ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-277

DSDP5 User Guide – Software for Semidefinite Programming*

by

Steven J. Benson† *and Yinyu Ye*‡

Mathematics and Computer Science Division

Technical Memorandum No. 277

September 2005

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†`http://www.mcs.anl.gov/~benson`

‡Department of Management Science and Engineering, Stanford University, Stanford, CA 94305, `http://www.stanford.edu/~yyye`

About Argonne National Laboratory
Argonne is managed by The University of Chicago for the U.S. Department of Energy's Office of Science, under contract W-31-109-Eng-38. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

Availability of This Report
This report is available, at no cost, at http://www.osti.gov/bridge. It is also available on paper to U.S. Department of Energy and its contractors, for a processing fee, from:

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
phone (865) 576-8401
fax (865) 576-5728
reports@adonis.osti.gov

Contents

DSDP5 User Guide - Software for Semidefinite Programming

by

Steven J. Benson and Yinyu Ye

Abstract

DSDP implements the dual-scaling algorithm for semidefinite programming. The source code of this interior-point solver, written entirely in ANSI C, is freely available. The solver can be used as a subroutine library, as a function within the Matlab environment, or as an executable that reads and writes to files. Initiated in 1997, DSDP has developed into an efficient and robust general-purpose solver for semidefinite programming. Although the solver is written with semidefinite programming in mind, it can also be used for linear programming and other constraint cones.

The features of DSDP include the following:

- a robust algorithm with a convergence proof and polynomially bounded complexity under mild assumptions on the data,
- primal and dual solutions,
- feasible solutions when they exist or approximate certificates of infeasibity,
- initial points that can be feasible or infeasible,
- relatively low memory requirements for an interior-point method,
- sparse and low-rank data structures,
- extensibility that allows applications to customize the solver and improve its performance,
- a subroutine library that enables it to be linked to larger applications,
- scalable performance for large problems on parallel architectures, and
- a well-documented interface and examples of its use.

The package has been used in many applications and tested for efficiency, robustness, and ease of use. We welcome and encourage further use under the terms of the license included in the distribution.

1 Notation

The DSDP package implements a dual-scaling algorithm to find solutions (X_j, y_i, S_j) to linear and semidefinite optimization problems of the form

$$(P) \qquad \inf \sum_{j=1}^{p} \langle C_j, X_j \rangle \quad \text{subject to} \quad \sum_{j=1}^{p} \langle A_{i,j}, X_j \rangle = b_i, \quad i = 1, \ldots, m, \qquad X_j \in K_j,$$

$$(D) \qquad \sup \sum_{i=1}^{m} b_i \; y_i \quad \text{subject to} \quad \sum_{i=1}^{m} A_{i,j} y_i + S_j = C_j, \quad j = 1, \ldots, p, \quad S_j \in K_j.$$

In this formulation, b_i and y_i are real scalars.

For semidefinite programming, the data $A_{i,j}$ and C_j are symmetric matrices of dimension n_j ($\mathbb{S}^{n_j}$), and the cone K_j is the set of symmetric positive semidefinite matrices of the same dimension. The inner product $\langle C, X \rangle := C \bullet X := \sum_{k,l} C_{k,l} X_{k,l}$, and the symbol $\succ$ ($\succeq$) means the matrix is positive (semi)definite. In linear programming, A_i and C are vectors of real scalars, K is the nonnegative orthant, and the inner product $\langle C, X \rangle$ is the usual vector inner product.

More generally, users specify $C_j, A_{i,j}$ from an inner-product space V_j that intersects a cone K_j. Using the notation summarized in Table 1, let the symbol $\mathcal{A}$ denote the linear map $\mathcal{A} : V \to \mathbb{R}^m$ defined by $(\mathcal{A}X)_i = \langle A_i, X \rangle$; its adjoint $\mathcal{A}^T : \mathbb{R}^m \to V$ is defined by $\mathcal{A}^T y = \sum_{i=1}^{m} y_i A_i$. Equivalent expressions for (P) and (D) can be written

$$\begin{array}{lllll} (P) & \inf \langle C, X \rangle & \text{subject to} & \mathcal{A}X = b, & X \in K, \\ (D) & \sup \; b^T y & \text{subject to} & \mathcal{A}^T y + S = C, & S \in K. \end{array}$$

Formulation (P) will be referred to as the *primal* problem, and formulation (D) will be referred to as the *dual* problem. Variables that satisfy the linear equations are called feasible, whereas the others are called infeasible. The interior of the cone will be denoted by $\hat{K}$, and the interior feasible sets of (P) and (D) will be denoted by $\mathcal{F}^0(P)$ and $\mathcal{F}^0(D)$, respectively.

Table 1: Terms and notation for linear (LP), semidefinite (SDP), and conic programming.

Term	LP	SDP	Conic	Notation
Dimension	n	n	$\sum n_j$	n
Data Space ($\ni C, A_i$)	$\mathbb{R}^n$	$\mathbb{S}^n$	$V_1 \oplus \ldots \oplus V_p$	V
Cone	$x, s \geq 0$	$X, S \succeq 0$	$X, S \in K_1 \oplus \ldots \oplus K_p$	$X, S \in K$
Interior of Cone	$x, s > 0$	$X, S \succ 0$	$X, S \in \hat{K}_1 \oplus \ldots \oplus \hat{K}_p$	$X, S \in \hat{K}$
Inner Product	$c^T x$	$C \bullet X$	$\sum \langle C_j, X_j \rangle$	$\langle C, X \rangle$
Norm	$\|x\|_2$	$\|X\|_F$	$\left(\sum \|X_j\|^2\right)^{1/2}$	$\|X\|$
Product	$[x_1 s_1 \ldots x_n s_n]^T$	XS	$X_1 S_1 \oplus \ldots \oplus X_p S_p$	XS
Identity Element	$[1 \ldots 1]^T$	I	$I_1 \oplus \ldots \oplus I_p$	I
Inverse	$[1/s_1 \ldots 1/s_n]^T$	S^{-1}	$S_1^{-1} \oplus \ldots \oplus S_p^{-1}$	S^{-1}
Dual Barrier	$\sum \ln s_j$	$\ln \det S$	$\sum \ln \det S_j$	$\ln \det S$

2 Dual-Scaling Algorithm

This section summarizes the dual-scaling algorithm for solving (P) and (D). For simplicity, parts of this discussion assume that the cone is a single semidefinite block, but an extension of the algorithm to multiple blocks and other cones is relatively simple. This discussion also assumes that the A_is are linearly independent, there exists $X \in \mathcal{F}^0(P)$, and a starting point $(y, S) \in \mathcal{F}^0(D)$ is known. The next section discusses how DSDP generalizes the algorithm to relax these assumptions.

It is well known that under these assumptions, both (P) and (D) have optimal solutions X^* and (y^*, S^*), which are characterized by the equivalent conditions that the duality gap $\langle X^*, S^* \rangle$ is zero and the product X^*S^* is zero. Moreover, for every $\nu > 0$, there exists a unique primal-dual feasible solution (X_ν, y_ν, S_ν) that satisfies the perturbed optimality equation $X_\nu S_\nu = \nu I$. The set of all solutions $\mathcal{C} \equiv \{(X_\nu, y_\nu, S_\nu) : \nu > 0\}$ is known as the central path, and $\mathcal{C}$ serves as the basis for path-following algorithms that solve (P) and (D). These algorithms construct a sequence $\{(X, y, S)\} \subset \mathcal{F}^0(P) \times \mathcal{F}^0(D)$ in a neighborhood of the central path such that the duality gap $\langle X, S \rangle$ goes to zero. A scaled measure of the duality gap that proves useful in the presentation and analysis of path-following algorithms is $\mu(X, S) = \langle X, S \rangle / n$ for all $(X, S) \in K \times K$. Note that for all $(X, S) \in \hat{K} \times \hat{K}$, we have $\mu(X, S) > 0$ unless $XS = 0$. Moreover, $\mu(X_\nu, S_\nu) = \nu$ for all points (X_ν, y_ν, S_ν) on the central path.

The dual-scaling algorithm applies Newton's method to $\mathcal{A}X = b$, $\mathcal{A}^T y + S = C$, and $X = \nu S^{-1}$ to generate

$$\begin{aligned} \mathcal{A}(X + \Delta X) &= b, & (1)\\ \mathcal{A}^T(\Delta y) + \Delta S &= 0, & (2)\\ \nu S^{-1} \Delta S S^{-1} + \Delta X &= \nu S^{-1} - X. & (3) \end{aligned}$$

Equations (1) - (3) will be referred to as the Newton equations; their Schur complement is

$$\nu \begin{pmatrix} \langle A_1, S^{-1}A_1S^{-1} \rangle & \cdots & \langle A_1, S^{-1}A_mS^{-1} \rangle \\ \vdots & \ddots & \vdots \\ \langle A_m, S^{-1}A_1S^{-1} \rangle & \cdots & \langle A_m, S^{-1}A_mS^{-1} \rangle \end{pmatrix} \Delta y = b - \nu \mathcal{A} S^{-1}. \quad (4)$$

The left-hand side of this linear system is positive definite when $S \in \hat{K}$. In this manuscript, it will sometimes be referred to as M. DSDP computes $\Delta' y := M^{-1} b$ and $\Delta'' y := M^{-1} \mathcal{A} S^{-1}$. For any ν,

$$\Delta_\nu y := \frac{1}{\nu} \Delta' y - \Delta'' y$$

solves (4). We use the subscript to emphasize that ν can be chosen after computing $\Delta' y$ and $\Delta'' y$ and that the value chosen for the primal step may be different from the value chosen for the dual step.

Using (2),(3), and $\Delta_\nu y$, we get

$$X(\nu) := \nu \left(S^{-1} + S^{-1}(\mathcal{A}^T \Delta_\nu y) S^{-1} \right), \quad (5)$$

which satisfies $\mathcal{A}X(\nu) = b$. Because $X(\nu) \in \hat{K}$ if and only if

$$C - \mathcal{A}^T(y - \Delta_\nu y) \in \hat{K}, \quad (6)$$

DSDP applies a Cholesky factorization on (6) to test the condition. If $X(\nu) \in \hat{K}$, a new upper bound

$$\bar{z} := \langle C, X(\nu) \rangle = b^T y + \langle X(\nu), S \rangle = b^T y + \nu \left(\Delta_\nu y^T \mathcal{A} S^{-1} + n \right) \tag{7}$$

can be obtained without explicitly computing $X(\nu)$. The dual-scaling algorithm does not require $X(\nu)$ to compute the step direction defined by (4), so DSDP does not compute it unless specifically requested. This feature characterizes the algorithm and its performance.

To get closer to the central path and make further use of M, whose computation and factorization usually dominate the computation time, DSDP generates a sequence of *corrector* steps. The corrector steps compute $\mathcal{A} S^{-1}$ using the current S and $\Delta^c y := M^{-1} \mathcal{A} S^{-1}$. Since the computation of M used previous values of S, the corrector step

$$\Delta^c_\nu y := \frac{1}{\nu} \Delta' y - \Delta^c y$$

is not a Newton step. A line search computes a step length, α_c, that improves the merit function

$$\phi_\nu(y) := b^T y + \nu \ln \det S. \tag{8}$$

Between 0 and 12 corrector steps are applied each iteration. The exact number can be chosen heuristically based on the square of the ratio of m and the dimension of the largest semidefinite block.

Either (y, S) or X reduces the dual potential function

$$\psi(y) := \rho \log(\bar{z} - b^T y) - \ln \det S \tag{9}$$

enough at each iteration to achieve linear convergence.

1: Set up data structures and factor A_i.
2: Choose y such that $S \leftarrow C - \mathcal{A}^T y \in \hat{K}$.
3: Choose an upper bound $\bar{z}$ and a barrier parameter ν.
4: **for** $k \leftarrow 0, \ldots, k_{max}$ **do**
5: Monitor solution and check for convergence.
6: Compute M and $\mathcal{A}S^{-1}$.
7: Solve $M\Delta' y = b$, $M\Delta'' y = \mathcal{A}S^{-1}$.
8: **if** $C - \mathcal{A}^T(y - \Delta_\nu y) \in \hat{K}$ **then**
9: $\bar{z} \leftarrow b^T y + \nu\left(\Delta_\nu y^T \mathcal{A}S^{-1} + n\right)$.
10: $\overline{y} \leftarrow y$, $\overline{\Delta y} \leftarrow \Delta_\nu y$, $\overline{\mu} \leftarrow \nu$.
11: **end if**
12: Select ν.
13: Find α_d to reduce ψ, and set $y \leftarrow y + \alpha_d \Delta_\nu y$, $S \leftarrow C - \mathcal{A}^T y$.
14: **for** $kk = 1, \ldots, kk_{max}$ **do**
15: Compute $\mathcal{A}S^{-1}$.
16: Solve $M\Delta^c y = \mathcal{A}S^{-1}$.
17: Select ν.
18: Find α_c to reduce ϕ_ν, and set $y \leftarrow y + \alpha_c \Delta^c_\nu y$, $S \leftarrow C - \mathcal{A}^T y$.
19: **end for**
20: **end for**
21: Optional: Compute X using $\overline{y}$, $\overline{\Delta y}$, $\overline{\mu}$.

3 Standard Form

The convergence of the algorithm assumes that both (P) and (D) have an interior feasible region and the current solutions are elements of the interior. To satisfy these assumptions, DSDP bounds the variables y such that $l \leq y \leq u$, where $l, u \in \mathbb{R}^m$. By default, $l_i = -10^7$ and $u_i = 10^7$ for each i from 1 through m. Furthermore, DSDP bounds the trace of X by a penalty parameter Γ whose default value is $\Gamma = 10^8$. Including these bounds and their associated Lagrange variables $x^l \in \mathbb{R}^m$, $x^u \in \mathbb{R}^m$, and r, DSDP solves the following pair of problems:

$$
\begin{array}{llccccccc}
(PP) & \text{minimize} & \langle C, X\rangle & + & u^T x^u & - & l^T x^l & & \\
 & \text{subject to} & \mathcal{A}X & + & x^u & - & x^l & = & b, \\
 & & \langle I, X\rangle & & & & & \leq & \Gamma, \\
 & & X \in K, & & x^u \geq 0, & & x^l \geq 0. & &
\end{array}
$$

$$
\begin{array}{llccc}
(DD) & \text{maximize} & b^T y - \Gamma r & & \\
 & \text{subject to} & C - \mathcal{A}^T y + Ir & = & S \in K, \\
 & & l \leq y \leq u, & & r \geq 0.
\end{array}
$$

The reformulations (PP) and (DD) are bounded and feasible, so the optimal objective values to this pair of problems are equal. Furthermore, (PP) and (DD) can be expressed in the form of (P) and (D).

Unless the user provides a feasible point y, DSDP uses the y values provided by the application (usually all zeros) and increases r until $C - \mathcal{A}^T y + Ir \in \hat{K}$. Large values of r

improve robustness, but smaller values often improve performance. In addition to bounding X, the parameter Γ penalizes infeasiblity in (D) and forces r toward zero. The nonnegative variable r increases the dimension m by one and adds an inequality to the original problem. The M matrix treats r separately by storing the corresponding row/column as a separate vector and applying the Sherman-Morrison-Woodbury formula. Unlike other inequalities, DSDP allows r to reach the boundary of the cone. Once $r = 0$, it is fixed and effectively removed from the problem.

The bounds on y add $2m$ inequality constraints to the original problem; and, with a single exception, DSDP treats them the same as the constraints on the original model. One difference between these bounds and the other constraints is that DSDP explicitly computes the corresponding Lagrangian variables x^l and x^u at each iteration to quantify the infeasibility in (P). The bounds l and u penalize infeasibility in (P), force x^l and x^u toward zero, and prevent numerical difficulties created by variables with large magnitude.

The solution to (PP) and (DD) is a solution to (P) and (D) when the optimal objective values of (P) and (D) exist and are equal and the bounds are sufficiently large. DSDP identifies unboundedness or infeasibility in (P) and (D) through examination of the solutions to (PP) and (DD). Given parameters ϵ_P and ϵ_D,

- if $r \leq \epsilon_r$, $\|\mathcal{A}X - b\|_\infty / \langle I, X \rangle > \epsilon_P$, and $b^T y > 0$, it characterizes (D) as unbounded and (P) as infeasible;
- if $r > \epsilon_r$ and $\|\mathcal{A}X - b\|_\infty / \langle I, X \rangle \leq \epsilon_P$, it characterizes (D) as infeasible and (P) as unbounded.

Normalizing unbounded solutions will provide an approximate certificate of infeasibility. Larger bounds may improve the quality of the certificate of infeasibility and permit additional feasible solutions, but they may also create numerical difficulties in the solver.

4 Iteration Monitor

The progress of the DSDP solver can be monitored by using standard output printed to the screen. The data below shows an example of this output.

```
Iter  PP Objective     DD Objective    PInfeas  DInfeas    Nu     StepLength   Pnrm
-------------------------------------------------------------------------------------
0    1.00000000e+02  -1.13743137e+05   2.2e+00  3.8e+02  1.1e+05  0.00  0.00   0.00
1    1.36503342e+06  -6.65779055e+04   5.1e+00  2.2e+02  1.1e+04  1.00  0.33   4.06
2    1.36631922e+05  -6.21604409e+03   5.4e+00  1.9e+01  4.5e+02  1.00  1.00   7.85
3    5.45799174e+03  -3.18292092e+03   1.5e-03  9.1e+00  7.5e+01  1.00  1.00  17.63
4    1.02930559e+03  -5.39166166e+02   1.1e-05  5.3e-01  2.7e+01  1.00  1.00   7.58
5    4.30074471e+02  -3.02460061e+01   3.3e-09  0.0e+00  5.6e+00  1.00  1.00  11.36
...
11   8.99999824e+00   8.99999617e+00   1.1e-16  0.0e+00  1.7e-08  1.00  1.00   7.03
12   8.99999668e+00   8.99999629e+00   2.9e-19  0.0e+00  3.4e-09  1.00  1.00  14.19
```

The program will print a variety of statistics for each problem to the screen.

`Iter`	the iteration number.
`PP Objective`	the upper bound $\bar{z}$ and objective value in (PP).
`DD Objective`	the objective value in (DD).
`PInfeas`	the primal infeasiblity in (P) is $\|x^u - x^l\|_\infty$.
`DInfeas`	the dual infeasibility in (D) is the variable r.
`Nu`	the barrier parameter ν.
`StepLength`	the multiple of the step-directions in (P) and (D).
`Pnrm`	the proximity to the central path: $\|\nabla\psi\|_{M^{-1}}$.

5 DSDP with MATLAB

Additional help using the DSDP can be found by typing `help dsdp` in the directory `DSDP5.X`. The command

```
> [STAT, y, X] = dsdp(b, AC)
```

attempts to solve the semidefinite program by using a dual-scaling algorithm. The first argument is the objective vector b in (D) and the second argument is a cell array that contains the structure and data for the constraint cones. Most data has a block structure, which should be specified by the user in the second argument. For a problem with p cones of constraints, `AC` is a $p \times 3$ cell array. Each row of the cell array describes a cone. The first element in each row of the cell array is a string that identifies the type of cone. The second element of the cell array specifies the dimension of the cone, and the third element contains the cone data.

5.1 Semidefinite Cones

If the cone j is a semidefinite cone consisting of a single block with n rows and columns in the matrices, then the first element in this row of the cell array is the string `'SDP'` and the second element is the number `n`, and the third element is a sparse matrix with $n(n+1)/2$ rows and $m+1$ columns. Columns 1 to m of this matrix represent the constraints $A_{1,j}, \ldots, A_{m,j}$ for this block, and column $m+1$ represents C_j.

The square symmetric data matrices $A_{i,j}$ and C_j map to the columns of `AC{j,3}` through the operator **dvec(·)** : $\mathbb{R}^{n\times n} \rightarrow \mathbb{R}^{n(n+1)/2}$, which is defined as

$$\mathbf{dvec}(A) \quad = [a_{1,1} \quad a_{1,2} \quad a_{2,2} \quad a_{1,3} \quad a_{2,3} \quad a_{3,3} \quad \ldots \quad a_{n,n}]^T.$$

In this definition, $a_{k,l}$ is the element in row k and column l of A. This ordering is often referred to as symmetric packed storage format. The inverse of **dvec( )** is **dmat(·)** : $\mathbb{R}^{n(n+1)/2} \rightarrow \mathbb{R}^{n\times n}$, which converts the vector into a square symmetric matrix. Using these operations, we obtain

$$A_{i,j} = \texttt{dmat(AC\{j,3\}(:,i))}, \quad C_j = \texttt{dmat(AC\{j,3\}(:,m+1))}$$

and

$$\texttt{AC\{j,3\}} = [\ \texttt{dvec}(A_{1,j}) \quad \ldots \quad \texttt{dvec}(A_{m,j}) \quad \texttt{dvec}(C_j)\ ];$$

For example, the problem

$$\begin{array}{lcccccc} \text{Maximize} & & & y_1 & + & & y_2 \\ \text{Subject to} & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & y_1 & + & \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} & y_2 & \preceq \begin{bmatrix} 4 & -1 \\ -1 & 5 \end{bmatrix} \end{array}$$

can be solved by

```
> b = [ 1 1 ]';
> AAC = [ [ 1.0 0 0 ]' [ 0 0 1.0 ]' [ 4.0 -1.0 5.0 ]' ];
> AC{1,1} = 'SDP';
```

```
> AC{1,2} = [2];
> AC{1,3} = sparse(AAC);
> [STAT,y,X]=dsdp(b,AC);
> XX=dmat(X{1});
```

The solution y is the column vector `y' = [ 3 4 ]'`, and the solution X is a $p \times 1$ cell array. In this example, the solution `X = [ 3 x 1  double ]` is `X{1}'=[ 1.0 1.0 1.0 ]`. Furthermore, `dmat(X{1})=[ 1 1 ; 1 1 ]`.

Each semidefinite block can be stated in a separate row of the cell array; only the available memory on the machine limits the number of cones that can be specified.

Each semidefinite block may, however, be grouped into a single row in the cell array. In order to group these blocks together, the second cell entry must be an array of integers stating the dimension of each block. The data from the blocks should be concatenated such that the number of rows in the data matrix increases whereas the number of columns remains constant. The following lines indicate how to group the semidefinite blocks in rows 1 and 2 of cell array `AC1` into a new cell array `AC2`:

```
> AC2{1,1} = 'SDP';
> AC2{1,2} = [AC1{1,2}  AC1{2,2}];
> AC2{1,3} = [AC1{1,3}; AC1{2,3}];
```

The new cell array `AC2` can be passed directly into DSDP. The advantage of grouping multiple blocks together is that it uses less memory – especially when there are many blocks and many of the matrices in these blocks are zero. The performance of DSDP, measured by execution time, will change very little.

This distribution contains several examples files in SDPA format. A utility routine called `readsdpa( · )` can read these files and put the problems in DSDP format. They may serve as examples of how to format an application for use by the `DSDP` solver. Another example can be seen in the file `maxcut( · )` , which takes a graph and creates an SDP relaxation of the maximum cut problem from a graph.

5.2 LP Cones

A cone of LP variables can specified separately. For example a randomly generated LP cone $A^T y \leq c$ with 3 variables y and 5 inequality constraints can be specified in the following code.

```
> n=5; m=3;
> b = rand(m,1);
> At=rand(n,m);
> c=rand(n,1);
> AC{1,1} = 'LP';
> AC{1,2} = n;
> AC{1,3} = sparse([At c]);
> [STAT,y,X]=dsdp(b,AC);
```

Multiple cones of LP variables may be passed into the DSDP solver, but for efficiency reasons, it is best to group them all together. This cone may also be passed to the DSDP solver as

a semidefinite cone, where the matrices A_i and C are diagonal. For efficiency reasons, however, it is best to identify them separately as belonging to the cone of 'LP' variables.

Although y variables that are fixed to a constant can be preprocessed and removed from a model, it is often more convenient to leave them in the model. It is more efficient to identify fixed variables to DSDP than to model these constraints as a pair of linear inequalities. The following example sets variables 1 and 8 to the values 2.4 and -6.1, respectively.

```
> AC{j,1} = 'FIXED'; AC{j,2} = [ 1 8 ]; AC{j,3} = [ 2.4 -6.1 ];
```

The corresponding variables x to these constraints may be positive or negative.

5.3 Solver Options

There are more ways to call the solver. The command

```
> [STAT,y,X] = DSDP(b,AC,OPTIONS)
```

specifies some options for the solver. The `OPTIONS` structure may contain any of the following fields that may *significantly* affect the performance of the solver. The following options affect the formulation of the problem:

`r0` initial value for r in (DD). If $r0 < 0$, a heuristic will select a very large number ($\sim$1e10). To improve convergence, try a smaller value. [default -1 (Heuristic)].

`zbar` an upper bound $\bar{z}$ on the objective value at the solution [default 1.0e10].

`penalty` penalty parameter Γ in (DD) that enforces feasibility in (D). IMPORTANT: This parameter must be positive and greater than the trace of the solution X of (P). [default 1e8].

`boundy` determines the bounds l and u on the variables y in (DD). That is, $-boundy = l \leq y_i \leq u = boundy$ for all i. [default: 1e7].

The following fields in the `OPTIONS` structure affect the stopping criteria for the solver:

`gaptol` tolerance for duality gap as a fraction of the value of the objective functions [default 1e-6].

`maxit` maximum number of iterations allowed [default 1000].

`steptol` tolerance for stopping because of small steps [default 1e-2].

`pnormtol` $\|P(\nu)\|$ of solution should also be less than [default 1e30].

`inftol` the value r in (DD) must be less than this tolerance to classify the final solution of (D) as feasible. [default 1e-8].

`dual_bound` Terminate the solver when it finds a feasible point of (D) with an objective greater than this value. (Helpful in branch-and-bound algorithms.) [default 1e+30].

The following fields in the `OPTIONS` structure affect printing:

`print` = k to display output at each k iteration, else = 0 [default 10].

`logtime` =1 to profile the performance of DSDP subroutines, else =0. (Assumes proper compilation flags.)

`cc` add this constant the objective value. This parameter is algorithmically irrelevant, but it can make the objective values displayed on the screen more consistent with the underlying application [default 0].

Other fields are also recognized in `OPTIONS` structure:

`rho` to set the potential parameter ρ in the function (9) to this multiple of the conic dimension n. [default: 3] IMPORTANT: Increasing this parameter to 4 or 5 may significantly improve performance.

`dynamicrho` to use dynamic rho strategy. [default: 1].

`bigM` if > 0, the variable r in (DD) will remain positive (as opposed to nonegative). [default 0].

`mu0` initial barrier parameter ν. [default -1: use heuristic]

`reuse` sets a maximum on the number of times the Schur complement matrix can be reused. Larger numbers reduce the number of iterations but increase the cost of each iteration. Applications requiring few iterations (< 60) should consider setting this parameter to 0. [default: 4]

For instance, the commands

```
> OPTIONS.gaptol = 0.001;
> OPTIONS.boundy = 1000;
> OPTIONS.rho = 5;
> [STAT,y,X] = DSDP(b,AC,OPTIONS);
```

ask for a solution with approximately three significant digits, bound the y variables by -1000 and $+1000$, and use a potential parameter ρ of 5 times the conic dimension. Some of these fields, especially `rho`, `r0`, and `ybound`, can significantly improve performance of the solver.

Using a fourth input argument, the command

```
> [STAT,y,X] = DSDP(b,AC,OPTIONS,y0);
```

specifies an initial solution `y0` in (D). The default starting vector is the zero vector.

5.4 Solver Performance and Statistics

The second and third output arguments return objective values for (D) and (P), respectively.

The first output argument is a structure with several fields that describe the solution of the problem:

`stype` `PDFeasible` if the solutions to both (D) and (P) are feasible, `Infeasible` if (D) in infeasible, and and `Unbounded` if (D) is unbounded.

`obj` an approximately optimal objective value.

`pobj` objective value of (PP).

`dobj` objective value of (DD).

`stopcode` equals 0 if the solutions to (PP) and (DD) satisfy the prescribed tolerances and equals nonzero if the solver terminated for other reasons.

Additional fields describe characteristics of the solution:

`tracex` the trace of the solution X of (P).

`r` the multiple of the identity element added to $C - \mathcal{A}^T(y)$ in the final solution to make S positive definite.

`mu` the final barrier parameter (ν).

`ynorm` the largest element of y (infinity norm).

`boundy` the bounds placed on the magnitude of each variable y.

`penalty` the penalty parameter Γ used by the solver, which must be greater than the trace of the variables X in (P). (see above).

Additional fields provide statistics from the solver:

`iterations` number of iterations used by the algorithm.

`pstep` the final step length in (PP)

`dstep` the final step length in (DD).

`pnorm` the final value $\|P(\nu)\|$.

`rho` the potential parameter (as a multiple of the total dimension of the cones).

`gaphist` a history of the duality gap.

`infhist` a history of the variable r in (DD).

`datanorm` the Frobenius norm of C, A and b.

DSDP has also provides several utility routines. The utility `derror( · )` verifies that the solution satisfies the constraints and that the objective values (P) and (D) are equal. The errors are computed according to the the standards of the DIMACS Challenge.

6 Reading SDPA Files

DSDP can be used if the user has a problem written in sparse SDPA format. These executables have been put in the directory `DSDPROOT/exec/`. The file name should follow the executable. For example,

```
> dsdp5 truss4.dat-s
```

Other options can also be used with DSDP. These should follow the SDPA filename.

`-gaptol <rtol>`	to stop the problem when the relative duality gap is less than this number.
`-mu0 <mu0>`	to specify the initial barrier parameter ν.
`-r0 <r0>`	to specify the initial value of r in (DD).
`-boundy <1e7>`	to bound the magnitude of each variable y in (DD).
`-save <filename>`	to save the solution into a file with a format similar to SDPA.
`-y0 <filename>`	to specify an initial vector y in (D).
`-maxit <iter>`	to stop the problem after a specified number of iterations.
`-rho <3>`	to set the potential parameter ρ to this multiple of the conic dimension n.
`-dobjmin <dd>`	to add a constraint that sets a lower bound on the objective value at the solution.
`-penalty <1e8>`	to set the penalty parameter Γ for infeasibility in (D).
`-print <1>`	print standard output at each k iteration.
`-bigM <0>`	treat the inequality $r \geq 0$ in (DD) as other inequalities and keep it positive.
`-dloginfo <0>`	to print more detailed output. Higher numbers produce more output.
`-dlogsummary <1>`	to print detailed timing information about each dominant computations.

7 Applying DSDP to Graph Problems

Within the directory `DSDPROOT/examples/` is a program `maxcut.c` that reads a file containing a graph, generates the semidefinite relaxation of a maximum cut problem, and solves the relaxation. For example,

```
> maxcut graph1
```

reads the graph in the file `graph1` and solves this graph problem. The first line of the graph should contain two integers. The first integer states the number of nodes in the graph, and the second integer states the number of edges. Subsequent lines have two or three entries separated by a space. The first two entries specify the two nodes that an edge connects. The optional third entry specifies the weight of the node. If no weight is specified, a weight of 1 will be assigned.

The same options that apply to reading SDPA files also apply here. Similar examples for the Lovász θ problem, maximum stable set problems, minimum graph coloring problem read also read a graph from a file, formulate the semidefinite relaxation, and solve it.

8 DSDP Subroutine Library

DSDP can also be used within a C application through a set of subroutines. There are several examples of applications that use the DSDP application program interface. Within the `DSDPROOT/examples/` directory, the file `dsdp.c` is a *mex* function that reads data from the Matlab environment, passes the data to the solver, and returns the solution. The file `readsdpa.c` reads data from a file for data in SDPA format, passes the data to the solver, and prints the solution. The files `maxcut.c` and `theta.c` read a graph, formulate a semidefinite relaxation to a combinatorial problem, and pass the data to a solver. The subroutines used in these examples are described in this chapter. Futher documentation on the routines and examples can be found in the HTML manual pages in `DSDPROOT/docs/dox/html/`.

Each of these applications includes the header file `DSDPROOT/include/dsdp5.h` and links to the library `DSDPROOT/lib/libdsdp.a`. All DSDP subroutines also return an `int` that represents an error code. A return value of zero indicates success, whereas a nonzero return value indicates that an error has occurred. The documentation of DSDP subroutines in this chapter will not show the return integer, but we highly recommend that applications check for errors after each subroutine.

8.1 Creating the Solver

To use DSDP through subroutines, first create a solver object with

```
int DSDPCreate(int m, DSDP *newsolver);
```

The first argument in this subroutine is the number of variables in the problem. The second argument should be the address of a `DSDP` variable. The `DSDP` class is defined as a pointer to an internal structure that contains the state of the solver. This subroutine will construct a new structure and point the `DSDP` variable to a new solver. A difference in typeset distinguishes the DSDP software from the `DSDP` class. Objects of this class apply the dual-scaling algorithm to the data.

Specify the objective function associated with these variables using the subroutine

```
int DSDPSetDObjective(DSDP dsdp, int i, double bi);
```

The first argument is the solver, and the second and third arguments specify a variable number and the objective value b_i associated with it. The variables are numbered 1 through m, where m is the number of variables specified in `DSDPCreate`. The objective associated with each variable must be specified individually. The default value is zero.

The next step is to provide the conic structure and data in the problem. These subroutines will be described in the next sections.

8.2 Semidefinite Cone

To specify an application with a cone of semidefinite constraints, one can use the subroutine

```
int DSDPCreateSDPCone(DSDP dsdp,int nblocks,SDPCone *newsdpcone);
```

to create a new object that describes a semidefinite cone with one or more blocks. The first argument is an existing semidefinite solver, the second argument is the number of blocks in this cone, and the final argument is an address of a `SDPCone` variable. This subroutine will allocate structures needed to specify the constraints and point the variable to this structure. Multiple cones can be created for the same solver, but it is usually more efficient to group all semidefinite blocks into the same conic structure.

All subroutines that pass data to the semidefinite cone require an `SDPCone` object in the first argument. The second argument often refers to a specific block. The blocks will be labeled from `0` to `nblocks-1`. The subroutine `int SDPConeSetBlockSize(SDPCone sdpcone, int blockj, int n)` can be used to specify the dimension of each block and the subroutine `int SDPConeSetSparsity(SDPCone sdpcone, int blockj, int nnzmat)` can be used to specify the number of nonzero matrices $A_{i,j}$ in each block. These subroutines are optional, but using them can improve error checking on the data matrices and perform a more efficient allocation of memory.

The data matrices can be specified by any of the following commands. The choice of data structures belongs to the user, and the performance of the problem depends upon this choice. In each of these subroutines, the first four arguments are an `SDPCone` object, the block number, the number of variable associated with it, and the number of rows and columns in the matrix. The blocks must be numbered consecutively, beginning with the number 0. The y variables are numbered consecutively from 1 to m. The objective matrices in (P) are specified as constraint number 0. The data passed to the `SDPCone` object will be used in the solver but not modified. The user is responsible for freeing the arrays of data it passes to `SDPCone` after solving the problem.

The square symmetric data matrices $A_{i,j}$ and C_j can be represented with a single array of numbers. DSDP supports the **s**ymmetric packed storage format. In symmetric packaged storage format, the elements of a matrix with n rows and columns are ordered as follows:

$$[\ a_{1,1} \quad a_{2,1} \quad a_{2,2} \quad a_{3,1} \quad a_{3,2} \quad a_{3,3} \quad \dots \quad a_{n,n}\ ]. \tag{10}$$

In this array $a_{k,l}$ is the element in row k and column l of the matrix. The length of this array is $n(n+1)/2$, which is the number of distinct elements on or below the diagonal. Several routines described below have an array of this length in their list of arguments. In this storage format, the element in row i and column j, where $i \geq j$, is in element $i(i-1)/2 + j - 1$ of the array.

This array can be passed to the solver by using the subroutine

```
int SDPConeSetADenseVecMat(SDPCone sdpcone,int blockj, int vari,
                           int n, double alpha, double val[], int nnz);
```

The first argument point to a semidefinite cone object, the second argument specifies the block number j, and the third argument specifies the variable i associated with it. Variables $1, \dots, m$ correspond to matrices $A_{1,j}, \dots, A_{m,j}$, whereas variable 0 corresponds to C_j. The fourth argument is the dimension (number of rows or columns) of the matrix, n. The sixth argument is the array, and the seventh argument is the length of the array. The data array will be multiplied by the scalar in the fifth argument. The application is responsible for allocating this array of data and eventually freeing it. `SDPCone` will directly access this

array in the course of the solving the problem, so it should not be freed until the solver is finished.

A matrix can be passed to the solver in sparse format by using the subroutine

```
int SDPConeSetASparseVecMat(SDPCone sdpcone,int blockj, int vari, int n,
                            double alpha, int ishift,
                            const int ind[], const double val[], int nnz);
```

In this subroutine, the first five arguments are the same as in the subroutine for dense matrices. The seventh and eighth arguments are an array an integers and an array of double-precision variables. The final argument states the length of these two arrays, which should equal the number of nonzeros in the lower triangular part of the matrix. The array of integers specifies which elements of the array (10) are included in the array of doubles. For example, the matrix

$$A_{i,j} = \begin{bmatrix} 3 & 2 & 0 \\ 2 & 0 & 6 \\ 0 & 6 & 0 \end{bmatrix} \tag{11}$$

could be inserted into the cone using one of several ways. If the first element in the `val` array is $a_{1,1}$, the first element in the `ind` array should be `0`. If the second element in the `val` array is $a_{3,2}$, then the second element in `ind` array should be 4. When the ordering of elements begins with 0, as just shown, the fifth argument `ishift` in the subroutine should be set to `0`. In general, the argument `ishift` specifies the index assigned to $a_{1,1}$. Although the relative ordering of the elements will not change, the indices assigned to them will range from `ishift` to `ishift` $+ n(n+1)/2 - 1$. Many applications, for instance, prefer to index the array from 1 to $n(n+1)/2$, setting the `index` argument to 1. The matrix (11) can be set in the block j and variable i of the semidefinite cone by using one of the routines

```
SDPConeSetASparseVecMat(sdpcone,j,i,3,1.0,0,ind1,val1,3);
SDPConeSetASparseVecMat(sdpcone,j,i,3,1.0,1,ind2,val2,3);
SDPConeSetASparseVecMat(sdpcone,j,i,3,1.0,3,ind3,val3,4);
SDPConeSetASparseVecMat(sdpcone,j,i,3,0.5,0,ind4,val4,3);
```

where

```
ind1 = [ 0  1  4 ];      val1 = [ 3  2  6 ];
ind2 = [ 1  2  5 ];      val2 = [ 3  2  6 ];
ind3 = [ 7  3  5  4 ];   val3 = [ 6  3  0  2 ];
ind4 = [ 0  1  4 ];      val4 = [ 6  4  12 ];
```

As these examples suggest, there are many other ways to represent the sparse matrix. The nonzeros in the matrix do not have to be ordered, but ordering them may improve the efficiency of the solver. `SDPCone` assumes that all matrices $A_{i,j}$ and C_j that are not explicitly defined and passed to the `SDPCone` structure will equal the zero matrix. Furthermore, there exist routines `SDPConeAddASparseVecMat` and `SDPConeAddADenseVecMat` that can be used to write a constraint matrix as a sum of multiple matrices. The arguments to these functions match those of the corresponding `SDPConeSetASparseVecMat` routines.

To check whether the matrix passed into the cone matches the one intended, one can use the subroutine

```
int SDPConeViewDataMatrix(SDPCone sdpcone, int blockj, int vari);
```

to print the matrix to the screen. The output prints the row and column numbers, indexed from 0 to $n-1$, of each nonzero element in the matrix. The subroutine `int SDPConeView(SDPCone sdpcone,int blockj)` can be used to view all of the matrices in a block.

After the `DSDP` solver has been applied to the data and the solution matrix X_j have been computed (see `DSDPComputeX`), the matrix can be accessed by using the command

```
int SDPConeGetXArray(SDPCone sdpcone, int blockj, double *xmat[], int *nn);
```

The third argument is the address of a pointer that will be set to the array containing the solution. The integer whose address is passed in the fourth argument will be set to the length of this array, $n(n+1)/2$, for the packed symmetric storage format. Since the X solutions are usually fully dense, no sparse representation is provided. These arrays were allocated by the `SDPCone` object during `DSDPSetup`, and the memory will be freed by the `DSDP` solver object when it is destroyed. The array used to store X_j could be overwritten by other operations on the `SDPCone` object. The command

```
int SDPConeComputeX(SDPCone sdpcone, int blockj, int n,
                    double xmat[], int nn);
```

recomputes the matrix X_j and places it into the array specified in the fourth argument. The length of this array is the fifth argument and the dimension of the block in the third argument. The vectors y and Δy needed to compute the matrices X_j are stored internally in `SDPCone` object. The subroutine

```
int SDPConeViewX(SDPCone sdpcone, int blockj, int n, double xmat[], int nn);
```

can be used to print this matrix to standard output.

The dimension of each block can be found by using the routine

```
int SDPConeGetBlockSize(SDPCone sdpcone, int blockj, int *n);
```

where the second argument is the block number and the third argument is the address of an integer variable.

The inner product of X_j with C_j, $A_{i,j}$, and I_j can be computed by using the routine

```
int SDPConeAddADotX(SDPCone sdpcone,int blockj, double alpha,
                    double xmat[], int nn, double adotx[], int mp2);
```

The second argument specifies which block to use, and the third argument is a scalar that will be multiplied by the inner products. The fourth argument is the array containing X_j, and the fifth argument is the length of the array. The sixth argument is an array of length $m+2$, and the seventh argument should equal $m+2$, where m is the number of variables in y. This routine will add `alpha` times $\langle C_j, X_j \rangle$ to the initial element of the array, `alpha` times $\langle A_{i,j}, X_j \rangle$ to element i of the array, and `alpha` times $\langle I_j, X_j \rangle$ to last element of the array.

The matrix S in (D) can be computed and copied into an array by using the command

```
int SDPConeComputeS(SDPCone sdpcone, int blockj, double c, double y[], int m,
                    double r, int n, double smat[], int nn);
```

The second argument specifies which block to use, and the fourth argument is an array containing the variables y. The third argument is the multiple of C to be added, and the sixth argument is the multiple of the identity matrix to be added. The sixth argument is the dimension of the block, and the seventh argument is an array for S whose length is given in the eighth argument.

Special support for combinatorial applications also exists. In particular, the subroutines

```
int SDPConeComputeXV(SDPCone sdpcone, int blockj, int *dpsdefinite);
int SDPConeXVMultiply(SDPCone sdpcone, int blockj,
                      double v1[], double v2[], int n);
int SDPConeAddXVAV(SDPCone sdpcone, int blockj, double v[], int n,
                   double vav[], int mp2);
```

support the use of randomized algorithms for combinatorial optimization. The first routine computes a matrix V_j such that $X = V_j V_j^T$. The second routine computes the matrix-vector product $w = V_j v$, where w and v are vectors of dimension equal to the dimension of the block. The third routine computes the vector-matrix-vector product $v^T A_{i,j} v$ for C, $A_{1,j}, \ldots, A_{m,j}$ and the identity matrix. The length of the array in the fifth argument is $m + 2$ and should be set in the final argument. In these applications, use of the routine `DSDPConeComputeX` may not be necessary if the full matrix is not required.

The memory required for the X_j matrix can be significant for large problems. If the application has an array of double-precision variables of length $n(n+1)/2$ available for use by the solver, the subroutine

```
int SDPConeSetXArray(SDPCone sdpcone,int blockj, int n,
                     double xmat[], int nn);
```

can be used to pass it to the cone. The second argument specifies the block number whose solution will be placed into the array `xmat`. The third argument is the dimension of the block. The dimension specified in the fifth argument `nn` refers to the length of the array. The `SDPCone` object will use this array as a buffer for its computations and store the solution X in this array at its termination. The application is responsible for freeing this array after the solution has been found.

DSDP also supports the **s**ymmetric full storage format. In symmetric full storage format, an $n \times n$ matrix is stored in an array of n^2 elements in row major order. That is, the elements of a matrix with n rows and columns are ordered

$$[\ a_{1,1}\ \ 0\ \ \ldots\ \ 0\ \ a_{2,1}\ \ a_{2,2}\ \ 0\ \ \ldots\ \ 0\ \ \ \ldots\ \ \ a_{n,1}\ \ \ldots\ \ a_{n,n}\ ]. \tag{12}$$

The length of this array is $n \times n$. Early versions of DSDP5 used the packed format exclusively, but this format has been added because it is more convenient for some applications. The routines

```
int SDPConeSetStorageFormat(SDPCone sdpcone,int blockj, char UPLQ);
int SDPConeGetStorageFormat(SDPCone sdpcone, int blockj, char *UPLQ);
```

set and get the storage format for a block, respectively. The second argument specifies the block number and the third argument should be `'P'` for packed storage format or `'U'` for full storage format. The default value is `'P'`. These storage formats correspond to the LAPACK storage formats for the upper half matrices in column major ordering. All of the above commands apply to the symmetric full storage format. One difference in their use, however, is that the size of the arrays in the arguments should be $n \times n$ instead of $n \times (n+1)/2$. With the symmetric full storage format, if the first element in the `val` array is $a_{1,1}$, the first element in the `ind` array should be `0`. If the second element in the `val` array is $a_{3,2}$, then the second element in `ind` array should be 8. The matrix (11) can be set in the block j and variable i of the semidefinite cone by using the routine

```
int SDPConeSetASparseVecMat(sdpcone,j,i,3,1.0,0,ind1,val1,3);
```

where

$$\texttt{ind1} = [\ 0 \quad 3 \quad 7\ ]; \quad \texttt{val1} = [\ 3 \quad 2 \quad 6\ ];\ .$$

8.3 LP Cone

To specify an application with a cone of linear scalar inequalities, one can use the subroutine

```
int DSDPCreateLPCone( DSDP dsdp, LPCone *newlpcone);
```

to create a new object that describes a cone with 1 or more linear scalar inequalities. The first argument is an existing `DSDP` object, and the second argument is the address of an `LPCone` variable. This subroutine will allocate structures needed to specify the constraints and point the variable to this structure. Multiple cones for these inequalities can be created for the same `DSDP` object, but it is usually more efficient to group all inequalities of this type into the same structure. All subroutines that pass data to the LP cone require an `LPCone` object in the first argument.

A list of n linear inequalities in (D) is passed to the object in sparse column format. The vector $c \in \mathbb{R}^n$ should be considered an additional column of the data. (In the formulation (P), the data A and c are represented in sparse row format.)

The data is passed to the `LPCone` using the subroutine

```
int LPConeSetData(LPCone lpcone, int n,
                  const int nnzin[], const int row[], const double aval[]);
```

In this case, the integer array `nnzin` has length $m+2$, and begins with 0, and `nnzin[i+1]-nnzin[i]` equals the number of nonzeros in column i ($i = 0, \ldots, m$) (or row i of A). The length of the second and third array equals the number of nonzeros in A and c. The arrays contain the nonzeros and the associated row numbers of each element (or column numbers in A). The first column contains the elements of c, the second column contains the elements corresponding to y_1, and the last column contains elements corresponding to y_m.

For example, consider the following problem in the form of (D):

$$\begin{array}{lrcrl} \text{Maximize} & y_1 & + & y_2 & \\ \text{Subject to} & 4y_1 & + & 2y_2 & \leq 6 \\ & 3y_1 & + & 7y_2 & \leq 10 \\ & & & -y_2 & \leq 12 \end{array}$$

This example has three inequalities, so the dimension of the x vector would be 3 and $n = 3$. The input arrays would be as follows:

$$\begin{array}{ll} \texttt{nnzin} & = \left[\begin{array}{cccc} 0 & 3 & 5 & 7 \end{array} \right] \\ \texttt{row} & = \left[\begin{array}{cccccccc} 0 & 1 & 2 & 0 & 1 & 0 & 1 & 2 \end{array} \right] \\ \texttt{aval} & = \left[\begin{array}{cccccccc} 6.0 & 10.0 & 12.0 & 4.0 & 3.0 & 2.0 & 7.0 & -1.0 \end{array} \right] \end{array}$$

An example of the use of this subroutine can be seen in the `DSDPROOT/examples/readsdpa.c`.

If it is more convenient to specify the vector c in the last column, consider using the subroutine:

```
int LPConeSetData2(LPCone lpcone,int n,
               const int ik[],const int cols[],const double vals[]);
```

This input is also sparse column input, but the c column comes last. In this form the input arrays would be as follows:

$$\begin{array}{ll} \texttt{nnzin} & = \left[\begin{array}{cccc} 0 & 2 & 5 & 7 \end{array} \right] \\ \texttt{row} & = \left[\begin{array}{cccccccc} 0 & 1 & 0 & 1 & 2 & 0 & 1 & 2 \end{array} \right] \\ \texttt{aval} & = \left[\begin{array}{cccccccc} 4.0 & 3.0 & 2.0 & 7.0 & -1.0 & 6.0 & 10.0 & 12.0 \end{array} \right] \end{array}$$

This subroutine is used in the DSDP Matlab mex function, which can be used as an example.

The subroutines

```
int LPConeView(LPCone lpcone);
int LPConeView2(LPCone lpcone);
```

can be used to view the data that has been set and verify the correctness of the data. Multiple `LPCone` structures can be used, but for efficiency, it is often better to include all linear inequalities in a single cone.

The variables s in (D) and x in (P) can be found by using the subroutines

```
int LPConeGetXArray(LPCone lpcone,double *xout[], int *n);
int LPConeGetSArray(LPCone lpcone,double *sout[], int *n);
```

In these subroutines, the second argument sets a pointer to an an array of doubles containing the variables, and the integer in the third argument is set to the length of this array. These array are allocated by the `LPCone` object, and the memory is freed when the `DSDP` object is destroyed. Alternatively, the application can give the `LPCone` an array in which to put the solution x of (P). This array should be passed to the cone by using the following subroutine:

```
int LPConeSetXVec(LPCone lpcone,double xout[], int n);
```

At completion of the `DSDP` solver, the solution x will be copied into this array `xout`, which must have length n. The slack variables s may be scaled. To get the unscaled vector, pass an array of appropriate length into the object using `int LPConeGetSArray2(LPCone lpcone, double s[], int n)`. This subroutine will copy the slack variables into the array.

A special type of semidefinite and LP cone contains only simple bounds on the variables y. Corresponding to lower and upper bounds on the y variables are surplus and slack variables in (P) with a cost. The subroutine

```
int DSDPCreateBCone(DSDP dsdp, BCone *bcone);
```

will create this cone from an existing `DSDP` object and point the `BCone` variable to the new structure. The bounds on the variables can be set by using the subroutines

```
int BConeSetLowerBound(BCone bcone, int vari, double yi);
int BConeSetUpperBound(BCone bcone, int vari, double yi);
```

The first argument is the conic object, the second argument identifies a variable from 1 to m, and the third argument is the bound. Here m is the total number of variables in the vector y. For applications using the formulation (P), the subroutines

```
int BConeSetPSurplusVariable(BCone bcone, int vari);
int BConeSetPSlackVariable(BCone bcone, int vari);
```

may be a more convenient way to represent an inequality in (P). These commands are equivalent to setting a lower or upper bound on a variable y_i to zero. To improve the memory allocation process, the application may use the subroutines `int BConeAllocateBounds( BCone bcone, int nbounds);` to tell the object how many bounds will be specified. This subroutine is optional, but it may improve the efficiency of the memory allocation. The subroutine `int BConeSetXArray(BCone, double xout[], int n)` will set an array in the cone where the cone will copy the variables x. To view the bounds, the application may use the subroutine `int BConeView(BCone bcone)`. To retrieve the dual variables x corresponding to these constraints, one can use the subroutine `int BConeCopyX(BCone bcone, double xl[], double xu[], int m)`. The second and third arguments of this routine are an array of length m, the fourth argument. This routine will set the values of this array to the value of the corresponding variable. When no bound is present, the variable will equal zero.

In some applications it may be useful to fix a variable to a number. Instead of modeling this constraint as a pair of linear inequalities, fixed variables can be passed directly to the solver by using the subroutine

```
int DSDPSetFixedVariables(DSDP dsdp, double vars[], double vals[],
                          double x[], int n);
```

In this subroutine, the array of variables in the second argument is set to the values in the array of the third argument. The fourth argument is an optional array in which the solver will put the sensitivities to these fixed variables. The final argument is the length of the arrays. Note that the values in the second argument are integer numbers from 1 to m represented in double precision. Again, the integers should be one of $1, \ldots, m$. Alternatively, a single variable can be set to a value by using the subroutine `int DSDPFixVariable(DSDP dsdp, int vari, double val)`.

8.4 Applying the Solver

After setting the data associated with the constraint cones, `DSDP` allocates internal data structures and factors the data in the subroutine

```
int DSDPSetup(DSDP dsdp);
```

This subroutine factors the data, creates a Schur complement matrix with the appropriate sparsity, and allocates additional resources for the solver. This subroutine should be called after setting the data but before solving the problem. Furthermore, it should be called only once for each `DSDP` object. On very large problems, insufficient memory on the computer may be encountered in this subroutine, so the error code should be checked. The subroutine

```
int DSDPSetStandardMonitor(DSDP dsdp, int k);
```

will tell the solver to print the objective values and other information at each k iteration to standard output. The subroutine `int DSDPLogInfoAllow(int,0)` will print even more information if the first argument is positive.

The subroutine

```
int DSDPSolve(DSDP dsdp);
```

attempts to solve the problem. This subroutine can be called more than once. For instance, the user may try solving the problem by using different initial points.

After solving the problem, the subroutine

```
int DSDPComputeX(DSDP dsdp);
```

can be called to compute the variables X in (P). These computations are not performed within the solver because these variables are not needed to compute the step direction.

Each solver created should be destroyed with the command

```
int DSDPDestroy(DSDP dsdp);
```

This subroutine frees the work arrays and data structures allocated by the solver.

8.5 Convergence Criteria

Convergence of the `DSDP` solver may be defined by using several options. The precision of the solution can be set by using the subroutine

```
int DSDPSetGapTolerance(DSDP dsdp, double rgaptol);
```

The solver will terminate if there is a sufficiently feasible solution such that the difference between the objective values in (DD) and (PP), divided by the sum of their absolute values, is less than the prescribed number. A tolerance of `0.001` provides roughly three digits of accuracy, whereas a tolerance of `1.0e-5` provides roughly five digits of accuracy. The subroutine

```
int DSDPSetMaxIts(DSDP dsdp, int maxits);
```

specifies the maximum number of iterations. The subroutine `int DSDPSetDualBound( DSDP, double)` specifies an upper bound on the objective value in (D). The algorithm will terminate when it finds a point when the variable r in (DD) is less than the prescribed tolerance and the objective value in (DD) is greater than this number.

8.6 Detecting Infeasibility

Infeasibility in either (P) or (D) can be determined by using the subroutine

```
int DSDPGetSolutionType(DSDP dsdp, DSDPSolutionType *pdfeasible);
```

This command sets the second argument to an enumerated type. There are four types for `DSDPSolutionType`:

- The type `DSDP_PDFEASIBLE` means that both (D) and (P) have feasible solutions and their objective values are bounded.
- The type `DSDP_UNBOUNDED` means that (D) is unbounded and (P) is infeasible. This type applies when the variable $r \leq \epsilon_r$ and $\|\mathcal{A}X - b\|_\infty / trace(X) > \epsilon_P$. In this case, at least one variable y_i will be near its bound. The subroutine `DSDPSetYBounds` can adjust these bounds if the user thinks that they are not big enough to permit feasibility. Large bounds may create numerical difficulties in the solver, but they may also permit feasible solutions and improve the quality of the certificate of infeasibility. Normalizing the vector y will provide an approximate certificate of infeasibility for (P).
- The type `DSDP_INFEASIBLE` means that (D) is infeasible and (P) is unbounded. This type applies when the variable $r > \epsilon_r$ and $\|\mathcal{A}X - b\|_\infty / trace(X) \leq \epsilon_P$. In this case, the trace of the variables X in (P) will be near the bound Γ. The subroutines `DSDPSetPenaltyParameter` can adjust Γ if the user thinks that it is too small, A larger parameter may create numerical difficulties in the solver, but it may also improve the quality of the certificate of infeasibility. Normalizing these variables so that to have a trace of 1.0 will provide an approximate certificate of infeasibility.
- The type `DSDP_PDUNKNOWN` means DSDP was unable to determine feasibility in either solution. This type applies when the initial point for (DD) was infeasible or if the bounds on y appear to be too small to permit a feasible solution.

The tolerance ϵ_r can be set by using the subroutine `int DSDPSetRTolerance(DSDP, double)`. The tolerance ϵ_P can be set using the subroutine `int DSDPSetPTolerance(DSDP, double)`. The subroutines `int DSDPGetRTolerance(DSDP, double*)` and `int DSDPGetPTolerance( DSDP, double*)` can be used to get the current tolerances.

8.7 Solutions and Statistics

The objective values in (PP) and (DD) can be retrieved by using the commands

```
int DSDPGetPPObjective(DSDP dsdp, double *pobj);
int DSDPGetDDObjective(DSDP dsdp, double *dobj);
```

The second argument in these routines is the address of a double-precision variable.

The solution vector y can be viewed by using the command

```
int DSDPGetY(DSDP dsdp, double y[], int m);
```

The user passes an array of size m where m is the number of variables in the problem. This subroutine will copy the solution into this array.

The success of DSDP can be interpreted with the command

```
int DSDPStopReason(DSDP dsdp, DSDPTerminationReason *reason);
```

This command sets the second argument to an enumerated type. The various reasons for termination are listed below.

DSDP_CONVERGED	The solutions to (PP) and (DD) satisfy the convergence criteria.
DSDP_MAX_IT	The solver applied the maximum number of iterations without finding a solution.
DSDP_INFEASIBLE_START	The initial point in (DD) was infeasible.
DSDP_INDEFINITE_SCHUR	Numerical issues created an indefinite Schur matrix that prevented further progress.
DSDP_SMALL_STEPS	Small step sizes prevented further progress.
DSDP_NUMERICAL_ERROR	Numerical issues prevented further progress.

The subroutines

```
int DSDPGetBarrierParameter(DSDP dsdp, double *mu);
int DSDPGetR(DSDP dsdp, double *r);
int DSDPGetStepLengths(DSDP dsdp, double *pstep, double *dstep);
int DSDPGetPnorm(DSDP dsdp, double *pnorm);
```

provide more information about the current solution. The subroutines obtain the barrier parameter, the variable r in (DD), the step lengths in (PP) and (DD), and a distance to the central path at the current iteration.

A history of information about the convergence of the solver can be obtained with the commands

```
int DSDPGetGapHistory(DSDP dsdp, double gaphistory[], int history);
int DSDPGetRHistory(DSDP dsdp, double rhistory[], int history);
```

These subroutines retrieve the history of the duality gap and the variable r in (DD) for up to 100 iterations. The user passes an array of double-precision variables and the length of this array. The subroutine

```
int DSDPGetTraceX(DSDP dsdp, double *tracex);
```

gets the trace of the solution X in (P). Recall that the penalty parameter must exceed this quantity in order to return a feasible solution from an infeasible starting point. The subroutine

```
int DSDPEventLogSummary(void)
```

will print a summary of time spent in each cone and many of the primary computational subroutines.

8.8 Improving Performance

The performance of the DSDP may be *significantly* improved with the proper selection of bounds, parameters, and initial point.

The application may specify an initial vector y to (D), a multiple of the identity matrix to make the initial matrix S positive definite, and an initial barrier parameter. The subroutine `int DSDPSetY0(DSDP dsdp, int vari, double yi0)` can specify the initial value of the variable y_i. Like the objective function in (D), the variables are labeled from 1 to m. By default the initial values of y equal 0. Since convergence of the algorithm depends on the proximity of the point to the central path, initial points can be difficult to determine. Nonetheless, the subroutine

```
int DSDPSetR0(DSDP dsdp, double r0);
```

will set the initial value of r in (DD). If `r0` < 0, a default value of `r0` will be chosen. If S^0 is not positive definite, the solver will terminate with an appropriate termination flag. The default value is usually very large ($1e10$), but smaller values can *significantly* improve performance.

The subroutine `int DSDPSetPotentialParameter(DSDP dsdp, double rho)` sets the potential parameter ρ. This parameter must be greater than 1. The default value is 4.0, but larger values such as 5 or 10 can significantly improve performance. Feasibility in (D) is enforced by means of a penalty parameter. By default it is set to $10e8$, but other values can affect the convergence of the algorithm. This parameter can be set by using `int DSDPSetPenaltyParameter(DSDP dsdp, double Gamma)`, where `Gamma` is the large positive penalty parameter Γ. This parameter must exceed the trace of the solution X in order to return a feasible solution from an infeasible starting point. The subroutine `int DSDPUsePenalty(DSDP dsdp,int yesorno)` is used to modify the algorithm. By default, the value is 0. A positive value means that the variable r in (DD) should be kept positive, treated like other inequalities, and penalized with the parameter `Gamma`. The subroutine `int DSDPSetZBar(DSDP dsdp, double zbar)` sets an initial upper bound on the objective value at the solution. This value corresponds to the objective value of any feasible point of (PP). The subroutine `int DSDPSetBarrierParameter(DSDP dsdp, double mu0)` sets the initial barrier parameter. The default heuristic is very robust, but performance can get generally be improved by providing a smaller value.

DSDP applies the same Schur complement matrix for multiple linear systems. This feature often reduces the number of iterations and improves robustness. The cost of each iteration increases, especially when the dimension of the semidefinite blocks is of similar dimension to or larger than the number of variables y. The subroutine

```
int SDPConeSetParameter(SDPCone sdpcone, DSDP dsdp);
```

sets an appropriate parameter based upon the dimension of the semidefinite blocks. To manually set this parameter, the subroutine `int DSDPReuseMatrix(DSDP dsdp, int reuse)` can set a maximum on the number of times the Schur complement matrix is reused. The default value is 4, although the Matlab mex function and SDPA file reader set this parameter between 0 and 15 depending on the size of the semidefinite blocks and the number of variables y. Applications whose semidefinite blocks are small relative to the number of variables y should probably use larger values, while applications whose semidefinite blocks have

size equal to or greater than the number of variables should probably set this parameter to zero.

The convergence of the dual-scaling algorithm assumes the existence of a strict interior in both (P) and (D). The use of a penalty parameter can add an interior to (D). An interior to (P) can be created by bounding the variables y. Default bounds of `-1e7` and `1e7` have been set, but applications may change these bounds by using the subroutine

```
int DSDPSetYBounds(DSDP dsdp, double minbound, double maxbound);
```

The second argument should be a negative number that is a lower bound of each variable y_i, and the third argument is an upper bound of each variable. These bounds should not be tight. If one of the variables nearly equals the bound at the solution, the solver will return a termination code saying (D) is unbounded. To remove these bounds, set both the lower and upper bound to zero.

8.9 Iteration Monitor

A standard monitor that prints out the objective value and other relevant information at the current iterate can be set by using the command

```
int DSDPSetStandardMonitor(DSDP dsdp, int k);
```

A user can write a customized subroutine of the form

```
int (*monitor)(DSDP dsdp,void* ctx);
```

This subroutine will be called from the `DSDP` solver each iteration. It is useful for writing a specialized convergence criteria or monitoring the progress of the solver. The objective value and other information can be retrieved from the solver by using the commands in the section 8.7. To set this subroutine, use the command

```
int DSDPSetMonitor(DSDP dsdp, int (*monitor)(DSDP,void*), void* ctx);
```

In this subroutine, the first argument is the solver, the second argument is the monitoring subroutine, and the third argument will be passed as the second argument in the monitoring subroutine. Examples of two monitors can be found in `DSDPROOT/src/solver/dsdpconverge.c`. The first monitor prints the solver statistics at each iteration, and the second monitor determines the convergence of the solver. A monitor can also be used to print the time, duality gap, and potential function at each iteration. Monitors have also been used to stop the solver after a specified time limit and change the parameters in the solver.

9 PDSDP

The DSDP package can also be run in parallel on multiple processors. In the parallel version, the Schur complement matrix is computed and solved in parallel. The parallel Cholesky factorization in PDSDP is performed by using SCALAPACK. The parallel Cholesky factorization in SCALAPACK uses a two-dimensional block cyclic structure to distribute the data. The blocking parameter in SCALAPACK determines how many rows and columns are in each block. Larger block sizes can be faster and reduce the overhead of passing messages, but smaller block sizes balance the work among the processors more equitably. PDSDP used a blocking parameter of 32 after experimenting with several choices. Since SCALAPACK uses a dense matrix structure, this version is not appropriate when the Schur complement matrix is sparse.

The following steps should be used to run an application in parallel using PDSDP.

1. Install DSDP. Edit `DSDPROOT/make.include` to set the appropriate compiler flags.

2. Install SCALAPACK. This package contains parallel linear solvers and is freely available to the public.

3. Go to the directory `DSDPROOT/src/pdsdp/scalapack/` and edit `Makefile` to identify the location of the DSDP and SCALAPACK libraries.

4. Compile the PDSDP file `pdsdpscalapack.c`, which implements the additional operations. Then compile the executable `readsdpa.c`, which will read an SDPA file.

A PDSDP executable can be used much like the serial version of DSDP that reads SDPA files. Given a SDPA file such as `truss1.dat-s`, the command

```
mpirun -np 2 dsdp5  truss1.dat-s -log_summary
```

will solve the problem using two processors. Additional processors may also be used. This implementation is best suited for very large problems.

Use of PDSDP as a subroutine library is also similar to the use of the serial version of the solver. The application must create the solver and conic object on each processor and provide each processor with a copy of the data matrices, objective vector, and options. At the end of the algorithm, each solver has a copy of the solution. The routines to set the data and retrieve the solution are the same.

The few differences between the serial and parallel version are listed below.

1. All PDSDP programs must include the header file:

   ```
   #include pdsdp5scalapack.h
   ```

2. Parallel applications should link to the DSDP library, the SCALAPACK library, and the compiled source code in `dsdpscalapack.c`. Linking to the BLAS and LAPACK libraries is usually included while linking to SCALAPACK.

3. The application should initialize and finalize MPI.

4. After creating the DSDP solver object, the application should call

```
int PDSDPUseSCALAPACKLinearSolver(DSDP dsdp);
```

5. The monitor should be set on only one processor, for example:

```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank==0) info = DSDPSetStandardMonitor(dsdp);
```

An example of the usage is provided in DSDPROOT/pdsdp/ScaLAPACK/readsdpa.c. Scalability of medium- and large-scale problems has been achieved on up to 64 processors. The corrector direction does not scale well, so consider reducing the number of these steps for large groups of processors. See [1] for more details.

Source code that uses the parallel conjugate gradient method in PETSc to solve the linear systems is also included in the distribution.

10 Data Structures and Parameters

The DSDP solver computes $\Delta' y$, $\Delta'' y$, $\Delta_\nu y$, $\|P(\nu)\|$, $\langle X(\nu), S\rangle$, and other quantities using operations on vectors, the Schur matrix, and the cones. Vector operations include sums and inner products. Operations on the Schur matrix include inserting elements and factoring the matrix. Objects representing a cone implement routines for computing its dual matrix S from y, evaluating the logarithmic barrier function, computing $\mathcal{A}S^{-1}$, and computing M. The solver object computes M, for example, by calling the corresponding operations on each cone and summing the results. The solver computes $\Delta' y$ through calls to the Schur matrix object that can factor the matrix and solve systems of linear equations. Table 2 shows eight of the primary data structures used in DSDP, the operations they implement, and the other objects required to implement those operations. For dense matrix structures, DSDP uses BLAS and LAPACK to operate on the data.

The LP cone and SDP cone objects implement the same interface for cones, but the implementation of these operations depends on whether the cone is a semidefinite cone, linear programming cone, or another type. The solver structure operates on the cone objects without knowing whether it is an SDP cone or another type. DSDP uses opaque pointers and function pointers to achieve polymorphic behavior.

Table 3 summarizes the significant parameters, options, and default values. The most important of these options is the initial variable r. By default, DSDP selects a value much larger than required to make $S \in \hat{K}$. Computational experience indicates that large values are more robust than smaller values. DSDP then sets the initial values of $\bar{z} = 1e10$ and $\nu = (\bar{z} - b^T y + \Gamma r)/(n\rho_n)$. Users can manually set $\bar{z}$ and ν, but choices better than the defaults usually require insight into the solution. The number of corrector steps can also significantly improve performance. In some examples, corrector steps can reduce the number of iterations by half—although the impact in total computation time is not as significant. Computational experience suggests that the number of corrector steps should be between 0 and 12, and the time spent in these steps should not exceed 30%.

Table 2: Summary of primary data structures and their functionality.

Solver: Implements an algorithm for linear and semidefinite programming. *Operations:* $\Delta' y$, $\Delta'' y$, $\Delta_\nu y$, $\Delta^c y$, $\|P(\nu)\|$, $\langle X(\nu), S\rangle$, reduce ν. *Implementations:* Dual-Scaling Algorithm. *Instances:* one. *Requires:* Vector, Cone, Schur.
Vector: Represents y, $\mathcal{A}S^{-1}$, b, $\Delta' y$, $\Delta'' y$, $\Delta_\nu y$, and other internal work vectors. *Operations:* sum, inner product, norm. *Implementations:* dense one-dimensional array. *Instances:* about two dozen.
Schur: Represents M, the Schur complement of Newton equations. *Operations:* add to row, add to diagonal, factor, solve, vector-multiply. *Implementations:* sparse, dense, parallel dense. *Instances:* one.
Cone: Represents data C and A_i. *Operations:* check if $S \leftarrow C - \mathcal{A}^* y \in K$, $\ln\det S$, $\mathcal{A}S^{-1}$, M, $X(\nu)$. *Implementations:* SDP Cone, LP Cone, Bounds on y, Variable $r \geq 0$. *Instances:* three or more. *Requires:* Vector, Schur. *SDP Cone requires:* V Matrix, Data Matrices, S Matrix, DS Matrix.
SDP V Matrix: Represents X, $S^{-1}A_i S^{-1}$, and a buffer for $C - \mathcal{A}^* y$. *Operations:* $V \leftarrow 0$, $V \leftarrow V + \gamma w w^T$, get array. *Implementations:* dense. *Instances:* one per block.
SDP Data Matrix: Represents a symmetric data matrix. *Operations:* $V \leftarrow V + \gamma A$, $\langle V, A\rangle$, $w^T A w$, get rank, get eigenvalue/vector. *Implementations:* sparse, dense, identity, low-rank. *Instances:* up to $m + 1$ per block.
SDP S Matrix: Represents S and checks whether $X(\nu) \succ 0$. *Operations:* $S \leftarrow V$, Cholesky factor, forward solve, backward solve, determinant. *Implementations:* sparse, dense. *Instances:* two per block.
SDP DS Matrix: Represents ΔS. *Operations:* $\Delta S \leftarrow V$, $w \leftarrow \Delta S v$. *Implementations:* dense, sparse, diagonal. *Instances:* one per block.

Table 3: Summary of important parameters and initial values.

r: Dual infeasibility. *Default:* heuristic (large) *Suggested Values:* $10^2 - 10^{12}$ *Comments:* Larger values ensure robustness, but smaller values can signficantly improve performance.
$\mathbf{y}$: Initial solution. *Default:* 0 *Suggested Values:* Depends on data *Comments:* Initial points that improve performance can be difficult to find.
ρ_n: Bound ρ above by $n \times \rho_n$ and influence the barrier parameter. *Default:* 3.0 *Suggested Values:* $2.0 - 5.0$ *Comments:* Smaller values ensure robustness, but larger values can significantly improve performance.
kk_{max} : Maximum number of corrector steps. *Default:* 4 *Suggested Values:* $0 - 15$ *Comments:* For relatively small block sizes, increase this parameter.
Γ: The penalty parameter r and the bound on the trace of X. *Default:* $1e8$. *Suggested Values:* $10^3 - 10^{15}$ *Comments:* Larger values suitable unless (D) is feasible but has no interior.
$\mathbf{l}$,u: Bounds on the variables y. *Default:* $-10^7, 10^7$ *Suggested Values:* Depends on the data. *Comments:* Tighter bounds do not necessarily improve performance.
$\bar{z}$: Upper bound on (D). *Default:* 10^{10} *Suggested Values:* Depends on the data. *Comments:* A high bound is usually sufficient.
ν: Dual barrier parameter. *Default:* Heuristic *Suggested Values:* Depends on the current solution. *Comments:* The default method sets $\nu = (\bar{z} - b^T y)/\rho$.
k_{max} : Maximum number of dual-scaling iterations. *Default:* 200 *Suggested Value:* $50 - 500$ *Comments:* Iteration counts of 20-60 are common.
η: Terminate when $(\bar{z} - b^T y)/(\lvert b^T y \rvert + 1)$ is less than η. *Default:* 10^{-6} *Suggested Values:* $10^{-2} - 10^{-7}$ *Comments:* Many problems do not require high accuracy.
ρ : Either a dynamic of a fixed value can be used. *Default:* Dynamic. *Suggested Values:* Dynamic *Comments:* The fixed strategy sets $\rho = n \times \rho_n$ and $\nu = (\bar{z} - b^T y)/\rho$, but its performance is usually inferior to the dynamic strategy.
ϵ_r, ϵ_P: Classify solutions as feasible. *Default:* 10^{-8}, 10^{-4} *Suggested Values:* $10^{-2} - 10^{-10}$ *Comments:* Adjust if the scaling of the problem is poor.

11 Previous Versions

DSDP began as a specialized solver for combinatorial optimization problems. Over the years, improvements in efficiency and design have enabled its use in many applications.

1997 At the University of Iowa the initial version of DSDPwas released. It solved the semidefinite relaxations of the maximum cut problem [3].

1999 DSDP version 2 increased functionality to address semidefinite cones with rank-one constraint matrices and LP constraints. It was used specifically for combinatorial problems such as minimum bisection, graph coloring, stable sets, and bound-constrained quadratic problems.

2000 DSDP version 3 was a preliminary implementation of a general purpose SDP solver that addressed applications from the Seventh DIMACS Implementation Challenge on Semidefinite and Related Optimization Problems. It ran in serial and parallel.

2002 DSDP version 4 added new sparse data structures to improve efficiency and precision. A Lanczos based line search and efficient iterative solver were added. It solved all problems in the SDPLIB collection that includes examples from control theory, truss topology design, and relaxations of combinatorial problems.

2004 DSDP version 5 [2] featured new data structures for semidefinite constraints, a corrector direction, and extensibility to structured applications in conic programming. Existence of the central path was ensured by bounding the variables.

Acknowledgments

We thank Xiong Zhang and Cris Choi for their help in developing this code. Xiong Zhang, in particular, was fundamental to the initial version of DSDP. We also thank Hans Mittelmann [4] for his efforts in testing and benchmarking the different versions of the code. We thank all of the users who have commented on previous releases and suggested improvements to the software. Their contributions have made DSDP a more reliable, robust, and efficient package.

References

[1] Steven J. Benson. Parallel computing on semidefinite programs. Technical Report ANL/MCS-P939-0302, Mathematics and Computer Science Division, Argonne National Laboratory, March 2003.

[2] Steven J. Benson and Yinyu Ye. DSDP5: Software for semidefinite programming. Preprint ANL/MCS-P1289-0905, Mathematics and Computer Science Division, Argonne National Laboratory, September 2005.

[3] Steven J. Benson, Yinyu Ye, and Xiong Zhang. Solving large-scale sparse semidefinite programs for combinatorial optimization. *SIAM Journal on Optimization*, 10(2):443–461, 2000.

[4] Hans D. Mittelmann. Benchmarks for optimization software, 2005. ftp://plato.asu.edu/ftp/{sdplib.txt,sparse_sdp.txt,dimacs.txt}.