CSDP 2.3 User's Guide

Brian Borchers

April 22, 1998.

Introduction

CSDP is a software package for solving semidefinite programming problems. CSDP is written in C for efficiency and portability. The code is designed to make use of highly optimized linear algebra routines from the LINPACK or LAPACK libraries. The package includes unoptimized versions of the LINPACK routines that have been translated into C. If optimized versions of these routines are available, they can be used in place of the routines supplied with CSDP for better performance.

CSDP also has a number of features that make it very flexible. 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. CSDP can handle linear inequality constraints as well as linear equality constraints. 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. In addition to its SDP solver, the CSDP library contains routines for reading and writing SDP 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.

This document describes how to install CSDP and how to use the stand alone solver and library routines after CSDP has been installed. A companion paper gives a detailed description of the algorithm used by CSDP, discusses its computational complexity, and presents some computational results from a set of test problems.

The SDP Problem

CSDP solves semidefinite programming problems of the form

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

where

$$A(X) = \begin{bmatrix} \text{tr}\,(A_1X) \\ \text{tr}\,(A_2X) \\ \dots \\ \text{tr}\,(A_kX) \end{bmatrix} \tag{2}$$

and

$$B(X) = \begin{bmatrix} \text{tr}\,(B_1X) \\ \text{tr}\,(B_2X) \\ \dots \\ \text{tr}\,(B_lX) \end{bmatrix}. \tag{3}$$

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

The dual of this SDP is

$$\begin{array}{lrcl} \min & a^Ty + b^Tt & & \\ & A^T(y) + B^T(t) - C & = & Z \\ & Z & \succeq & 0 \\ & t & \geq & 0 \end{array} \tag{4}$$

where

$$A^T(y) = \sum_{i=1}^{k} y_iA_i \tag{5}$$

and

$$B^T(t) = \sum_{i=1}^{l} t_iB_i. \tag{6}$$

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: 1.000e+00 Pobj: 1.464466e+05 Ad: 1.000e+00 Dobj: 0.000000e+00
Iter: 1 Ap: 8.100e-01 Pobj: 2.777776e+04 Ad: 1.000e+00 Dobj: 1.805108e+03
Iter: 2 Ap: 8.100e-01 Pobj: 5.283552e+03 Ad: 1.000e+00 Dobj: 2.030492e+03
Iter: 3 Ap: 9.000e-01 Pobj: 5.297482e+02 Ad: 1.000e+00 Dobj: 2.283411e+03
Iter: 4 Ap: 9.000e-01 Pobj: 5.388389e+01 Ad: 1.000e+00 Dobj: 2.560102e+03
Iter: 5 Ap: 9.000e-01 Pobj: 6.303050e+00 Ad: 1.000e+00 Dobj: 2.785857e+03
Iter: 6 Ap: 1.000e+00 Pobj: 1.019541e+00 Ad: 1.000e+00 Dobj: 2.318478e+03
Iter: 7 Ap: 1.000e+00 Pobj: 1.057472e+00 Ad: 1.000e+00 Dobj: 2.907398e+02
Iter: 8 Ap: 1.000e+00 Pobj: 1.456330e+00 Ad: 8.100e-01 Dobj: 8.575052e+01
Iter: 9 Ap: 1.000e+00 Pobj: 4.878950e+00 Ad: 5.905e-01 Dobj: 4.421853e+01
Iter: 10 Ap: 8.100e-01 Pobj: 1.566597e+01 Ad: 8.100e-01 Dobj: 3.110675e+01
Iter: 11 Ap: 9.000e-01 Pobj: 1.992626e+01 Ad: 9.000e-01 Dobj: 2.494451e+01
Iter: 12 Ap: 8.100e-01 Pobj: 2.159309e+01 Ad: 9.000e-01 Dobj: 2.340923e+01
Iter: 13 Ap: 9.000e-01 Pobj: 2.254956e+01 Ad: 1.000e+00 Dobj: 2.310986e+01
Iter: 14 Ap: 9.000e-01 Pobj: 2.290146e+01 Ad: 9.000e-01 Dobj: 2.302053e+01
Iter: 15 Ap: 1.000e+00 Pobj: 2.297471e+01 Ad: 9.000e-01 Dobj: 2.300609e+01
Iter: 16 Ap: 1.000e+00 Pobj: 2.299664e+01 Ad: 9.000e-01 Dobj: 2.300111e+01
Iter: 17 Ap: 1.000e+00 Pobj: 2.299952e+01 Ad: 9.000e-01 Dobj: 2.300018e+01
Iter: 18 Ap: 1.000e+00 Pobj: 2.299993e+01 Ad: 9.000e-01 Dobj: 2.300003e+01
Iter: 19 Ap: 1.000e+00 Pobj: 2.299999e+01 Ad: 9.000e-01 Dobj: 2.300000e+01
Iter: 20 Ap: 1.000e+00 Pobj: 2.300000e+01 Ad: 9.000e-01 Dobj: 2.300000e+01
SDP Solved.
Primal objective value: 2.2999998e+01
Dual objective value: 2.3000001e+01
Gap: 2.3148370e-06
Relative Gap: 9.6451546e-08
Relative primal infeasibility: 6.1071362e-16
Relative dual infeasibility: 6.2532081e-17
>
```

One line of output appears for each iteration of the algorithm, giving the iteration number, primal stepsize (Ap), primal objective value (Pobj), dual stepsize (Ad), and dual objective value (Dobj). The last four lines of output show the primal and dual optimal objective values and the relative primal and dual infeasibility in the optimal solution.

Converting problems to/from SDPA format

The CSDP package includes two programs for converting problems from SDPA sparse format to the format used by SDPpack, and from the SDPpack format to SDPA sparse format. The sdpatosqlp program converts from SDPA sparse format to SDPpack format. Usage is:

```
sdpatosqlp <sdpa file> <sqlp file>
```

The sqlptosdpa program converts from SDPpack format to SDPA sparse format. Usage is:

```
sqlptosdpa <sqlp file> <sdpa file>
```

Note that sqlptosdpa converts the linear and quadratic cone constraints in the SDPpack format into semidefinite programming constraints. This conversion can be quite inefficient for problems with many variables involved in quadratic cone constraints.

Using the subroutine interface to CSDP

Storage Conventions

All matrices 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 ldan, element (i,j) of A can be accessed within a C program as A[ijtok(i,j,ldan)].

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)

All arrays in the SDP routine are either of size `ldan` or `ldam`. The parameter `ldan` is used to specify the leading dimension of arrays such as X, C, and Z which hold arrays of size n. The parameter `ldam` is used to specify the leading dimension of the array O, which holds the m by m system matrix.

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 `ldan` or `ldam`. Since the zeroth element of the vector is wasted, these vectors must actually be of size `ldan+1` or `ldam+1`.

It is important to initialize all arrays and vectors used by the SDP routine to valid floating point numbers. This is typically done by using `memset` to initialize the arrays and vectors to 0. See the source code for the csdp solver for an example. If elements of these arrays are not properly initialized and happen to be NaN's, then the results of subsequent computations will also be NaN's, even though these NaN's are multiplied by zero. This is particularly an issue on Alpha machines, where the operating system initializes memory to NaN's.

Calling The SDP Routine

```
int sdp(n,k,l,ldam,ldan,bandwidth,nblocks,block_structure,C,a,b,
        constant_offset,start_a,ai,aj,aent,start_b,bi,bj,bent,X,y,t,Z,
        pobj,dobj,workn1,workn2,workn3,workvec1,Zi,O,rhs,dZ,dZ1,dX,dX1,
        dy,dy1,dt,dt1,Fp,Fd,printlevel)
     int n;          /* Dimension of X */
     int k;          /* Number of equality constraints.  */
     int l;          /* Number of inequality constraints. (m=k+l) */
     int ldam;       /* leading dimension for arrays of size m. */
     int ldan;       /* leading dimension for arrays of size n. */
     int bandwidth;  /* Bandwidth of X, Z, C, and A matrices. */
     int nblocks;    /* Number of blocks in X, Z, C, and A matrices. */
     int *block_structure;    /* Sizes of blocks in X, Z, C, and A matrices. */
     double *C;      /* C matrix (n by n), in column major order. */
     double *a;      /* The a vector. */
     double *b;      /* The b vector. */
     double constant_offset;  /* Constant added to objective value */
     int *start_a;   /* start_a[i] is the index in a,ai,aj of the
                        start of array A_i, */
     int *ai;        /* i indices of elements of the A_i's. */
     int *aj;        /* j indices of elements of the A_i's. */
     double *aent;   /* entries in the A_i's. */
     int *start_b;   /* start_b[i] is the index in b,bi,bj of the
                        start of array B_i */
     int *bi;        /* i indices of elements of the B_i's. */
     int *bj;        /* j indices of elements of the B_i's. */
     double *bent;   /* entries in the B_i's. */
     double *X;      /* X matrix, in column major order. */
     double *y;      /* y vector. */
     double *t;      /* t vector. */
     double *Z;      /* Z matrix, in column major order. */
double *pobj;  /* primal objective. */
double *dobj;  /* dual objective. */
double *workn1; /* first work array of size ldan */
double *workn2; /* second work array of size ldan */
double *workn3; /* third work array of size ldan */
double *workvec1; /* Work vector of size > max(ldam,ldan) */
double *Zi;     /* Z inverse. of size ldan */
double *O;      /* System matrix, of size ldam */
double *rhs;    /* right hand side for the system equations. size ldam */
double *dZ;     /* dZ matrix, size ldan */
double *dZ1;    /* dZ1 matrix, size ldan */
double *dX;     /* dX matrix, size ldan */
double *dX1;    /* dX1 matrix, size ldan */
double *dy;     /* dy vector, size ldam */
double *dy1;    /* dy1 vector, size ldam */
double *dt;     /* dt vector, size ldam */
double *dt1;    /* dt1 vector, size ldam */
double *Fp;     /* A(x)-a residual vector, of size ldam */
double *Fd;     /* C-A'(y)-B'(t)-Z residual matrix, size ldan */
int printlevel; /* 0=print nothing, 1=one line per iter, 2=debugging info */
```

Input Parameters

1. `n`. This parameter defines the dimensions of the matrices `X`, `C`, `Z`, etc.

2. `k`. This parameter defines the number of equality constraints.

3. `l`. This parameter defines the number of inequality constraints.

4. `ldan`. This parameter defines the leading dimension for the Fortran style arrays of size n.

5. `ldam`. This parameter defines the leading dimension for the Fortran style array `O`.

6. `bandwidth`. This parameter defines the size of the largest block in the A, C, Z, and X matrices. If the matrices have no block structure, use n.

7. `nblocks`. This parameter defines the number of blocks in the block diagonal structure of A, C, Z, and X. If the matrices have no block structure, simply use 1.

8. `block_structure`. This array gives the sizes of the individual blocks in A, C, Z, and X. If the matrix has no block structure, simply let `block_structure[1]=n`.

9. `C`. This Fortran style array contains the objective function coefficients.

10. `a`. This vector contains the right hand side coefficients for the equality constraints. Note that these are stored starting with `a[1]`, not `a[0]`.

11. `b`. This vector contains the right hand side coefficients for the inequality constraints. Note that these are stored starting with `b[1]`, not `b[0]`.

12. `constant_offset`. This parameter is added to the primal and dual objective function values.

13. `start_a`. This vector specifies the location of array A_i in `ai`, `aj`, and `aent` vectors. In particular, the entries of array A_i will be located in `aent[start_a[i]], ..., aent[start_a[i+1]-1]`.

14. `ai, aj, aent`. These vectors contain the entries in the constraint matrices A_i. Note that only the entries in the upper right triangle of the matrix are included- the lower left entries are assumed to be symmetric. For example, if A_1 contains the entries $A_{15} = 0.5$, $A_{23} = 0.6$, then `ai[1]=1, aj[1]=5, aent[1]=0.5, ai[2]=2, aj[2]=3, aent[2]=0.6`.

15. `start_b, bi, bj, bent`. Entries of the B_i matrices, stored in the same fashion as the A_i matrices.

16. `X`. Starting value for X, stored as a Fortran style array with leading dimension ldan. X must be positive definite, but it is not necessary that $A(X) = a$.

17. `y`. Starting value for y.

18. `t`. Starting value for t. It is assumed that $t > 0$.

19. `Z`. Starting value for Z, stored as a Fortran style array with leading dimension `ldan`. Z must be positive definite.

20. `printlevel`. This parameter determines how much output is produced by CSDP. Use `printlevel=0` for no output, `printlevel=1` for normal output, and `printlevel=2` for debugging output.

Output Parameters

1. `pobj`. Primal objective value.

2. `dobj`. Dual objective value.

3. `X`. The optimal primal solution, X.

4. `y`. The optimal dual multipliers y.

5. `t`. The optimal dual multipliers t.

6. `Z`. The optimal dual slacks Z.

Return Codes

If CSDP succeeds in solving the problem, the `sdp` routine will return 0. If CSDP fails for some reason, the `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 6 is usually indicative of such a solution. The nonzero return codes are

1. Failure: $||X||_F$ is very large. This suggests the possibility of dual infeasibility.
2. Failure: $||Z||_F$ is very large. This suggests the possibility of primal infeasibility.
3. Failure: maximum iterations reached.
4. Failure: invalid parameters were supplied.
5. Failure: initial solution was infeasible.
6. Failure: X, Z, or O became singular.
7. Failure: primal or dual line search failed.

Work Arrays

1. `workn1, workn2, workn3, Zi, dZ, dZ1, dX, dX1, Fd`. Work arrays of size `ldan` by `ldan`.
2. `O`. Work array of size `ldam` by `ldam`.
3. `workvec1, rhs, dy, dy1, dt, dt1, Fp`. Work vectors all should be of size `ldam+1`.

The User Exit Routine

By default, the `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^{-7}. 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,l,ldan,C,a,b,dobj,pobj,constant_offset,
      start_a,ai,aj,aent,start_b,bi,bj,bent,X,y,t,Z)
     int n;           /* Dimension of X */
     int k;           /* Number of equality constraints.  */
     int l;           /* Number of inequality constraints. (m=k+l) */
     int ldan;        /* leading dimension for arrays of size n. */
     double *C;       /* C matrix (n by n), in column major order. */
     double *a;       /* The a vector. */
     double *b;       /* The b vector. */
     double dobj;     /* dual objective. */
     double pobj;     /* primal objective. */
     double constant_offset; /* Constant added to objective value */
     int *start_a;    /* start_a[i] is the index in a,ai,aj of the
                         start of array A_i, */
     int *ai;         /* i indices of elements of the A_i's. */
     int *aj;         /* j indices of elements of the A_i's. */
     double *aent;    /* entries in the A_i's. */
     int *start_b;    /* start_b[i] is the index in b,bi,bj of the
                         start of array B_i */
     int *bi;         /* i indices of elements of the B_i's. */
     int *bj;         /* j indices of elements of the B_i's. */
     double *bent;    /* entries in the B_i's. */
     double *X;       /* X matrix, in column major order. */
     double *y;       /* y vector. */
     double *t;       /* t vector. */
     double *Z;       /* Z matrix, in column major order. */
```

Customized Routines for the Operators $A()$ and $B()$

In problems with special structure, the operators $A()$, $A^T()$, $B()$, and $B^T()$ may be considerably simplified. For example, if the constraints are $X_{i,i} = 1$, $i = 1 \ldots n$, then $A(X) = \mathrm{diag}(X)$ and $A^T(y) = \mathrm{diag}(y)$. You can improve the performance of CSDP on such problems by implementing your own routines to compute these operators.

The calling sequence for the routine `op_a` is

```
void op_a(n,k,lda,X,start_a,ai,aj,aent,result)
     int n;           /* Dimension of X */
     int k;           /* Number of equality constraints.  */
     int lda;         /* leading dimension for arrays of size n. */
     double *X;       /* the argument of A(), in column major order. */
     int *start_a;   /* start_a[i] is the index in a,ai,aj of the
                        start of array A_i, */
     int *ai;        /* i indices of elements of the A_i's. */
     int *aj;        /* j indices of elements of the A_i's. */
     double *aent;   /* entries in the A_i's. */
     double *result; /* The result, A(X) */
```

The input parameters are `n`, `k`, `l`, `lda`, `X`, `start_a`, `ai`, `aj`, and `aent`. The result is returned in the vector `result`.

The calling sequence for the routine `op_at` is

```
void op_at(n,k,ldan,y,start_a,ai,aj,aent,result,nblocks,block_structure)
     int n;          /* Dimension of X */
     int k;          /* Number of equality constraints.  */
     int ldan;       /* leading dimension for arrays of size n. */
     double *y;      /* The argument vector */
     int *start_a;   /* start_a[i] is the index in a,ai,aj of the
                        start of array A_i, */
     int *ai;        /* i indices of elements of the A_i's. */
     int *aj;        /* j indices of elements of the A_i's. */
     double *aent;   /* entries in the A_i's. */
     double *result; /* The result, a matrix in column order */
     int nblocks;   /* Number of blocks in X, Z, C, and A matrices. */
     int *block_structure;  /* Sizes of blocks in X, Z, C, and A matrices. */
```

The input parameters are `n`, `k`, `l`, `ldan`, `y`, `start_a`, `ai`, `aj`, `aent`, `nblocks`, and `block_structure`. The result is returned in the matrix `result`.

The calling sequences for `op_b` and `op_bt` are identical to the calling sequences for `op_a` and `op_at`.

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_size` is used to determine the size of problem in SDPA format. After storage has been allocated, 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

```
void write_prob(fname,n,k,l,lda,nblocks,block_structure,C,a,b,
                start_a,ai,aj,aent,start_b,bi,bj,bent,workn1,workn2)
     char *fname;    /* Name of the file to write to */
     int n;          /* Dimension of X */
     int k;           /* Number of equality constraints.  */
     int l;           /* Number of inequality constraints. (m=k+l) */
     int lda;         /* leading dimension for arrays of size n. */
     int nblocks;     /* Number of blocks in A, C, X, and Z. */
     int *block_structure /* Block Structure of A, C, X, and Z. */
     double *C;       /* C matrix (n by n), in column major order. */
     double *a;       /* The a vector. */
     double *b;       /* The b vector. */
     int *start_a;    /* start_a[i] is the index in a,ai,aj of the
                         start of array A_i, */
     int *ai;         /* i indices of elements of the A_i's. */
     int *aj;         /* j indices of elements of the A_i's. */
     double *aent;    /* entries in the A_i's. */
     int *start_b;    /* start_b[i] is the index in b,bi,bj of the
                         start of array B_i */
     int *bi;         /* i indices of elements of the B_i's. */
     int *bj;         /* j indices of elements of the B_i's. */
     double *bent;    /* entries in the B_i's. */
     int *work1;      /* workvector of size at least n+2 */
     int *work2;      /* workvector of size at least n+2 */
```

The calling sequence for `read_size` is

```
int read_size(fname,pn,pk,nonza,nblocks,block_structure,bandwidth)
     char *fname;         /* Name of file to read. */
     int *pn;             /* n, the size of the A, C, X, Z matrices. */
     int *pk;             /* k, the number of equality constraints. */
     int *nonza;          /* total number of nonzero entries in A_i matrices. */
     int *nblocks;        /* The number of blocks in the A, C, X, Z matrices. */
     int *block_structure; /* Sizes of indvidual blocks. */
     int *bandwidth;        /* Size of the largest block. */
```

The calling sequence for `read_prob` is

```
void read_prob(fname,pn,pk,pl,lda,nblocks,block_structure,C,a,b,
               start_a,ai,aj,aent,start_b,bi,bj,bent)
     char *fname;     /* Name of the file to read from */
     int *pn;         /* Dimension of X */
     int *pk;         /* Number of equality constraints.  */
     int *pl;         /* Number of inequality constraints. (m=k+l) */
     int lda;         /* leading dimension for arrays of size n. */
     int nblocks;     /* Number of blocks in A, C, X, and Z. */
     int *block_structure /* Block Structure of A, C, X, and Z. */
     double *C;       /* C matrix (n by n), in column major order. */
     double *a;       /* The a vector. */
     double *b;       /* The b vector. */
     int *start_a;    /* start_a[i] is the index in a,ai,aj of the
                         start of array A_i, */
     int *ai;         /* i indices of elements of the A_i's. */
     int *aj;         /* j indices of elements of the A_i's. */
     double *aent;    /* entries in the A_i's. */
     int *start_b;    /* start_b[i] is the index in b,bi,bj of the
                         start of array B_i */
     int *bi;         /* i indices of elements of the B_i's. */
     int *bj;         /* j indices of elements of the B_i's. */
     double *bent;    /* entries in the B_i's. */
```

The calling sequence for `write_sol` is

```
void write_sol(fname,n,k,l,lda,nblocks,block_structure,X,y,t,Z)
     char *fname;     /* name of file to write solution to. */
     int n;           /* Dimension of X */
     int k;           /* Number of equality constraints.  */
     int l;           /* Number of inequality constraints. (m=k+l) */
     int lda;         /* leading dimension for arrays of size n. */
     int nblocks;     /* Number of blocks in A, C, X, and Z. */
     int *block_structure /* Block Structure of A, C, X, and Z. */
     double *X;       /* X matrix, in column major order. */
     double *y;       /* y vector. */
     double *t;       /* t vector. */
     double *Z;       /* Z matrix, in column major order. */
```

The calling sequence for `read_sol` is

```
void read_sol(fname,pn,pk,pl,lda,nblocks,block_structure,X,y,t,Z)
     char *fname;     /* name of file to read solution from. */
     int *pn;         /* Dimension of X */
     int *pk;         /* Number of equality constraints.  */
     int *pl;         /* Number of inequality constraints. (m=k+l) */
     int lda;         /* leading dimension for arrays of size n. */
     int nblocks;     /* Number of blocks in A, C, X, and Z. */
     int *block_structure /* Block Structure of A, C, X, and Z. */
     double *X;       /* X matrix, in column major order. */
     double *y;       /* y vector. */
     double *t;       /* t vector. */
     double *Z;       /* Z matrix, in column major order. */
```

Installing CSDP 2.3 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 into a directory in your search path, such as "/usr/local/bin" and copy the libraries libsdp.a, libblas.a, and liblinpack.a into a directory where they will be found by your C compiler, such as "/usr/local/lib".

Installing CSDP 2.3 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 csdp2.3.tar" to expand the tar archive. Under Windows, use "pkunzip csdp23.zip" to expand the zip archive.

2. Determine whether you'll use LINPACK or LAPACK libraries, and where these libraries can be found. 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 you can use the linear algebra routines supplied with CSDP, although performance may suffer.

3. If you are using the linear algebra routines supplied with CSDP, then simply issue the command "make generic" to build the system. This will compile the C source code and produce the CSDP libraries and executable programs.

4. If you want to make use of preexisting LINPACK or LAPACK libraries, then use "make syslinpack" or "make syslapack" to build the system. You will probably have to modify the "LIBS=" line in the makefiles to specify where the libraries are located.

5. Once you have built the code, you can test that everything worked by going to the theta/testprob directory and running "../theta g50" The file g50.out contains correct output for this problem- if your output isn't extremely close, then something is wrong. Similarly, you can test the stand alone solver by going to the solver/testprob directory and running "../csdp theta1.dat–s". The file theta1.csdp.out contains correct output for this problem.

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

7. In order to use the SDP routine with your own code, you'll need to copy the files libsdp.a, libblas.a, and liblinpack.a from their respective directories and put them where the compiler will be able to find them when you compile your code. You should also copy the programs csdp, sqlptosdpa, sdpatosqlp, theta, rand_graph, complement, and graphtoprob to a directory that is in your search path.

Likely Problems

1. Passing strings to Fortran subroutines. This is done in different ways on different systems. If you make use of LINPACK or LAPACK library that already exists on your system, you may have problems calling the Fortran subroutines in these libraries. This is a known problem with AIX systems. To activate code that passes arguments to LINPACK/LAPACK correctly under AIX, add "-DAIX" to the "CFLAGS=" line in the make files. It is also often necessary to include Fortran runtime libraries when linking to the LINPACK or LAPACK library.

2. 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.

Installation Experience

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

System	Make target	Notes
Sun Solaris 2.5.1	make generic	CFLAGS=-O5 -xtarget=native -dalign -fnonstd
Sun SunOS 4.1.3	make generic	CFLAGS=-O4
IBM AIX 4.x	make generic	CFLAGS=-O3
Linux (RedHat 5.0)	make generic	CFLAGS=-O3
MS Windows-95	make generic	Used MINGW32 gcc C compiler and utilities make, ar, cp,rm. CFLAGS=-O3
DEC Alpha OSF/1	make generic	CFLAGS=-O4 -Olimit 750
IBM AIX 4.x	make syslinpack	added -lxlf -lxlfutil -lxlf90 to LIBS= CFLAGS=-DAIX
IBM AIX 4.x	make syslapack	added -lxlf -lxlfutil -lxlf90 to LIBS= CFLAGS=-DAIX -DLAPACK
Sun Solaris 2.5.1	make syslinpack	added -lsunperf -lM77 -lF77 -lsunmath to LIBS= CFLAGS=-O5 -xtarget=native -dalign Used SunSoft performance library
Sun Solaris 2.5.1	make syslapack	added -lsunperf -lM77 -lF77 -lsunmath to LIBS= CFLAGS=-O5 -xtarget=native -dalign -DLAPACK Used SunSoft performance library