

# DSDP5 User Guide – The Dual-Scaling Algorithm for Semidefinite Programming

Steven J. Benson  
Mathematics and Computer Science Division  
Argonne National Laboratory  
Argonne, IL U.S.A.  
<http://www.mcs.anl.gov/~benson>

Yinyu Ye  
Department of Management Science and Engineering  
Stanford University  
Stanford, CA U.S.A.  
<http://www.stanford.edu/~yyye>

Technical Report ANL/MCS-TM-255

November 3, 2004

## Abstract

DSDP is an implementation of the dual-scaling algorithm for conic programming. The source code, written entirely in ANSI C, is freely available. The solver can be used as a subroutine library, 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:

- a robust algorithm with a convergence proof and polynomial complexity under mild assumptions on the data,
- feasible primal and dual solutions,
- low memory requirements compared to other interior-point methods,
- ability to exploit sparsity in the data,
- extensible data structures that allow applications to customize the solver and improve its performance,
- a subroutine library that enables it to be used in larger applications,
- scalable performance for large problem on parallel architectures,
- well documented interface.

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.

## Contents

<b>1</b>	<b>Conic Programming</b>	<b>1</b>
<b>2</b>	<b>Dual-Scaling Algorithm</b>	<b>2</b>
<b>3</b>	<b>Standard Output</b>	<b>3</b>
<b>4</b>	<b>DSDP with MATLAB</b>	<b>3</b>
4.1	Semidefinite Cones . . . . .	3
4.2	LP Cones . . . . .	5
4.3	Solver Options . . . . .	5
4.4	Solver Performance and Statistics . . . . .	6
<b>5</b>	<b>Reading SDPA files</b>	<b>7</b>
<b>6</b>	<b>Applying DSDP to Graph Problems</b>	<b>8</b>
<b>7</b>	<b>DSDP Subroutine Library</b>	<b>9</b>
7.1	Creating the Solver . . . . .	9
7.2	Semidefinite Cone . . . . .	9
7.3	LP Cone . . . . .	12
7.4	Fixed Variables . . . . .	13
7.5	Applying the Solver . . . . .	13
7.6	Convergence Criteria . . . . .	14
7.7	Solver Status . . . . .	14
7.8	Improving Performance . . . . .	15
7.9	Standard Monitors . . . . .	16
7.10	Custom Monitors . . . . .	16
<b>8</b>	<b>PDSDP</b>	<b>16</b>
<b>9</b>	<b>A Brief History</b>	<b>18</b>
<b>10</b>	<b>Referencing DSDP</b>	<b>18</b>
<b>11</b>	<b>Acknowledgments</b>	<b>19</b>
<b>12</b>	<b>Copyright</b>	<b>20</b>

## 1 Conic Programming

The DSDP package uses a dual-scaling algorithm to solve conic optimization problems of the form

$$(P) \quad \text{minimize} \quad \sum_{j=1}^{n_b} \langle C_j, X_j \rangle \quad \text{subject to} \quad \sum_{j=1}^{n_b} \langle A_{i,j}, X_j \rangle = b_i, \quad i = 1, \dots, m, \quad X_j \in K_j.$$

$$(D) \quad \text{maximize} \quad \sum_{i=1}^m b_i y_i \quad \text{subject to} \quad \sum_{i=1}^m A_{i,j} y_i + S_j = C_j, \quad j = 1, \dots, n_b, \quad S_j \in K_j,$$

where  $K_j$  is a cone,  $\langle \cdot, \cdot \rangle$  is the associated inner product, and  $b_i, y_i$  are scalars. For semidefinite programming each cone  $K_j$  is a set of symmetric positive definite matrices, the data  $A_{i,j}$  and  $C_j$  are symmetric matrices of the same dimension, and the inner product  $\langle C, X \rangle := \text{trace } C^T X = \sum_{k,l} C_{k,l} X_{k,l}$ . In linear programming the cone  $K$  is the nonnegative orthant ( $\mathbb{R}_+^n$ ), the data  $A_i$  and  $C$  are vectors of the same dimension, and the inner product  $\langle C, X \rangle$  is the usual vector inner product.

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 constraints are called feasible, while the others are called infeasible.

Provided (P) and (D) both have a feasible point and there exist a strictly feasible point to either them, (P) and (D) have solutions and their objective values are equal. To satisfy these assumptions, DSDP bounds the dual variables  $y$  such that  $l \leq y \leq u$  where  $l, u \in \mathbb{R}^m$ . Furthermore, it uses an auxiliary variable  $r$  that represent the dual infeasibility of (D) and associates it with a penalty parameter  $\Gamma$ . The use of the bounds and penalty parameter also allows DSDP to apply a feasible-point algorithm even when  $y$  and  $S$  are infeasible.

The standard form used by DSDP is given by the following pair of problems:

$$(PP) \quad \text{minimize} \quad \sum_{j=1}^{n_b} \langle C_j, X_j \rangle + u^T x^u - l^T x^l$$

$$\text{subject to} \quad \sum_{j=1}^{n_b} \langle A_{i,j}, X_j \rangle + x_i^u - x_i^l = b_i, \quad i = 1, \dots, m,$$

$$\sum_{j=1}^{n_b} \langle I_j, X_j \rangle \leq \Gamma$$

$$X_j \in K_j, \quad x^u, x^l \geq 0,$$

$$(DD) \quad \text{maximize} \quad \sum_{i=1}^m b_i y_i - \Gamma r$$

$$\text{subject to} \quad C_j - \sum_{i=1}^m A_{i,j} y_i + r I_j = S_j \in K_j, \quad j = 1, \dots, n_b,$$

$$l_i \leq y_i \leq u_i, \quad i = 1, \dots, m,$$

$$r \geq 0,$$

where  $I_j$  is the identity element of  $K_j$ , and  $x^l, x^u \in \mathbb{R}^m$  correspond to the dual values of the lower and upper bounds on  $y$ , respectively. Note that (PP) and (DD) can be expressed in the form of (P) and (D). By default, the bounds and the penalty parameter  $\Gamma$  are very large so that a change to the solutions on problems with a bounded solution sets is unlikely. Unbounded and infeasible solutions to either (P) or (D) are determined through examination of the solutions to (PP) and (DD).

## 2 Dual-Scaling Algorithm

This chapter summarizes the dual-scaling algorithm for conic optimization. For simplicity, the notation will refer to semidefinite programming, but the algorithm can be extended to other cones.

Let  $K$  be the cone of positive semidefinite matrices. The data  $C, A_i \in \mathbb{R}^{n \times n}$  are given symmetric matrices,  $b \in \mathbb{R}^m$  is a given vector, and the variables  $X, S$  reside in the cone  $K$  of symmetric positive semidefinite matrices. Following conventional notation, let

$$\mathcal{A}X = \begin{bmatrix} \langle A_1, X \rangle \\ \vdots \\ \langle A_m, X \rangle \end{bmatrix} \quad \text{and} \quad \mathcal{A}^T y = \sum_{i=1}^m A_i y_i,$$

Given  $X, S \in K$  a dual point  $(y, S)$  such that  $\mathcal{A}^T y + S = C$  and  $S \succ 0$ , and a barrier parameter  $\hat{\mu} > 0$ , each iteration of the dual-scaling algorithm computes a step direction  $\Delta y$  by solving the linear system

$$\begin{pmatrix} \langle A_1, S^{-1} A_1 S^{-1} \rangle & \cdots & \langle A_1, S^{-1} A_m S^{-1} \rangle \\ \vdots & \ddots & \vdots \\ \langle A_m, S^{-1} A_1 S^{-1} \rangle & \cdots & \langle A_m, S^{-1} A_m S^{-1} \rangle \end{pmatrix} \Delta y = \frac{1}{\hat{\mu}} b - \mathcal{A}(S^{-1}) \quad (1)$$

For arbitrary feasible  $X$ , this linear system (1) can be derived by taking the Schur complement of the equations

$$\mathcal{A}(X + \Delta X) = b \quad \mathcal{A}^T(\Delta y) + \Delta S = 0 \quad \hat{\mu} S^{-1} \Delta S S^{-1} + \Delta X = \hat{\mu} S^{-1} - X, \quad (2)$$

which are the Newton equations for the nonlinear system

$$\mathcal{A}X = b \quad \mathcal{A}^T y + S = C \quad \hat{\mu} S^{-1} = X, \quad (3)$$

at a point  $(X, y, S)$  such that  $\mathcal{A}X = b$ , and for a feasible dual point the step direction is equivalent to the Newton step to maximize the function

$$\phi(y) = b^T y + \hat{\mu} \ln \det S \quad (4)$$

subject to  $\mathcal{A}^T y + S = C$  and minimizes the dual potential function

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

over a trust region [7]. In this dual potential function  $\bar{z} = \langle C, X \rangle$  for a feasible matrix  $X$ ,  $\rho > n + \sqrt{n}$ , and  $\hat{\mu} = \frac{\bar{z} - b^T y}{\rho}$ . The relationships between these derivations can be found in [11].

The algorithm then selects a step size  $\beta_k \in (0, 1]$  such that  $y^{k+1} = y^k + \beta_k \Delta y$  and  $S^{k+1} = S^k + \beta_k \Delta S \succ 0$ . Using the dual step direction and (2), the primal matrix

$$X(S, \hat{\mu}) = \hat{\mu} S^{-1} - \hat{\mu} S^{-1} \Delta S S^{-1} \quad (5)$$

and satisfies the constraints  $\mathcal{A}X(S, \hat{\mu}) = b$ .

Given a feasible dual starting point and appropriate choices for step-length and  $\hat{\mu}$ , convergence results in [7] show that either the new dual point  $(y, S)$  or the new primal point  $X$  is feasible and reduces the Tanabe-Todd-Ye primal-dual potential function

$$\Psi(X, S) = \rho \ln(X \bullet S) - \ln \det X - \ln \det S$$

enough to achieve linear convergence.

A more thorough explanation of the the dual-scaling algorithm and its convergence properties, can be found in [7].

### 3 Standard Output

The progress of the DSDP solver can be monitored using standard output printed to the screen. The following is an example output from a small random problem.

Iter	Primal	Dual	DInfeas	Mu	StepLength	Pnrm
0	1.00000000e+10	0.00000000e+00	1.0e+02	1.0e+02	0.00 0.00	0.00
1	0.00000000e+00	-2.01399998e+02	0.0e+00	1.0e+02	1.00 0.00	1e+14
2	8.96197978e+00	-3.63540418e+00	0.0e+00	6.3e+00	1.00 0.00	1e+09
3	8.37939680e+00	6.37939828e+00	0.0e+00	1.0e-00	1.00 0.00	1e+09
4	7.01692844e+00	6.96891411e+00	0.0e+00	2.4e-02	1.00 0.00	2e+09
5	7.00019863e+00	6.99844570e+00	0.0e+00	8.8e-04	1.00 0.00	2e+09
6	7.00000053e+00	6.99992229e+00	0.0e+00	3.9e-05	1.00 0.00	2e+09
7	7.00000000e+00	6.99999611e+00	0.0e+00	1.9e-06	1.00 0.00	2e+09
8	7.00000000e+00	6.99999981e+00	0.0e+00	9.7e-08	1.00 0.00	2e+09

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

<b>Iter</b>	the current iteration number
<b>Primal</b>	the current estimate of the primal objective function (P)
<b>Dual</b>	the current dual objective function (D)
<b>DInfeas</b>	the infeasibility in the current dual solution. This number is the multiple of the identity matrix that has been added to the dual matrix
<b>Mu</b>	the current barrier parameter. This parameter decreases to zero as the points get closer to the solution
<b>StepLength</b>	the multiple of the primal and dual step-directions used
<b>Pnrm</b>	The proximity to a point on the central path: $\ S^{.5}XS^{.5} - \hat{\mu}I\ $

## 4 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 dual objective vector 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.

### 4.1 Semidefinite Cones

If the  $j$ th cone 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  $n$ . The third element in this row of the cell array is a sparse matrix with  $n(n+1)/2$  rows and  $m+1$  columns.

Columns 1 to  $m$  of this matrix represent the constraints  $A_{1,j}, \dots, A_{m,j}$  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  $\text{AC}\{j, 3\}$  through the operator  $\text{dvec}(\cdot) : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}^{n(n+1)/2}$ , which is defined as

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

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  $\text{dvec}(\cdot)$  is  $\text{dmat}(\cdot) : \mathbb{R}^{n(n+1)/2} \rightarrow \mathbb{R}^{n \times n}$ , which converts the vector into a square symmetric matrix. Using these operations,

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

and

$$\text{AC}\{j, 3\} = [ \text{dvec}(A_{1,j}) \ \dots \ \text{dvec}(A_{m,j}) \ \text{dvec}(C_j) ];$$

For example, the problem:

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

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} = 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 case,  $X = [ 3 \times 1 \ \text{double} ]$ ,  $X\{1\}' = [ 1.0 \ 1.0 \ 1.0 ]$ , and  $\text{dmat}(X\{1\}) = [ 1 \ 1 ; 1 \ 1 ]$ .

Each semidefinite block can be stated in a separate cone 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 cone in the cell array. 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 while the number of columns remains constant. The following lines indicate how to group the semidefinite blocks in rows 1 and 2 of cell array  $\text{AC1}$  into a new cell array  $\text{AC2}$

```
> 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  $\text{AC2}$  can be passed directly into  $\text{DSDP}$ . 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  $\text{DSDP}$  measured by execution time, will change very little.

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

## 4.2 LP Cones

A cone of LP variables can be 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.

## 4.3 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 affect the formulation of the problem are:

- r0** multiple of the identity matrix added to the initial dual matrix.  $S^0 = C - \sum A_i y_i^0 + r^0 * I$ . If  $r0 < 0$ , a dynamic selection will be used. IF the solver does not converge, increase this number by an order of magnitude. IMPORTANT: To improve robustness and convergence, TRY this option [default -1].
- zbar** an upper bound on the objective value at the solution [default 1.0e10].
- penalty** penalty parameter used to enforce dual feasibility ( $>0$ ). IMPORTANT: This parameter can significantly influence performance. This parameter must be greater than the trace of the primal solution  $X$ . [default 1e8].
- boundy** bound the magnitude of the variables  $y$ . The convergence of this solver assumes the solution set of these variables is bounded. Try this option if convergence is not achieved [1e6].

Fields in the OPTIONS structure that affect the stopping criteria for the solver are:

- 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 400].
- infctol** tolerance for stopping because of suspicion of dual infeasibility [default 1e-8].
- stepctol** tolerance for stopping because of small steps [default 1e-2].

**dual\_bound** a bound for the dual solution. The solver stops when a feasible dual iterate with an objective greater than this value is found. (Helpful in brach-and-bound algorithms.) [default 1e+30].

Fields in the `OPTIONS` structure that affect printing are:

**print** *k*, if want to display result in each *k* iteration, else = 0 [default 10].

**logtime** =1 to profile DSDP subroutines, else =0. (Assumes proper compilation flags.)

**dloginfo** = positive to print too much information about the solution process. (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 recognized in `OPTIONS` structure are:

**fastblas** to use LAPACK for factorizations wherever possible. Set to 1 if Atlas or other fast BLAS routines are used in Matlab. [default 0].

**bigM** if  $> 0$ , the dual infeasibility  $r$  with remain positive and be the inequality  $r > 0$  will be treated like other inequalities. [default 0].

**dobjmin** to add a constraint that places a lower bound on the objective value at the solution.

**maxtrustradius** maximum trust radius used in the solver.

**rho** multiple of problem dimension used in potential function. [default: 2]

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

**mu0** initial barrier parameter. This parameter can significantly affect the performance of the solver. Smaller number can improve performance, but are generally less robust.

For instance, the commands

```
> OPTIONS.gaptol = 0.001;
> OPTIONS.penalty = 10000;
> [STAT,y,X] = DSDP(b,AC,OPTIONS);
```

asks for a solution with approximately three significant digits using a penalty parameter of 10000.

Using a fourth input argument, the command

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

specifies an initial dual vector  $y_0$ . The default starting vector is the zero vector.

#### 4.4 Solver Performance and Statistics

The second and third output arguments return the dual (D) and primal (P) solutions, respectively.

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

<code>obj</code>	an approximately optimal objective value
<code>primal</code>	an approximately optimal objective value if convergence to the solution was detected, <code>infeasible</code> if primal infeasibility is suspected, and <code>unbounded</code> if dual infeasibility is suspected.
<code>dual</code>	an approximately optimal objective value if convergence to the solution was detected, <code>infeasible</code> if dual infeasibility is suspected, and <code>unbounded</code> if primal infeasibility is suspected.

Additional fields describe characteristics of the solution:

<code>tracex</code>	the trace of the primal solution.
<code>r</code>	the multiple of the identity matrix added to $C - \mathcal{A}^T(y)$ in the final solution to make $S$ positive definite.
<code>mu</code>	the final barrier parameter.
<code>ynorm</code>	the largest element of $y$ (infinity norm).
<code>boundy</code>	the bounds placed on the magnitude of each variable $y$ .
<code>penalty</code>	the penalty parameter used by the solver, which must be greater than the trace of the primal solution (see above).

Additional fields provide statistics from the solver:

<code>iterations</code>	number of iterations used by the algorithm.
<code>pstep</code>	the final primal step length.
<code>dstep</code>	the final dual step length.
<code>pnorm</code>	the final norm from the targeted located on the central path.
<code>gaphist</code>	a history of the duality gap.
<code>infhist</code>	a history of the dual infeasibility.
<code>termcode</code>	0 solution found, 1: primal infeasible, 2: dual infeasible, -3: Numerical issues prevented further progress.
<code>datanorm</code>	the Frobenius norm of $C$ , $A$ and $b$ .

DSDP has also provides several utility routines. The utility `error(·)` verifies that the solution satisfies the constraints and that the objective values (P) and (D) are equal. The errors are computed according to the standards of the DIMACS Challenge[8].

## 5 Reading SDPA files

DSDP can also be run without the MATLAB environment 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.
  - r0 <r0> to add this multiple of the identity matrix to the initial dual solution to make it feasible.
  - boundy <1e6> to bound the magnitude of each dual variable  $y$ .
  - save <filename> to save the solution into a file with a format similar to SDPA.
  - y0 <filename> to specify an initial dual vector.
  - maxit <iter> to stop the problem after a specified number of iterations.
  - 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 for infeasibility in (D).
  - bigM <0> treat dual infeasibility as other inequalities.
  - benchmark <filename> to run many problems listed in a file.
  - dloginfo <0> to print more detailed output. Higher number produce more output.
  - dlogsummary <0> to print detailed timing information about each dominant computations.

## 6 Applying DSDP to Graph Problems

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

```
> maxcut graph1
```

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.

A similar program reads a graph from a file, formulates a minimum bisection problem or Lovasz theta problem, and solves it. For example,

```
> theta graph1
```

reads the graph in the file `graph1` and solves this graph problem.

## 7 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 API. 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 DSDP solver, and returns the solution. The file `readsdp.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 DSDP solver. The routines used in these examples are described in this chapter.

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, while a nonzero return value indicates that an error has occurred. The documentation of DSDP routines in this chapter will not show the return integer, but we highly recommend that applications check for errors after each routine.

### 7.1 Creating the Solver

To use DSDP through subroutine, the solver object must first be created with the command

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

The first argument in this routine is the number of variables in the problem. The second argument should be the address of a DSDP variable. This routine will allocate memory for the solver and set the address of the DSDP variable to a new solver object. A difference in typeset distinguishes the DSDP package from the DSDP solver object.

The objective function associated with these variables should be specified using the routine

```
DSDPSetDualObjective(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. A constant may also be added to the objective function. The routine `DSDPAddObjectiveConstant(DSDP dsdp, double obj)` will add such a constant to the objective function. This constant is algorithmically irrelevant, but it makes the objective values displayed by the solver more consistent with the underlying application.

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

### 7.2 Semidefinite Cone

To specify an application with a cone of semidefinite constraints, the routine

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

can be used to create a new object that describes a semidefinite cone with 1 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 routine 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 blocks into the same conic structure.

All routines that pass data to the semidefinite cone use this `SDPCone` variable in the first argument. The second argument often refers to a specific block. The blocks will be labeled from 0 to `nblocks-1`. The routine `SDPConeSetBlockSize(SDPCone sdpcone, int blockj, int n)` can be used to specify the dimension of each block and the routine `SDPConeSetSparsity(SDPCone sdpcone, int blockj, int nnzmat)` can be used to specify the number of nonzero matrices  $A_{i,j}$  in each block. These routines 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 is up to the user, and the performance of the problem depends upon this choice of data structures. In each of these routines, the first four arguments are a pointer to the `SDPCone` object, the block number, and 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 primal objective matrices are specified as constraint number 0. The data that is 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 DSDP after solving the problem.

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

$$[ a_{1,1} \ a_{2,1} \ a_{2,2} \ a_{3,1} \ a_{3,2} \ a_{3,3}, \ \dots, \ a_{n,n} ]. \quad (6)$$

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. This array can be passed to the solver using the routine

```
SDPConeSetDenseVecMat(SDPCone sdpcone,int blockj, int vari,
                      int n, const 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  $A_{1,j}, \dots, A_{m,j}$  while variable 0 corresponds to  $C_j$ . The fourth argument is the dimension (number of rows or columns) of the matrix,  $n$ . The fifth argument is the array, and sixth argument is the length of the array ( $n(n+1)/2$ ). The application is responsible for allocating this array of data and eventually freeing it. DSDP 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 using the routine

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

In this routine, the first four arguments are the same as in the routine for dense matrices. The sixth and seventh 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 (6) are included in the array of doubles. For example 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 routine 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.

For example, the matrix

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

could be inserted into the cone using any of these the following ways:

```
ierr=SDPConeSetSparseVecMat(sdpcone,j,i,3,0,ind1,val1,3);
ierr=SDPConeSetSparseVecMat(sdpcone,j,i,3,1,ind2,val2,3);
ierr=SDPConeSetSparseVecMat(sdpcone,j,i,3,3,ind3,val3,4);
```

where

$$\begin{aligned} \text{ind1} &= [ 0 & 1 & 4 ] & \text{val1} &= [ 3 & 2 & 6 ] \\ \text{ind2} &= [ 1 & 2 & 5 ] & \text{val2} &= [ 3 & 2 & 6 ] \\ \text{ind3} &= [ 7 & 3 & 5 & 4 ] & \text{val3} &= [ 6 & 3 & 0 & 2 ] \end{aligned}$$

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. DSDP 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.

To check whether the matrix passed into the cone matches the one intended, the routine

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

can be used to print out the 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 routine SDPConeView(SDPCone sdpcone,int blockj) can be used to view all of the matrices in a block.

After the dual-scaling algorithm has been applied to the data, the solution matrix  $X_j$  can be found using the command

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

The integer in the fourth argument will be set the the dimension of the matrix, and the third argument will be set the the array containing the solution. The array has length  $n(n+1)/2$ , and it represents the matrix as in (6). Since the  $X$  solutions are usually fully dense, no sparse representation is provided. These arrays were allocated by the SDPCone object and the memory will be freed with the DSDP solver object is destroyed.

The array used to store  $X_j$  could be overwritten by other operations on the SDPCone object. The command,

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

recomputes the matrix  $X_j$  and places it into the array specified in the third argument. This array should have length  $n(n+1)/2$ , where  $n$  is the size of the block and the fourth argument. The vectors  $y$  and  $\Delta y$  needed to compute  $X$  are stored internally in SDPCone object.

The dual matrix  $S$  can be computed and copied into an array using the command

```
SDPConeComputeS(SDPCone sdpcone, int blockj, double y[], int nvars, double smat[], int n);
```

The second argument specifies which block to use, the third argument is an array containing the variables  $y$ . The fifth argument is an array of length  $n(n+1)/2$ , where  $n$  is the size of the block and the sixth argument.

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 routine

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

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 dimension specified in the fourth argument `n` refers to the number of rows and columns in the matrix, and not the length of the array. The DSDP solver 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.

### 7.3 LP Cone

To specify an application with a cone of linear scalar inequalities, the routine

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

can be used to create a new object that describes a cone with 1 or more linear scalar inequalities. The first argument is an existing DSDP solver and the second argument is the address of an `LPCone` variable. This routine 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 solver, but it is usually more efficient to group all inequalities of this type into the same structure.

All routines that pass data to the LP cone use this `LPCone` variable in the first argument. The data in  $c$  and the operator  $A : \mathbb{R}^n \rightarrow \mathbb{R}^m$  should be specified in sparse row format. The vector  $c$  should be considered an additional row of  $A$ . Alternatively, the data can be seen as the transpose of  $A$  and  $c$  in sparse column format.

Pass the data to the `LPCone` using the routine:

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

In this case, the integer array `nnzin` has length  $m + 2$ , begins with 0, and `nnzin[i + 1] - nnzin[i]` equals the number of nonzeros in row  $i$  of  $A$  (or column of  $A^T$ ). 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 column numbers in  $A$  (or row numbers in  $A^T$ ).

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

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

In this example, there two inequalities, so the dimension of the  $x$  vector would be 2 and  $n = 2$ . The input arrays would be as follows:

$$\begin{array}{ll} n & = 2 \\ \text{nnzin} & = [ 0 \ 2 \ 4 \ 6 ] \\ \text{row} & = [ 0 \ 1 \ 0 \ 1 \ 0 \ 1 ] \\ \text{aval} & = [ 4.0 \ 3.0 \ 2.0 \ 7.0 \ 6.0 \ 10.0 ] \end{array}$$

The routine

```
LPConeView(LPCone lpcone)
```

can be used to view the data that has been set. Multiple `LPCone` structure can be used, but for efficiency purposes, it is often better to include them all in one cone.

The values of slack variables  $s$  and primal variables  $x$  can be found using the routines

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

In these routines, the second argument sets a pointer to an array of doubles containing the variables and the integer in the third argument will be set to the length of this array. These array were allocated by the LPCone object and the memory will be freed with the DSDP solver object is destroyed. Alternatively, the application can give the LPCone an array in which to put the primal solution  $x$ . This array should be passed to the cone using the following routine

```
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  $n$ .

## 7.4 Fixed Variables

Fixing variables to a particular value may be necessary in some applications. These constraints do not represent a cone, and should be specified directly to the DSDP solver.

```
DSDPFixVariable(DSDP dsdp, int vari, double val);
```

Again, the integer should be one of  $1, \dots, m$ .

## 7.5 Applying the Solver

After setting the data associated with the constraint cones, DSDP must allocate internal data structures and factor the data in the routine

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

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

The routine

```
DSDPSetStandardMonitor(DSDP dsdp)
```

will tell the solver to print the objective values and other information at each iteration to standard output.

The command

```
DSDPSolve(DSDP dsdp)
```

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

Each solver created should be destroyed with the command

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

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

## 7.6 Convergence Criteria

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

```
DSDPSetGapTolerance(DSDP, double);
```

The solver will terminate if there is a sufficiently feasible solution such that the difference between the primal and dual objective values, normalized by the sum of the magnitudes of the primal and dual objective values, is less than the prescribed number. A tolerance of 0.001 provides roughly three digits of accuracy, while a tolerance of  $1.0e-5$  provides roughly five digits of accuracy. The routine

```
DSDPSetMaxIts(DSDP, int)
```

specifies the maximum number of iterations. The routine `DSDPSetRTolerance(DSDP, double)` specifies how small the dual infeasibility constant  $r$  must be to be an approximate solution, and the routine `DSDPSetDualBound(DSDP, double)` specifies an upper bound on the dual solution. The algorithm will terminate when it finds a point whose dual infeasibility is less than the prescribed tolerance and whose dual objective value is greater than this number.

## 7.7 Solver Status

The primal and dual objective values can be retrieved using the commands

```
DSDPGetPrimalObjective(DSDP dsdp, double *pobj);
DSDPGetDualObjective(DSDP dsdp, double *dobj);
```

The dual dual objective value is usually more accurate is recommended as the solution.

The dual solution vector  $y$  can be viewed by using the command

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

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

The success of DSDP can be interpreted with the command

```
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 primal and dual solution that satisfies the convergence criteria.
DSDP_UNBOUNDED	The dual problem (D) is unbounded and (P) is infeasible.
DSDP_INFEASIBLE	The dual problem (D) is infeasible and (P) is unbounded
DSDP_MAX_IT	The solver applied the maximum number of iterations without finding solution.
DSDP_INFEASIBLE_START	The initial dual point was infeasible.

`DSDP_INDEFINITE_SCHUR` Numerical issues created an indefinite Schur matrix that prevented the further progress.

`DSDP_SMALL_STEPS` Small step sizes prevented further progress.

The routines

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

provide more information about the current solution. The routines obtain the barrier parameter, dual infeasibility, primal and dual step lengths, 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

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

retrieve the history of the duality gap and the dual infeasibility for up to 100 iterations. The user passes an array of double precision variables and the length of this array. The routine

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

gets the trace of the primal solution  $X$ . Recall that the penalty parameter must exceed this quantity in order to return a feasible solution from an infeasible starting point. The routine

```
DSDPEventLogSummary(void)
```

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

## 7.8 Improving Performance

The performance of the DSDP may be improved with the proper selection of parameters and initial point. In particular, the application may specify an initial dual vector  $y$ , a multiple of the identity matrix to make the initial dual matrix positive definite, and an initial barrier parameter. The routine `DSDPSetY0(DSDP dsdp, int vari, double yi0)` can specify the initial value of the variable  $y_i$ . Like the dual objective, the variables are labelled from 1 to  $m$ . By default the initial values of  $y$  equal 0. The routine `DSDPSetR0(DSDP dsdp, double r0)` will set the multiple of the identity matrix and routine. More specifically, it specifies a positive number  $r$  and sets  $S^0 = C - \sum A_i y_i^0 + r0 * I$ , where  $I$  is the identity matrix. If  $r0 < 0$ , a default value will of  $r0$  will be chosen. If  $S^0$  is not positive definite, the solver will terminate will an appropriate termination flag.. The routine `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 (P).

Dual feasibility is forced 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 through the routine `DSDPSetPenaltyParameter(DSDP dsdp, double M)`, where  $M$  is the large positive penalty parameter. This parameter must exceed the trace of the primal solution  $X$  in order to return a feasible solution from an infeasible starting point. The routine `DSDPUsePenalty(DSDP dsdp, int yesorno)` is used to modify the algorithm. By default, the value is 0. A positive value means that the dual infeasibility should be treated as a cone, kept positive, and penalized with the specified value. The routine `DSDPSetInitialBarrierParameter(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.

The convergence of the dual-scaling algorithm assumes the existence of a strict interior in both the primal and dual problem. The use of a “big M” penalty parameter can add an interior to the dual problem (D). An interior to the primal (P) can be created by bounding the dual variables  $y$ . If lower and upper bounds on these variables can be determined beforehand, they may be set using the routine

```
DSDPBoundDualVariables(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. Even if the solution set for  $y$  is bounded, and bounds have not already been incorporated into the model, the developers suggest trying this parameter! It can significantly improve the performance of the solver. However, if one of the variables nearly equals the bound at the solution, the solver will return a termination code saying dual problem is unbounded.

## 7.9 Standard Monitors

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

```
DSDPSetStandardMonitor(DSDP dsdp);
```

## 7.10 Custom Monitors

A user can write a customized routine of the form

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

This routine 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 using the commands in the section 7.7. To set this routine, use the command

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

In this routine, the first argument is the solver, the second argument is the monitoring routine, and the third argument will be passed as the second argument in the monitoring routine. 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.

## 8 PDSDP

The DSDP package can also be run in parallel using multiple processors. In the parallel version, the Schur complement matrix is computed and solved in parallel. The parallel Cholesky factorization in PDSDP is performed using PLAPACK[10]. The parallel Cholesky factorization in PLAPACK uses a two-dimensional block cyclic structure to distribute the data. The blocking parameter in PLAPACK 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 PLAPACK 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 PLAPACK. This package contains parallel linear solvers and is freely available to the public.
3. Go to the directory **DSDPROOT/src/pdsdp/plapack/** and edit **Makefile** to identify the location of the DSDP and PLAPACK libraries.
4. Compile the PDSDP file **pdsdpplapack.c**, which implements the additional operations. Then compile the executable **readsdp.c**, which will read an SDPA file.

A PDSDP executable can be used much like 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 very 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 pdsdp5plapack.h
```

2. Parallel applications should link to the DSDP library, the PLAPACK library, and the compiled source code in **pdsdpplapack.c**. Linking to the BLAS and LAPACK libraries are usually included while linking to PLAPACK.
3. The application should initialize and finalize MPI.
4. After creating the DSDP solver object, the application should call

```
int PDSDPUsePLAPACKLinearSolver(DSDP dsdp, MPI_Comm comm);
```

Most applications can set the variable **comm** to **MPI\_COMM\_WORLD**.

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/plapack/readsdp.c**. Scalability of medium and large-scale problems has been achieved on up to 64 processors. See [2] for more details.

Source code that uses the parallel conjugate gradient method in PETSc[1] to solve the linear systems is also included in the distribution.

## 9 A Brief History

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. Its success has resulted in hundreds of citations in research journals. Below is a brief history of DSDP.

1997 At the University of Iowa the authors release the initial version of DSDP. It solved the semidefinite relaxations of the maximum cut, minimum bisection, s-t cut, and bound constrained quadratic problems[7].

1999 DSDP version 2 increased functionality to address semidefinite cones with rank-one constraint matrices and LP constraints [6]. It was used extensively for combinatorial problems such as graph coloring, stable sets[3], and satisfiability problems.

2000 DSDP version 3 was a general purpose SDP solver that addressed large-scale applications included in the the Seventh DIMACS Implementation Challenge on Semidefinite and Related Optimization Problems [8]. DSDP 3 also featured the initial release of PSDSP[2], the first parallel solver for semidefinite programming.

2002 DSDP version 4 added new sparse data structures and linked to BLAS and LAPACK to improve efficiency and precision[5]. A Lanczos based line search and efficient iterative solver were added. It solved all problems in the SDPLIB collection that includes applications of control theory, truss topology design, and relaxations of combinatorial problems [4].

2004 DSDP version 5 features a new efficient interface for semidefinite constraints, and extensibility to structured applications in conic programming. Existence of the central path was ensured by bounding the variables. New applications from in computational chemistry, global optimization, and sensor network location motivated the improvements in efficiency in robustness.

## 10 Referencing DSDP

DSDP is provided free of charge. We ask those who use it to cite the appropriate references in their reports and publications. The following information can be used to cite DSDP.

```
@Article{byz1,
  author = "Steven J. Benson and Yinyu Ye and Xiong Zhang",
  title = "Solving Large-Scale Sparse Semidefinite Programs for Combinatorial Optimization",
  journal = "SIAM Journal on Optimization",
  volume = 10,
  number = 2,
  year = 2000,
  pages = "443--461",
}
```

```
@TechReport{dsdp-user-manual,
  author = "Steven J. Benson and Yinyu. Ye",
  title = "DSDP5 User Manual",
  institution = "Mathematics and Computer Science Division, Argonne National Laboratory",
  number = "ANL/MCS-TM-255",
  month = "March",
  year = "2004",
  note = "\url{http://www.mcs.anl.gov/~benson/dsdp}"
```

```
@TechReport{pdsdp,
  author = "Steven J. Benson",
  title = "Parallel Computing on Semidefinite Programs",
```

```
institution = 'Mathematics and Computer Science Division, Argonne National Laboratory',
number = 'ANL/MCS-P939-0302',
month = 'March',
year = 2003,
location = 'ftp://info.mcs.anl.gov/pub/tech_reports/reports/P939.pdf',
}
```

## 11 Acknowledgments

We thank Xiong Zhang and Cris Choi for their help in developing this code. Xiong Zhang, in particular, made an enormous contribution in the initial version of DSDP. We also thank Hans Mittelmann[9] for his efforts in testing and benchmarking it. Finally, we thank all of the users who have commented on previous releases. Their contributions have made DSDP a more reliable, robust, and efficient package.

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

## References

- [1] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.2.0, Argonne National Laboratory, 2004. <http://www.mcs.anl.gov/petsc>.
- [2] S. J. Benson. Parallel computing on semidefinite programs. Technical Report ANL/MCS-P939-0302, Mathematics and Computer Science Division, Argonne National Laboratory, March 2003.
- [3] S. J. Benson and Y. Ye. Approximating maximum stable set and minimum graph coloring problems with the positive semidefinite relaxation. In *Applications and Algorithms of Complementarity*, volume 50 of *Applied Optimization*, pages 1–18. Kluwer Academic Publishers, 2000.
- [4] S. J. Benson and Y. Ye. DSDP3: Dual-scaling algorithm for general positive semidefinite programming. Technical Report ANL/MCS-P851-1000, Mathematics and Computer Science Division, Argonne National Laboratory, February 2001.
- [5] S. J. Benson and Y. Ye. DSDP4: A software package implementing the dual-scaling algorithm for semidefinite programming. Technical Report ANL/MCS-TM-255, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, June 2002.
- [6] S. J. Benson, Y. Ye, and X. Zhang. Mixed linear and semidefinite programming for combinatorial and quadratic optimization. *Optimization Methods and Software*, 11:515–544, 1999.
- [7] S. J. Benson, Y. Ye, and X. Zhang. Solving large-scale sparse semidefinite programs for combinatorial optimization. *SIAM Journal on Optimization*, 10(2):443–461, 2000.
- [8] DIMACS. The Seventh DIMACS Implementation Challenge: 1999-2000. <http://dimacs.rutgers.edu/Challenges/Seventh/>, 1999.
- [9] Hans D. Mittelmann. SDPLIB benchmarks. <ftp://plato.asu.edu/pub/sdplib.txt>, 2003.
- [10] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. Scientific and Engineering Computing. MIT Press, Cambridge, MA, 1997. <http://www.cs.utexas.edu/users/plapack>.
- [11] Y. Ye. *Interior Point Algorithms: Theory and Analysis*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, New York, 1997.

## 12 Copyright

The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

## Index

API, 9

block structure, 9, 10

bounds, 1, 5, 8, 16

convergence, 14–16

DSDP, 9

DSDPAddObjectiveConstant(), 9

DSDPBoundVariables(), 16

DSDPCreate(), 9

DSDPCreateLPCone(), 12

DSDPCreateSDPCone(), 9

DSDPDestroy(), 13

DSDPEventLogSummary(), 15

DSDPFixVariable(), 13

DSDPGetBarrierParameter(), 15

DSDPGetDualObjective(), 14

DSDPGetGapHistory(), 15

DSDPGetPnorm(), 15

DSDPGetPrimalObjective(), 14

DSDPGetR(), 15

DSDPGetRHistory(), 15

DSDPGetStepLengths(), 15

DSDPGetTraceX(), 15

DSDPGetY(), 14

DSDPSetDualBound(), 14

DSDPSetDualObjective(), 9

DSDPSetGapTolerance(), 14

DSDPSetInitialBarrierParameter(), 15

DSDPSetMaxIts(), 14

DSDPSetMonitor(), 16

DSDPSetPenaltyParameter(), 15

DSDPSetR0(), 15

DSDPSetRTolerance(), 14

DSDPSetStandardMonitor(), 13, 16

DSDPSetup(), 13

DSDPSetY0(), 15

DSDPSetZBar, 15

DSDPSolve(), 13

DSDPStopReason(), 14

DSDPTerminationReason, 14, 15

DSDPUsePenalty(), 15

error check, 7, 9, 14

feasible, 5

Fixed variables, 13

graph problems, 8, 18

header files, 9

infeasible, 8

iteration, 14, 16

Lovász  $\theta$ , 8

LPCone, 12

LPConeGetSArray(), 12, 13

LPConeGetXArray(), 12, 13

LPConeSetData(), 12

LPConeSetXVec(), 13

LPConeView(), 12

maxcut, 8, 9

monitor, 16

MPI, 17

output, 3

penalty parameter, 1, 5, 8, 15

PETSc, 17

PLAPACK, 16, 17

print, 3, 16

SDPA format, 7, 9, 17

SDPCone, 9

SDPConeComputeS(), 11

SDPConeComputeX(), 11

SDPConeGetXArray(), 11

SDPConeSetBlockSize(), 10

SDPConeSetDenseVecMat(), 10

SDPConeSetSparseVecMat(), 10, 11

SDPConeSetSparsity(), 10

SDPConeSetXArray(), 12

SDPConeView(), 11

SDPConeViewDataMatrix(), 11

sparse data, 10–12

symmetric packed storage format, 4, 10

time, 15

trace, 15