Mathematical Programming manuscript No.
(will be inserted by the editor)

Brian Borchers

Implementation Issues in CSDP

February 28, 2001

Abstract. CSDP is a software package that implements the XZ primal–dual interior point method for semidefinite programming. In this method there are a number of computationally intensive steps including the construction of the Schur complement matrix O, the factorization of O, multiplications and Cholesky factorizations of matrices of size n. Profiling shows that for different problems, each of these types of operations can dominate the CPU time used by CSDP. We discuss ways in which the performance of these operations has been optimized in CSDP. We also discuss techniques for improving the accuracy of the solutions by scaling and ignoring small pivots in the factorization of the Schur complement matrix. As with interior point methods for linear programming, we have found that this can improve the accuracy of the solutions that are obtained.

Key words. Semidefinite Programming – Interior Point Methods

Introduction

CSDP is a software package for semidefinite programming that consists of a callable library and a stand alone solver for problems expressed in the SDPA sparse format. CSDP is written in portable ANSI C, and has been ported to many platforms, including Windows, Linux, Sun Solaris 7, AIX, and other versions of UNIX. Further information on CSDP can be found in [2].

This paper discusses several important issues in the implementation of CSDP. These issues include support for matrices with block diagonal structure, the computation of the Schur complement matrix O, and the process of solving the resulting system of equations. Our primary concern in this paper is issues of computational complexity and speed in practice, but we will also briefly discuss some issues of numerical accuracy.

Brian Borchers e-mail: borchers@nmt.edu: New Mexico Tech

CSDP solves semidefinite programming problems of the form:

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

where

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

Here the matrices C, X, and A_i are of symmetric matrices of size n by n. The constraint $X \succeq 0$ means that X must be a positive semidefinite matrix. 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

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

The DIMACS challenge library consists of mixed semidefinite quadratic linear programs. CSDP is not designed to handle quadratic cone constraints. However, linear constraints can easily be encoded as semidefinite constraints, and CSDP can be used to solve mixed semidefinite linear programming problems.

The algorithm used by CSDP is a predictor–corrector variant of the primal–dual interior point method of Helmberg, Rendl, Vanderbei, and Wolkowicz [6]. This method was also independently discovered by Kojima, Shindo and Hara [7], and Monteiro [8]. Thus the method is known as the HRVW/KSH/M or HKM method. This method is also known as the XZ method because the complementarity condition is expressed in the form $XZ = \mu I$.

The problem data consist of the matrices A_i, the objective matrix C, and the right hand side vector a. In addition, an initial primal–dual solution (X^0,y^0,Z^0) must be provided. If the A_i matrices are dense, then $O(mn^2)$ storage could be required for the problem data. However, the A_i are typically sparse so that $O(n^2)$ storage is required for the problem data and initial solution. In addition to the problem data, the XZ method requires $O(m^2)$ storage for the m by m Schur complement matrix O. In practice m is often considerably larger than n and thus the storage required by the Schur complement matrix often limits the size of problems that can be solved by CSDP.

Two of the most important steps in this algorithm are the construction of the m by m Schur complement matrix O and its Cholesky factorization. Other important operations include matrix multiplications involving matrices of size n, and Cholesky factorizations of matrices of size n. It is relatively straightforward to analyze the computational complexity of these basic operations. For dense constraint matrices, constructing O takes $O(mn^3 + m^2n^2)$ time. For sparse constraint matrices with $O(1)$ entries, constructing O takes $O(mn^2 + m^2)$ time. Factoring O takes $O(m^3)$ time. Multiplying matrices of size n takes $O(n^3)$ time. Factoring matrices of size n takes $O(n^3)$ time.

In practice, m is typically of the same size or larger than n. Thus it would appear that the operations of constructing O and factoring O should be the most time consuming. However, since the computational complexity depends very much on the relative sizes of m and n, as well as the sparsity of the constraint matrices A_i, it can be difficult to apply the above analysis in practice. Table 1 shows a summary of the time spent in constructing O, factoring O, and other operations in solving a collection of test problems from the DIMACS Challenge library. This collection consists of all of the problems from the library that were small enough to be solved by CSDP within 512 megabytes of storage. Notice that all three categories are dominant on at least some of the problems. Thus it is important to pay attention to all of these areas in implementing the primal–dual interior point method.

Block Diagonal Structure

Many semidefinite programming problems involve several semidefinite matrices $X^{(1)}$, $X^{(2)}$, ..., $X^{(k)}$. Since a block diagonal matrix is positive semidefinite if and only if its individual blocks are positive semidefinite, we can construct X by using the $X^{(i)}$ matrices as diagonal blocks of X,

$$X = \begin{bmatrix} X^{(1)} & & & \\ & X^{(2)} & & \\ & & \cdots & \\ & & & X^{(k)} \end{bmatrix}. \tag{5}$$

Other problems involve a vector x of nonnegative variables as in linear programming (LP). The linear programming constraint $x \geq 0$ can be expressed in SDP form by setting up a diagonal matrix block X^{LP} with $X^{\mathrm{LP}} = \mathrm{diag}(x)$. Since some problems involve a very large number of LP variables, it is worthwhile to store this block of X as a vector containing only the diagonal elements. This can save considerable storage. We

Problem	m	n	Make O	Factor O	Other
bm1	883	882	15%	2%	83%
copo14	1275	560	8%	72%	20%
copo23	5820	2300	4%	89%	7%
hamming_7_5_6	1793	128	7%	74%	19%
hamming_9_8	2305	512	10%	47%	43%
hinf12	43	24	74%	1%	25%
hinf13	57	30	32%	10%	58%
minphase	48	48	75%	$<$1%	25%
torusg3–8	512	512	1%	3%	96%
toruspm3–8–50	512	512	1%	3%	96%
truss5	208	331	38%	15%	47%
truss8	496	628	48%	25%	27%

Table 1. CPU time profiles for sample problems.

also save time by using special purpose routines for matrix multiplication and testing for positive definiteness which take advantage of the block's diagonal structure.

An examination of the problems in the DIMACS Challenge library reveals that some of these problems have a relatively large number of relatively small SDP blocks. Table 2 shows the block structure of our collection of problems from the DIMACS Challenge library. For example, the truss5 problem has 33 blocks of size 10 by 10 and one block of size 1 by 1.

CSDP takes advantage of this block diagonal structure in two important ways. First, CSDP conserves storage by storing the individual blocks in X and not storing the off diagonal zeros. Many large problems would require far more memory without this approach. Second, CSDP processes the blocks within X one at a time. Since operations on the blocks typically run in time proportional to the size of the block cubed, this is vastly more efficient than operating on the entire X matrix at one time.

Exploiting Sparsity

In the XZ method used by CSDP, as well as in other primal–dual methods for SDP, the major step in each iteration of the algorithm is the construction and solution of a large dense symmetric and positive definite linear system of equations. In CSDP, this system is of the form $O\Delta y = A(Z^{-1}F_d X) - a$. The matrix O is of size m by m. Since O is typically dense, it becomes very difficult to store O as the number of constraints increases. For example, on a workstation with 512 megabytes of RAM, and using double precision arithmetic, a practical limit on the size of O is about $m = 5,000$.

An analysis of the problems in the DIMACS Challenge library of SDP test problems shows that many of the constraints in these problems are extremely sparse. In some problems, as many as 99% of the constraints have only one entry in the upper triangle of the A_i matrix. In other problems, many of the constraint blocks are empty. This extreme sparsity of the constraint matrices is critical to the performance of CSDP. Table 3 summarizes the sparsity of our collection of test problems.

Problem	m	n	SDP blocks	LP variables
bm1	883	882	1x882	0
copo14	1275	560	14x14	364
copo23	5820	2300	23x23	1771
hamming_7_5_6	1793	128	1x128	0
hamming_9_8	2305	512	1x515	0
hinf12	43	24	1x6; 1x6; 1x12	0
hinf13	57	30	1x7; 1x9; 1x14	0
minphase	48	48	1x48	0
torusg3–8	512	512	1x512	0
toruspm3–8–50	512	512	1x512	0
truss5	208	331	33x10; 1x1	0
truss8	496	628	33x19; 1x1	0

Table 2. Block Structure of the SDP Problems.

CSDP has been designed to take advantage of sparse constraints in the construction of the Schur complement matrix O. O is given by

$$O = \left[A(Z^{-1}A_1X) \ \ldots \ A(Z^{-1}A_mX) \right] \tag{6}$$

Column j of O is given by

$$O_{\cdot,j} = A(Z^{-1}A_jX) \tag{7}$$

We can also write the entry of O in row i, column j with

$$O_{i,j} = O_{j,i} = \text{tr}(A_iZ^{-1}A_jX). \tag{8}$$

Since taking the trace of a block diagonal matrix is additive across the blocks of the matrix, we can go further and write

$$O_{i,j} = O_{j,i} = \text{tr}(A_i^{(1)}{Z^{(1)}}^{-1}A_j^{(1)}X) + \ldots + \text{tr}(A_i^{(k)}{Z^{(k)}}^{-1}A_j^{(k)}X). \tag{9}$$

CSDP uses different approaches for constructing O depending on the number and sparsity of the constraint matrices A_i. Furthermore, since the structure and sparsity of different blocks of the constraint matrices

Problem	% 0 Blocks	% One Entry	% Fill
bm1	0.0%	99.9%	100.0%
copo14	86.6%	13.3%	100.0%
copo23	91.7%	8.3%	100.0%
hamming_7_5_6	0.0%	99.9%	100.0%
hamming_9_8	0.0%	100.0%	100.0%
hinf12	33.3%	32.6%	71.3%
hinf13	33.3%	32.7%	65.0%
minphase	0.0%	2.1%	100.0%
torusg3–8	0.0%	100.0%	1.4%
toruspm3–8–50	0.0%	100.0%	1.4%
truss5	60.1%	39.9%	100.0%
truss8	50.9%	49.1%	100.0%

Table 3. Sparsity of the sample problems.

can vary greatly, CSDP selects different methods for each block within the block diagonal structure. Contributions from the blocks 1, 2, . . ., k are added into the O matrix one at a time. To simplify the notation in the following, we will proceed as if our matrices have only a single block.

The first approach is to simply compute the entries of O by direct application of (8). Suppose that A_i has an entry $A_{i,p,q}$ in row p and column q, while A_j has an entry $A_{j,r,s}$ in row r and column s. Then the contribution to the trace from these elements is given by $A_{i,p,q}Z^{-1}_{q,r}A_{j,r,s}X_{s,p}$. We add up the contributions from each pair of elements $A_{i,p,q}$ and $A_{j,r,s}$.

The second approach is a direct application of equation (7). For each column j, $j = 1, 2, \ldots, m$, we compute the matrix $Z^{-1}A_jX$, and then apply the $A()$ operator to this matrix. In computing $Z^{-1}A_jX$, we can treat A_j as a dense matrix and multiply $Z^{-1}A_j$ using conventional matrix multiplication and then multiply this product times X, or we can treat A_j as a sparse matrix and compute the contributions to the product corresponding to individual entries in A_j. If A_j has an entry in row p, column q, then its contribution to row r, column s of $Z^{-1}A_jX$ is given by $Z^{-1}_{r,p}A_{j,p,q}X_{q,s}$. For constraint matrices A_j with a very small number of entries, this approach is much more efficient than conventional matrix multiplication.

The advantage of the first approach is that it is very fast when both A_i and A_j are extremely sparse. In our experience, the constraint matrices often include blocks with only one or two nonzero entries. Furthermore, since we are dealing with individual blocks within the A_i and A_j matrices, it is often the case that one of the two blocks is completely empty, and no computation needs to be performed. However, the first approach can be very slow whenever one of the blocks in A_i or A_j is relatively dense. The advantage of the second approach is that it can save work by computing $Z^{-1}A_jX$ once and then reusing this matrix throughout the computation of column j of O. The disadvantage of the second approach is that computing $Z^{-1}A_jX$ is time consuming, and this time might be greater than the time saved by reusing the product.

CSDP uses a mixed approach that combines the advantages of both approaches. The second approach is used for columns j where A_j is relatively dense, while the first, one element at a time, approach is used for columns where A_j is sparse.

It is important to note that the $A()$ operator requires only those entries in $Z^{-1}A_jX$ which correspond to some nonzero element in one of the constraint matrices A_i, $i = 1, 2, \ldots, j$. For problems with extremely sparse constraint matrices, only a small fraction of the entries of $Z^{-1}A_jX$ are commonly required. It is relatively straightforward to modify the sparse matrix multiplication approach described above to compute only the required entries of $Z^{-1}A_jX$.

The dual constraint $A^T(y) - C = Z$ insures that the nonzero entries of Z and of the dual residual F_d occur only positions corresponding to nonzero entries in the C and A_i matrices. The % Fill column of Table 3 shows the percentage of nonzero entries in the Z matrix for the test problems. For some problems the density of nonzero entries is quite low, while for most problems in this set, Z can be fully dense. When Z is sparse, CSDP takes advantage of the sparsity in matrix multiplications.

Optimized Linear Algebra Routines

Our profiling has shown that many of the most time consuming operations in CSDP are in the numerical linear algebra. This includes computing the Cholesky factorization of O, using the Cholesky factorization of O to solve systems of equations, computing Z^{-1}, computing various matrix products, and computing Cholesky factorizations of X and Z to check for positive definiteness. These operations are all performed on dense matrices. Using the fastest available routines for these operations is critical to the performance of CSDP.

CSDP makes use of routines from the BLAS, LINPACK, and LAPACK libraries [1,3–5]. Source code reference implementations for these routines are freely available and are included with CSDP. Furthermore, many manufacturers have developed versions of the libraries which are optimized for their computers. Experience has shown that using optimized linear algebra routines can speed up CSDP by a factor of 3 on some problems on some systems.

Table 4 gives a summary of the performance of CSDP on our collection of test problems using the reference BLAS and LINPACK routines, versus the performance with highly optimized BLAS and LAPACK routines. In both cases, CSDP itself was compiled using the same optimization. The only difference is in the linear algebra libraries. These runs were performed on a 450 Mhz Pentium II system. The ASCI Red Pentium Pro BLAS version 1.1o and LAPACK routines configured for the Pentium II cache were used in the optimized case. On the larger problems, the code with optimized linear algebra routines is typically two to three times faster than the base code. For smaller problems which fit into cache, the performance of the two versions of CSDP is very similar.

Accuracy Issues

A major problem in the XZ method is that the Schur complement matrix O typically becomes singular or even indefinite as the solution approaches optimality. Two techniques are used to deal with this problem. In

solving the system of equations

$$O\Delta y = A(Z^{-1}F_D X) - a \tag{10}$$

we scale the system of equations to improve conditioning. In factoring O, we ignore small pivots. In particular, any pivot element of value less than 1.0e-14 in the Cholesky factorization is set to a very large floating point value. As a result, the corresponding elements of Δy are set to 0. This technique has been shown to work well both in practice and in theory in interior point methods for LP [9]. It also seems to help in the XZ method for SDP.

Table 5 shows the results of tests on our sample problems. These runs were performed on a 450 Mhz Pentium II machine. The first set of results are for the code without scaling and ignoring small pivots. The second set of results include both features. For several of these problems CSDP's default stopping criteria are easily met with either version of the code. For this reason the tolerance on the relative duality gap

$$\frac{|a^T y - \text{tr}(CX)|}{1 + |\text{tr}(CX)|} \tag{11}$$

Problem	Reference	Optimized
bm1	10273	4921
copo14	341	126
copo23	33763	8751
hamming_7_5_6	578	180
hamming_9_8	2102	719
hinf12	1	1
hinf13	1	1
minphase	5	5
torusg3–8	385	207
toruspm3–8–50	367	198
truss5	3	3
truss8	36	29

Table 4. CPU times for CSDP 3.2 using reference implementations of BLAS and LINPACK versus optimized BLAS and LAPACK.

was decreased to 1.0e-14. For each run, we give the relative duality gap of the best solution obtained by CSDP, the relative primal infeasibility and the relative dual infeasibility. For problems copo14, copo23, hamming_7_5_6, hamming_9_8, torusg3–8, toruspm3–8–50, truss5, and truss8, both versions of the code produced very good solutions. For problems bm1, hinf12, hinf13, and minphase, neither version of the code was able to produce very accurate solutions.

However, on these difficult problems the version of the code with these two features consistently produced more accurate solutions than the code without these two features. For example, on problem hinf12, the version of the code without the two features was unable to find a dual feasible solution, while the other version of the code found a solution that met CSDP's tolerances for primal and dual infeasibility and had a relative duality gap of 6.5e-04.

Problem	w/o dropping small pivots			dropping small pivots		
	reldgap	relpinf	reldinf	reldgap	relpinf	reldinf
bm1	5.9e-03	5.9e-07	1.3e-03	4.6e-05	2.5e-07	9.2e-05
copo14	4.3e-14	9.3e-15	2.5e-16	6.0e-15	5.6e-12	1.6e-16
copo23	2.8e-12	7.5e-14	1.8e-16	4.5e-12	1.0e-13	1.9e-16
hamming_7_5_6	4.1e-11	1.1e-16	0.0e+00	7.0e-15	2.9e-16	0.0e+00
hamming_9_8	3.7e-15	5.7e-17	0.0e+00	2.7e-15	1.5e-16	0.0e+00
hinf12	8.0e-13	9.8e-12	1.8e+01	6.5e-04	2.7e-11	1.2e-08
hinf13	2.7e-02	4.8e-05	9.3e-12	3.5e-06	2.5e-06	1.4e-10
minphase	2.0e-03	8.7e-09	0.0e+00	2.8e-04	2.1e-08	0.0e+00
torusg3–8	9.7e-15	2.7e-15	1.0e-16	8.5e-15	1.6e-15	1.0e-16
toruspm3–8–50	7.7e-15	1.7e-15	1.0e-16	8.6e-15	1.8e-15	9.3e-17
truss5	3.5e-13	1.8e-11	6.0e-15	1.6e-12	4.2e-10	5.7e-15
truss8	3.4e-12	1.2e-09	6.0e-15	4.9e-13	5.6e-10	8.7e-15

Table 5. Relative duality gaps with and without dropping small pivots.

Conclusions

In this paper we have discussed a number of techniques used in the implementation of CSDP. Profiling showed that on different problems the amount of CPU time spent on the various steps of the algorithm vary tremendously. Thus it was worthwhile to optimize each of the steps in the algorithm.

Optimization implemented in CSDP include exploiting block structure in the X matrix, exploiting sparsity in the constraint matrices, using high performance BLAS and LAPACK routines for numerical linear algebra, and ignoring small pivots in the factorization of the Schur complement matrix. Profiling and benchmarks have confirmed the effectiveness of these optimizations.

Numerical difficulties caused by the fact that the Schur complement matrix becomes numerically singular as the solution approaches optimality are addressed by scaling the system of equations and by ignoring small pivots in the Cholesky factorization of the Schur complement matrix. These techniques resulted in improved solutions on some of the problems from the DIMACS Challenge library.

References

1. E. Andersen, Z. Bai, C. Bischof, L. S. Blackford, and J. Demmel. *Lapack User's Guide*. SIAM, Philadelphia, 2000.
2. Brian Borchers. CSDP, a C library for semidefinite programming. *Optimization Methods & Software*, 11 & 12:613–623, 1999.
3. J. J. Dongarra, J. Ducroz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.
4. J. J. Dongarra, J. Ducroz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.
5. J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *Linpack User's Guide*. SIAM, Philadelphia, 1979.
6. Christoph Helmberg, Franz Rendl, Robert J. Vanderbei, and Henry Wolkowicz. An interior–point method for semidefinite programming. *SIAM Journal on Optimization*, 6(2):342–361, 1996.
7. Masakazu Kojima, Susumu Shindoh, and Shinji Hara. Interior–point methods for the monotone semidefinite linear complementarity problem in symmetric matrices. *SIAM Journal on Optimization*, 7(1):86–125, 1997.

8. Renato D. C. Monteiro. Primal–dual path–following algorithms for semidefinite programming. *SIAM Journal on Optimization*, 7(3):663–678, 1997.

9. S. J. Wright. Modified Cholesky factorizations in interior-point algorithms for linear programming. *SIAM Journal on Optimization*, 9:1159–1191, 1999.