char *fname;                 /* file to read  */
   int *pn;                     /* Dimension of X */
   int *pk;                     /* # of constraints */
   struct blockmatrix *pC;      /* The C matrix */
   double **pa;                 /* The a vector */
   struct constraintmatrix **pconstraints;  /* The constraints */
   int printlevel;              /* =0 for no output, =1 for normal
                                      output, >1 for debugging */
```

Note that the read_prob routine allocates all storage required by the problem.

The calling sequence for write_sol is

```
int write_sol(fname,n,k,X,y,Z)
   char *fname;                      /* Name of the file to write to */
   int n;                            /* Dimension of X */
   int k;                            /* # of constraints */
   struct blockmatrix X;             /* Primal solution X */
   double *y;                        /* Dual vector y */
   struct blockmatrix Z;             /* Dual matrix Z */
```

This routine returns 0 if successful and exits if it is unable to write the solution file.

The calling sequence for `read_sol` is

```
int read_sol(fname,n,k,C,pX,py,pZ)
    char *fname;                    /* file to read */
    int n;                          /* dimension of X */
    int k;                          /* # of constraints */
    struct blockmatrix C;           /* The C matrix */
    struct blockmatrix *pX;         /* The X matrix */
    double **py;                    /* The y vector */
    struct blockmatrix *pZ;         /* The Z matrix */
```

Note that `read_sol` allocates storage for X, y, and Z. This routine returns 0 when successful, and exits if it is unable to read the solution file.

Freeing Problem Memory

The routine `free_prob` can be used to automatically free the memory allocated for a problem. The calling sequence for `free_prob` is

```
void free_prob(n,k,C,a,constraints,X,y,Z)
    int n;                                  /* Dimension of X */
    int k;                                  /* # of constraints */
    struct blockmatrix C;                   /* The C matrix */
    double *a;                              /* The a vector */
    struct constraintmatrix *constraints;   /* the constraints */
    struct blockmatrix X;                   /* X matrix. */
    double *y;                              /* the y vector. */
    struct blockmatrix Z;                   /* Z matrix. */
```

Installing CSDP 5.0 from binaries

Precompiled binary versions of CSDP are available for a number of architectures. The binaries can be found at http://www.nmt.edu/∼borchers/csdp.html.

To install the binaries, simply copy the programs csdp, theta, rand_graph, complement, and graphtoprob from the bin directory into a directory in your search path, such as "/usr/local/bin".

Installing CSDP 5.0 from source

Follow these steps to install CSDP.

1. Obtain the latest CSDP source code. The CSDP source code and documentation can be found at http://www.nmt.edu/∼borchers/csdp.html. The source code for CSDP is available in two formats, a .tar archive

(for Unix) and a .zip archive (for Windows.) Under Unix, use "tar -xvf csdp5.0.tar" to expand the tar archive. Under Windows, use "pkunzip csdp50.zip" to expand the zip archive.

2. First, determine whether you'll find the BLAS and LAPACK library routines needed by CSDP. Many manufacturers provide optimized libraries of these routines (essl for IBMs, perflib for Suns, etc.) If you have one of these libraries, then you should probably use it. If not, then a package called ATLAS, available from http://www.netlib.org, provides all of the needed routines.

3. Modify the LIBS= lines in solver/Makefile and theta/Makefile to refer to your BLAS and LAPACK libraries. If the libraries are located in unusual places, you may have to modify the "LIBS=" line in the makefiles to specify where the libraries are located with the "-L" option.

4. In the csdp directory, issue the command "make all" to make the CSDP library, stand alone solver, and Lovasz ϑ codes.

5. Once you've built the code, you can test that everything worked:

    (a) Go to the theta/testprob directory and run ../theta g50. The file g50.out contains correct output for this problem.

    (b) Go to the solver/testprob directory and run ../csdp theta1.dat-s. The file theta1.out contains correct output for this problem.

6. In order to use the SDP routine with your own code, you'll want to copy the file libsdp.a from the lib directory to a directory where the compiler will be able to find it when you compile your code. You should also copy the csdp, theta, rand_graph, complement, and graphtoprob programs to a directory in your search path.

7. If you want to optimize the code, you can modify the "CFLAGS=" lines in the Makefiles under lib, solver, and theta. After adding compiler flags to optimize the code, rebuild CSDP with "make clean" followed by "make all".

8. The MATLAB interface assumes that the csdp executable is available in a directory within the search path of your shell. Make sure that csdp is working and installed before you try to use the MATLAB interface. You can simply copy the .m files from the matlab directory into your working directory, or you can install them in a system wide directory.

Important Installation Notes

1. Performance. The default makefiles don't include any compiler optimizations. Once you have tested the code, it would be a good idea to modify the "CFLAGS=" lines in the makefile to include the maximum possible

optimization. Then use "make clean" to clean out the unoptimized object code, and "make current" to rebuild the system.

2. Names of the BLAS/LAPACK routines. On most systems, the BLAS and LAPACK routines will have names which are in lower case and include an underscore. For example, "dgemm_" is the BLAS routine used for matrix multiplication. On some systems, these names may instead be capitalized or not have the underscore. The compilation flags -DCAPSBLAS, -DCAPSLAPACK, -DNOUNDERBLAS, and -DNOUNDERLAPACK can be used to configure the code to deal with these variations.

3. Use of short variables. By default, CSDP uses "unsigned short" variables for the indices array blocks. This saves some storage, but makes it impossible to solve problems with very large (dimensions more than 65,536) blocks. Use the "-DNOSHORTS" compilation flag to use regular "int" variables for these indices. With "-DNOSHORTS", CSDP can handle much larger problems.

4. On systems with an I32LP64 programming model, it is relatively simple to build CSDP in 64 bit mode. We assume that there are 64 bit BLAS/LAPACK routines available, and that these routines use 32 bit integer arguments for array dimensions. In building the code, use the compiler flag "-DBIT64" to turn on a few 64 bit specific bits of code in CSDP. CSDP has been successfully run in 64 bit mode under Sun Solaris, IBM's AIX, and under Linux on an AMD Opteron system.

5. Passing strings to Fortran subroutines. This is done in different ways on different systems. See your vendor's documentation on interfacing C and Fortran to learn what particular tricks are necessary on your system.

6. It is often necessary to include Fortran runtime libraries such as "-lf77" when linking to the BLAS and LAPACK libraries.

7. ATLAS. In addition to the BLAS routines, the ATLAS library contains all of the LAPACK routines that are used by CSDP. If you're using ATLAS, then be sure to include the lapack, f77blas, cblas, and atlas libraries when linking. In particular, use LIBS=-L../lib -lsdp -llapack -lf77blas -lcblas -latlas.

8. xerbla(). The BLAS error routine xerbla() may cause problems because it uses Fortran I/O, and the Fortran I/O libraries may not have been included in your C program. This can be fixed by explicitly including your system's Fortran I/O library. For example, on Linux machines using g77, adding -lg2c to the LIBS line of the makefile solves the problem. An alternative approach is simply to write your own dummy xerbla() routine which does nothing. The xerbla routine is never actually called during execution of CSDP, so this solves the problem.

Installation Experience

This software has been installed and tested on the following systems:

System	BLAS/LAPACK	Notes
MS Windows-ME	ATLAS	Used MINGW32 gcc 3.2 C compiler and utilities make, ar, cp,rm. CFLAGS=-march=pentium4 -O3 -ansi -pedantic -malign-double -mno-ieee-fp -ffast-math -funroll-loops -fomit-frame-pointer -DCAPSBLAS -DCAPSLAPACK -DNOUNDERBLAS -DNOUNDERLAPACK LIBS=-L../lib -lsdp -llapack -lf77blas -lcblas -latlas -lm
Linux/IA32	ATLAS	Used gcc 3.2.3, plus ATLAS libraries optimized for various CPU's. CFLAGS=-march=pentium4 -O3 -ansi -pedantic -malign-double -mno-ieee-fp -ffast-math -funroll-loops -fomit-frame-pointer LIBS=-L../lib -lsdp -llapack -lf77blas -lcblas -latlas -lg2c -lm
Mac OS X/G5	vecLib	CFLAGS=-O3 -ffast-math -fomit-frame-pointer -mcpu=G5 -faltivec LIBS=-L../lib -lsdp -framework vecLib

References

[1] B. Borchers. CSDP, a C library for semidefinite programming. *Optimization Methods & Software*, 11-2(1-4):613 – 623, 1999.

[2] K. Fujisawa, M. Kojima, K. Nakata, and M. Yamashita. SDPA (semidefinite programming algorithm) users manual - version 6.00. Technical Report B–308, Tokyo Institute of Technology, 1995.

[3] C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz. An interior-point method for semidefinite programming. *SIAM Journal on Optimization*, 6(2):342 – 361, May 1996.

[4] H. D. Mittelmann. An independent benchmarking of SDP and SOCP solvers. *Mathematical Programming*, 95(2):407 – 430, February 2003.

[5] J. F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods & Software*, 11-2(1-4):625 – 653, 1999.