CSDP 6.2.0 User's Guide

Brian Borchers

July 24, 2017

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 [**?**]. A more detailed, but now somewhat outdated description of the algorithms in CSDP can be found in [**?**]. CSDP is written in C for efficiency and portability. On systems with multiple processors and shared memory, CSDP can run in parallel. CSDP uses OpenMP directives in the C source code to tell the compiler how to parallelize various loops. The parallel implementation is described in [**?**]. 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 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 has been used within a branch and bound code for maximum independent set problems to terminate the bounding calculations as soon as a bound has been obtained that is good enough to fathom the current note. The library also 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 [**?**]. An interface to MATLAB and the open source MATLAB clone Octave is also provided. This interface can be used to solve problems that are in the format used by the SeDuMi [**?**].

This document describes how to use the stand alone solver, MATLAB and Octave interface, and library routines. For detailed instructions on how to compile and install CSDP see the INSTALL file in the main directory.

The SDP Problem

CSDP solves semidefinite programming problems of the form

$$\begin{array}{rrcl} \max & \text{tr}\,(CX) & & \\ & A(X) & = & a \\ & X & \succeq & 0 \end{array} \tag{1}$$

where

$$A(X) = \begin{bmatrix} \text{tr}\,(A_1X) \\ \text{tr}\,(A_2X) \\ \cdots \\ \text{tr}\,(A_mX) \end{bmatrix}. \tag{2}$$

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

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

where

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

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

$$\begin{array}{rcl} \frac{\text{tr}(XZ)}{1+|a^Ty|+|\text{tr}(CX)|} & < & 1.0 \times 10^{-8} \\ \frac{\|A(x)-a\|_2}{1+\|a\|_2} & < & 1.0 \times 10^{-8} \\ \frac{\|A^T(y)-C-Z\|_F}{1+\|C\|_F} & < & 1.0 \times 10^{-8} \\ X, Z & \succeq & 0. \end{array} \tag{5}$$

Note that for feasible primal and dual solutions, $a^Ty - \text{tr}(CX) = \text{tr}(XZ)$. 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 $\text{tr}(XZ)$. In some cases, the duality gap may even become negative. Because of this ambiguity, we use the $\text{tr}(XZ)$ 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 $\text{tr}(XZ)$ 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 [**?**]. For convenience in benchmarking, CSDP includes these DIMACS error measures in its output.

To test for primal infeasibility, CSDP checks the inequality

$$\frac{-a^T y}{\|A^T(y) - Z\|_F} > 1.0 \times 10^8. \tag{6}$$

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

$$\frac{\mathrm{tr}(CX)}{\|A(X)\|_2} > 1.0 \times 10^8. \tag{7}$$

If CSDP detects that a problem is dual infeasible, it announces this in its output and returns a primal solution with $\mathrm{tr}(CX) = 1$, 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
CSDP 6.2.0
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: 5.55e-17
Relative dual infeasibility: 3.93e-09
Real Relative Gap: 7.21e-09
XZ Relative Gap: 7.82e-09
DIMACS error measures: 5.55e-17 0.00e+00 1.00e-07 0.00e+00 7.21e-09 7.82e-09
Elements time: 0.001091
Factor time: 0.000620
Other time: 0.016636
Total time: 0.018348
```

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.

The last four lines give the time in seconds used by various steps in the algorithm. The first line, "Elements" shows the time spent in constructing the Schur complement matrix. The second line, "Factor" shows the time spent in factoring the Schur complement matrix. The third line, "Other" shows the time spent in all other operations. The fourth line gives the total time used in solving the problem. Note that the times given here are "wall clock" times, not CPU time. On a system that is running other programs, the wall clock time may be considerably larger than the CPU time. On multiprocessor systems, the wall clock time will not include all of the CPU time used by the different processors. The reported time will typically vary on repeated runs of CSDP, particularly for small problems like the one solved here.

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 **pinftol** 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 $\mathrm{tr}(XZ)$ 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. If **perturbobj** is 0, then the objective will not be perturbed. If **perturbobj** is 1, then the objective function will be perturbed by a default amount. Larger values of **perturbobj** (e.g. 100.0) increase the size of the perturbation. This can be helpful in solving some difficult problems. 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 somewhat faster, but also somewhat less accurate.

Calling CSDP from MATLAB or Octave

An interface to the stand alone solver from MATLAB or Octave is included in CSDP. This requires MATLAB 6.5 or later) or Octave 2.9.5 or later. The 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,x0,y0,z0)
%
% Uses CSDP to solve a problem in SeDuMi format.
%
% Input:
%        At, b, c, K      SDP problem in SeDuMi format.
%        pars             CSDP parameters (optional parameter.)
%        x0,y0,z0         Optional starting point.
%
% 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 or read back the solution.
%
```

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     Attributes

  At        125x21            11376  double    sparse
  K           1x1               192  struct
  ans         2x1                16  double
  b          21x1               168  double
  c         125x1               112  double    sparse

>> pars.objtol=1.0e-9

pars =

    objtol: 1.0000e-09

>> [x,y,z,info]=csdp(At,b,c,K);
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:  1.7784632e+01
Iter: 18 Ap: 1.00e+00 Pobj:  1.7784626e+01 Ad: 1.00e+00 Dobj:  1.7784627e+01
Iter: 19 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: 2.06e-09
Relative dual infeasibility: 7.31e-10
Real Relative Gap: 7.13e-10
XZ Relative Gap: 1.51e-09
DIMACS error measures: 2.06e-09 0.00e+00 1.65e-09 0.00e+00 7.13e-10 1.51e-09
Elements time: 0.027664
Factor time: 0.011323
Other time: 1.187727
Total time: 1.226715
3.416u 0.015s 0:01.43 239.1% 0+0k 7688+16io 24pf+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 SDPA file, in quotes
%      A,b,c,K         Problem in SeDuMi form
%      pars            Optional parameters.
%                      pars.printlevel=0           No printed output
%                      pars.printlevel=1 (default) Some printed output.
%
%      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 declarations 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 **constraints** 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.

For an example of how to setup these data structures, refer to the example directory in the CSDP distribution. This directory contains a program that solves the very small SDP

$$\begin{array}{rrcl} \max & \text{tr}\,(CX) & & \\ & \text{tr}\,A_1X & = & 1 \\ & \text{tr}\,A_2X & = & 2 \\ & X & \succeq & 0 \end{array} \tag{8}$$

where

$$C = \begin{bmatrix} 2 & 1 & & & & & \\ 1 & 2 & & & & & \\ & & 3 & 0 & 1 & & \\ & & 0 & 2 & 0 & & \\ & & 1 & 0 & 3 & & \\ & & & & & 0 & \\ & & & & & & 0 \end{bmatrix} \tag{9}$$

$$A_1 = \begin{bmatrix} 3 & 1 & & & & & \\ 1 & 3 & & & & & \\ & & 0 & 0 & 0 & & \\ & & 0 & 0 & 0 & & \\ & & 0 & 0 & 0 & & \\ & & & & & 1 & \\ & & & & & & 0 \end{bmatrix} \tag{10}$$

$$A_2 = \begin{bmatrix} 0 & 0 & & & & & \\ 0 & 0 & & & & & \\ & & 3 & 0 & 1 & & \\ & & 0 & 4 & 0 & & \\ & & 1 & 0 & 5 & & \\ & & & & & 0 & \\ & & & & & & 1 \end{bmatrix}. \tag{11}$$

In this problem, the X, Z, A_1, A_2 and C matrices have three blocks. The first block is a 2 by 2 matrix. The second block is a 3 by 3 matrix. The third block is a diagonal block with 2 entries.

In addition to setting up and solving this problem, the example program calls the write_prob() routine to produce a file containing the SDP problem in SDPA sparse format. This is stored in the file prob.dat-s.

```
2
3
2 3 -2
1.00000000000000000e+00 2.00000000000000000e+00
0 1 1 1 2.00000000000000000e+00
0 1 1 2 1.00000000000000000e+00
0 1 2 2 2.00000000000000000e+00
0 2 1 1 3.00000000000000000e+00
0 2 1 3 1.00000000000000000e+00
0 2 2 2 2.00000000000000000e+00
0 2 3 3 3.00000000000000000e+00
1 1 1 1 3.00000000000000000e+00
1 1 1 2 1.00000000000000000e+00
1 1 2 2 3.00000000000000000e+00
1 3 1 1 1.00000000000000000e+00
2 2 1 1 3.00000000000000000e+00
2 2 2 2 4.00000000000000000e+00
2 2 3 3 5.00000000000000000e+00
2 2 1 3 1.00000000000000000e+00
2 3 2 2 1.00000000000000000e+00
```

The 2 in the first line indicates that this problem has two constraints. The 3 in the second line indicates that there are three blocks in the X and Z matrices. The third line gives the sizes of the three blocks. Note that the third block's size is given as -2. The minus sign indicates that this is a diagonal block. The fourth line gives the values of the right hand sides of the two constraints.

The remaining lines in the file describe the entries in the C, A_1, and A_2 matrices. The first number in each line is the number of the matrix, with 0 for the C matrix. The second number specifies a block within the matrix. The third and fourth numbers give the row and column of a nonzero entry within this block. The fifth number gives the actual value at that position within the

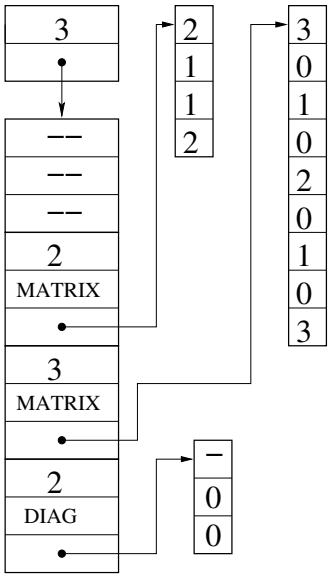


Figure 1: The C matrix.

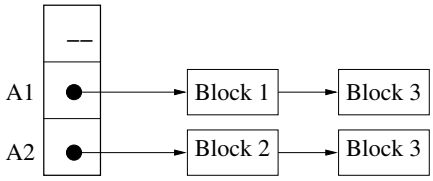


Figure 2: The constraints.

block. Comparing this file to the problem statement above can be helpful in understanding the SDPA sparse file format.

Figure 1 shows a graphical representation of the data structure that holds the C matrix. Notice that individual matrix blocks of the C matrix are stored as Fortran arrays and that the diagonal block is stored as a vector, with the 0 entry unused. The data structures for X and Z are similar.

Figure 2 shows the overall structure of the constraints. There is a vector of pointers to linked lists of constraint blocks. The 0th entry in this array is ignored. Blocks that contain only zero entries are not stored in the linked lists. Figure 3 shows the detail of the data structure for block 1 of the constraint matrix A_1.

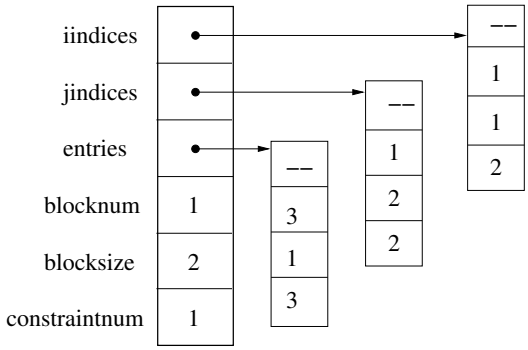


Figure 3: Block 1 of A_1.

After solving the problem, the example program outputs the solution to prob.sol using the write_sol() routine. The output produced on different computers might vary because of floating point round–off differences. However, the following output is typical.

```
7.49999999674811235e-01 9.99999995736339464e-01
1 1 1 1 2.50000018710683558e-01
1 1 1 2 -2.50000000325189320e-01
1 1 2 2 2.50000018710683558e-01
1 2 1 1 6.89527285114916582 7e-10
1 2 1 3 -4.26366025129774837 6e-10
1 2 2 2 2.00000000263161049e+00
1 2 3 3 1.99999999836795217e+00
1 3 1 1 7.50000019361059422e-01
1 3 2 2 1.00000001542258765e+00
2 1 1 1 1.25000001467082567e-01
2 1 1 2 1.24999992664581755e-01
2 1 2 2 1.25000001467082567e-01
2 2 1 1 6.66666970820890570e-01
2 2 1 3 -4.51833481144514214 7e-07
2 2 2 2 2.20062933863723688 3e-10
2 2 3 3 2.20063510893323199 8e-10
2 3 1 1 5.86834155603549469 9e-10
2 3 2 2 4.40125847850854104 7e-10
```

The first line of the file gives the optimal y values. The lines that start with "1 " give the nonzero entries in the optimal Z matrix. As in the SDPA input file there are five numbers per line. The first number is the number of the matrix, where 1 is used for Z and 2 is used for X. The second number specifies a block within the matrix. The third and fourth numbers are the row and column within the block. The final number is the actual value at the position in the block. For example,

```
2 2 1 3 -4.51833481144514214 7e-07
```

means that in the 1st row, third column of block 2 of the X matrix, the entry is $-4.518334811445142147 \times 10^{-7}$. Since the matrices are symmetric, we only record the entries in the upper triangle of the matrix. The same entry will also appear in row 3, column 1.

So, the optimal solution to the example problem is (rounding the numbers off to two or three digits)

$$y = \left[\begin{array}{cc} 0.75 & 1.00 \end{array} \right] \tag{12}$$

$$Z = \begin{bmatrix} 0.25 & -0.25 & & & & & & \\ -0.25 & 0.25 & & & & & & \\ & & 0 & 0 & 0 & & & \\ & & 0 & 2.00 & 0 & & & \\ & & 0 & 0 & 2.00 & & & \\ & & & & & 0.75 & & \\ & & & & & & 1.00 & \end{bmatrix} \tag{13}$$

$$X = \begin{bmatrix} 0.125 & 0.125 & & & & & & \\ 0.125 & 0.125 & & & & & & \\ & & 0.667 & 0 & 0 & & & \\ & & 0 & 0 & 0 & & & \\ & & 0 & 0 & 0 & & & \\ & & & & & 0 & & \\ & & & & & & 0 & \end{bmatrix} \tag{14}$$

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

$$\text{Storage} = 8(m^2 + 11(n_1^2 + n_2^2 + \ldots + n_s^2)) \tag{15}$$

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.

The parallel version of CSDP requires additional storage for work matrices used by the routine that computes the Schur complement matrix. If the OpenMP maximum number of threads (typically the number of processors on the system) is p, and $p > 1$, then CSDP will allocate an additional $16(p-1)n_{\max}^2$ bytes of storage for workspace.

Calling The SDP Routine

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	Explanation
0	Problem solved to optimality
1	Problem is primal infeasible.
2	Problem is dual infeasible.
3	Problem solved to near optimality
4	Maximum iterations reached.
5	Stuck at edge of primal feasibility.
6	Stuck at edge of dual feasibility.
7	Lack of progress.
8	X, Z, or O is singular.
9	NaN or Inf values encountered.
10	Program stopped by signal (SIXCPU, SIGTERM, etc.)

Table 1: Return codes for easy_sdp() and CSDP.

Code	Explanation
200	No problem data file named.
201	Couldn't open problem data file.
202	Couldn't open initial solution data file.
203	Couldn't write problem data file.
204	Couldn't write solution data file.
205	Storage allocation failed.
206	An internal error occurred.

Table 2: Exit codes for non-recoverable errors.

code, you should examine the return and the solution to see what happened. In other cases, the code will stop because it can no longer make progress and the solution doesn't satisfy the required tolerances. However, the solution may be good enough for your purposes.

The return codes from the easy_sdp() routine are given in Table 1. For the command line solver, csdp, these are also used as exit codes. Additional exit codes are used for significant errors that did not allow execution. These are given Table 2.

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. If CSDP is compiled with the "-DUSESIGTERM" flag, then default routine will also stop the solution process whenever the process receives a TERM signal. 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 bqlockmatrix 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. */
```