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

February 11, 2005

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, 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:

- a robust algorithm with a convergence proof and polynomially bounded complexity under mild assumptions on the data,
- feasible primal and dual solutions,
- relatively low memory requirements for an interior-point method,
- the ability to exploit sparsity and low rank structure 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 problems on parallel architectures, and
- a 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

1 Conic Programming

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

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

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

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 $\langle C, X \rangle :=$ 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, whereas 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 variables y such that $l \le y \le u$ where $l, u \in \mathbb{R}^m$. Furthermore, it uses an auxiliary variable r that represents the infeasibility of (D) and associates it with a penalty parameter Γ . 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:

$$\begin{array}{llllll}
(PP) & \text{minimize} & \displaystyle\sum_{j=1}^{n_b} \langle C_j, X_j \rangle & + & u^T x^u - l^T x^l & \\
& \text{subject to} & \displaystyle\sum_{j=1}^{n_b} \langle A_{i,j}, X_j \rangle & + & x_i^u - x_i^l & = \; b_i, \quad i = 1, \ldots, m, \\
& & X_j \in K_j, & & x^u, x^l \ge 0, & \\
& & \displaystyle\sum_{j=1}^{n_b} \langle I_j, X_j \rangle & & & \le \; \Gamma,
\end{array}$$

$$\begin{array}{lllll}
(DD) & \text{maximize} & \displaystyle\sum_{i=1}^{m} b_i \; y_i - \Gamma r & & \\
& \text{subject to} & C_j - \displaystyle\sum_{i=1}^{m} A_{i,j} y_i + r I_j & = \; S_j \in K_j, & j = 1, \ldots, n_b, \\
& & l_i \le y_i \le u_i, & & i = 1, \ldots, m, \\
& & r \ge 0, & &
\end{array}$$

where I_j is the identity element of K_j, and $x^l, x^u \in \mathbb{R}^m$ are paired with the lower and upper bounds on y, respectively. The parameters Γ, l, and u are set to penalize infeasiblity in (D) and (P) and force the variables r, x^l, and x^u to be near zero at the solution. By default, the bounds and the penalty parameter Γ are very large. 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 solving (P) and (D). It assumes that both of these programs bounded and feasible. Note that (PP) and (DD) are bounded and feasible, and they can be expressed in that form. For simplicity, the notation will refer to semidefinite programming, but the algorithm can be generalized 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 $X, S \succeq 0$. The symbol $\succ$ ($\succeq$) means the matrix is positive (semi)definite. Following conventional notation, let

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

For feasible (y, S), let $\bar{z} = \langle C, X\rangle$ for some feasible X and consider the dual potential function

$$\psi(y, \bar{z}) = \rho \ln(\bar{z} - b^T y) - \ln\det S. \tag{1}$$

The dual-scaling algorithm reduces this function, whose the first term decreases the duality gap and second term keeps S in the interior of the positive semidefinite matrix cone. The gradient of this potential function is

$$\nabla\psi(y, \bar{z}) = -\frac{\rho}{\bar{z} - b^T y} b + \mathcal{A}(S^{-1}). \tag{2}$$

Beginning with a strictly feasible point (y^k, S^k) and upper bound $\bar{z}^k$, each iteration solves the following problem.

$$\text{Minimize} \quad \nabla\psi^T(y^k, \bar{z}^k)(y - y^k) \qquad \text{subject to} \quad \|(S^k)^{-.5}\mathcal{A}^T(y - y^k)(S^k)^{-.5}\| \le \alpha, \tag{3}$$

where α is a positive constant. Denoting $S = S^k$, the step direction Δy solves the linear system

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

The left-hand side of this linear system is a Gram matrix and is positive definite when $S \succ 0$ and the A_is are linearly independent. In this manuscript, it will sometimes be referred to as M.

Using the ellipsoidal constraint, the minimal solution, y^{k+1}, of (3) is given by

$$y^{k+1} - y^k = \beta^k \Delta y \quad \text{where} \quad \beta^k = \frac{\alpha}{\sqrt{-\nabla\psi^T(y^k, \bar{z}^k)\Delta y}}.$$

Practically, the algorithm then selects a step size $\beta_k \in (0, 1]$ such that $S^{k+1} = S^k + \beta^k \Delta S$ is positive definite and the potential function $\psi(y, \bar{z})$ is reduced.

To find a feasible point X, we solve the least squares problem

$$\text{minimize} \quad \|S^{.5}XS^{.5} - \tfrac{\Delta^k}{\rho} I\| \quad \text{subject to} \quad \mathcal{A}X = b. \tag{5}$$

The answer to (5) is given explicitly by

$$X(\bar{z}^k) = \frac{\bar{z} - b^T y}{\rho} S^{-1}\left(\mathcal{A}^T\Delta y + S\right) S^{-1}. \tag{6}$$

Computing this matrix at each iteration is expensive and unnecessary. The next step direction only requires a new upper bound $\bar{z}^{k+1}$, which can be set to

$$\bar{z}^{k+1} := C \bullet X(\bar{z}^k) = b^T y^k + X(\bar{z}^k) \bullet S^k = b^T y^k + \frac{\bar{z} - b^T y^k}{\rho} \left(\Delta y^T \mathcal{A}(S^k)^{-1} + n \right).$$

if $\mathcal{A}^T \Delta y + S \succeq 0$.

Alternatively, consider $\mu > 0$ and

$$\text{maximize} \quad \phi(y) := b^T y + \mu \ln \det S \quad \text{subject to} \quad \mathcal{A}^T y + S = C. \tag{7}$$

The first-order KKT conditions are $\mathcal{A}X = b$, $\mathcal{A}^T y + S = C$, $\mu S^{-1} = X$, and the corresponding Newton equations are

$$\mathcal{A}(X + \Delta X) = b \qquad \mathcal{A}^T(\Delta y) + \Delta S = 0 \qquad \mu S^{-1} \Delta S S^{-1} + \Delta X = \mu S^{-1} - X. \tag{8}$$

The Schur complement of these equations is (4) if we set $\mu = \frac{\bar{z} - b^T y}{\rho}$, and $X(\bar{z}^k) = \mu S^{-1} - \mu S^{-1} \Delta S S^{-1}$ satisfies the constraints $\mathcal{A}X(S, \mu) = b$.

Given a feasible starting point and appropriate choices for step-length and μ, convergence results in [6] show that either the new point (y, S) or the new 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. When $\rho > n$, the infimum of the potential function occurs at an optimal solution. A more thorough explanation of the the dual-scaling algorithm and its convergence properties, can be found in [6].

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   P Objective       D Objective     DInfeas     Mu     StepLength   Pnrm
-----------------------------------------------------------------------------
0      1.00000000e+02   -1.13743137e+05   3.8e+02   1.1e+05  0.00   0.00   0.00
1      1.36503342e+06   -6.65779055e+04   2.2e+02   1.1e+04  1.00   0.33   4.06
2      1.36631922e+05   -6.21604409e+03   1.9e+01   4.5e+02  1.00   1.00   7.85
3      5.45799174e+03   -3.18292092e+03   9.1e+00   7.5e+01  1.00   1.00  17.63
4      1.02930559e+03   -5.39166166e+02   5.3e-01   2.7e+01  1.00   1.00   7.58
5      4.30074471e+02   -3.02460061e+01   0.0e+00   5.6e+00  1.00   1.00  11.36
...
18     8.99999824e+00    8.99999617e+00   0.0e+00   1.7e-08  1.00   1.00   7.03
19     8.99999668e+00    8.99999629e+00   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 current iteration number,
`P Objective`	the current objective value in (PP),
`D Objective`	the current objective value in (DD),
`DInfeas`	the variable r in (DD) that corresponds to the infeasibility of y and S in (D).

`Mu`	the current barrier parameter $\frac{\bar{z}-b^T y}{\rho}$. This parameter decreases to zero as the points get closer to the solution,
`StepLength`	the multiple of the step-directions in (PP) and (DD),
`Pnrm`	the proximity to a point on the central path: $\sqrt{-\nabla\psi^T(y^k,\bar{z}^k)\Delta y}$.

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 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.

4.1 Semidefinite Cones

If the *jth* 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 $A_{1,j},\ldots,A_{m,j}$ 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 $\mathbf{dvec}(\cdot):\mathbb{R}^{n\times n}\to\mathbb{R}^{n(n+1)/2}$, 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 $\mathbf{dmat}(\cdot):\mathbb{R}^{\mathbf{n(n+1)/2}}\to\mathbb{R}^{\mathbf{n\times n}}$, which converts the vector into a square symmetric matrix. Using these operations,

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

and

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

For example, the problem:

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

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 x 1  double ]`, `X{1}'=[ 1.0 1.0 1.0 ]`, and `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 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 on 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.

4.2 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 the 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 for 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.

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 may *significantly* affect the performance of the solver. Options that affect the formulation of the problem are:

`r0`	is the multiple of the identity matrix added to the initial variable S in (DD). Specifically, $S^0 = C - \sum A_i y_i^0 + r^0 * I$. If $r0 < 0$, a dynamic selection will be used that selects a very large number (1e10). IMPORTANT: To improve convergence, use 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 $i = 1, \ldots, m$. The convergence of this solver assumes the solution set of these variables is bounded. These bounds do not have to be tight. The default value is 1e6, but smaller bounds may improve performance.

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 1000].
`steptol`	tolerance for stopping because of small steps [default 1e-2].
`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`	an upper bound for the objective value in (D). The solver stops when it finds a feasible point of (D) with an objective greater than this value. (Helpful in branch-and-bound algorithms.) [default 1e+30].

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

`print`	= k to display output in each k iteration, else = 0 [default 10]. (See Section: 3 to interpret the output).
`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 recognized in `OPTIONS` structure are:

`rho`	to set the potential parameter ρ in the function (1) 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 nonnegative). [default 0].
`mu0`	initial barrier parameter. $\mu = (\bar{z} - b^T y^k)/(\rho)$. [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);
```

asks 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.

4.4 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 (P).
`dobj`	objective value of (D).
`stopcode`	equals 0 if solver converged within 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 matrix added to $C - \mathcal{A}^T(y)$ in the final solution to make S positive definite.
`mu`	the final barrier parameter ($\mu = (\bar{z} - b^T y^k)/(\rho)$)
`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 norm from the targeted located on the central path.
`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[7].

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 specify the initial value of r in (DD).
`boundy <1e6>`	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.
`-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).
`-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 number produce more output.
`-dlogsummary <1>`	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 Lovász Θ 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` 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 `DSDP` 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 `DSDP` solver. The subroutines 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, 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.

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 subroutine is the number of variables in the problem. The second argument should be the address of a `DSDP` variable. This subroutine 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 subroutine

`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 subroutine `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 subroutines will be described in the next sections.

7.2 Semidefinite Cone

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

`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 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 blocks into the same conic structure.

All subroutines 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 subroutine `SDPConeSetBlockSize(SDPCone sdpcone, int blockj, int n)` can be used to specify the dimension of each block and the subroutine `SDPConeSetSparsity( SDPCone sdpcone, int blockj, int nnzmat)` can be used to specify the number of nonzero matrices $\mathtt{A}_{\mathtt{i,j}}$ 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 is up to the user, and the performance of the problem depends upon this choice of data structures. In each of these subroutines, 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 objective matrices in (P) 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. DSDP supports the **symmetric packed** storage format. In symmetric packaged storage format, the elements of a matrix with n rows and columns are ordered as follows:

$$[\ a_{1,1} \quad a_{2,1} \quad a_{2,2} \quad a_{3,1} \quad a_{3,2} \quad a_{3,3}, \quad \ldots, \quad a_{n,n}\ ]. \tag{9}$$

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 its 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 using the subroutine

```
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 $1, \ldots, m$ correspond to matrices $A_{1,j}, \ldots, A_{m,j}$ whereas variable 0 corresponds to C_j. The fourth argument is the dimension (number of rows or columns) of the matrix, n. The fifth argument is the array, and sixth argument is the length of the array. 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 subroutine

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

In this subroutine, the first four arguments are the same as in the subroutine 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 (9) are included in the array of doubles. For example, the matrix

$$A_{i,j} = \begin{bmatrix} 3 & 2 & 0 \\ 2 & 0 & 6 \\ 0 & 6 & 0 \end{bmatrix} \tag{10}$$

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 (10) can be set in the block j and variable i of the semidefinite cone using one of the routines

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

where

$$\begin{array}{ll} \texttt{ind1} = [\ 0 \quad 1 \quad 4\ ] & \texttt{val1} = [\ 3 \quad 2 \quad 6\ ] \\ \texttt{ind2} = [\ 1 \quad 2 \quad 5\ ] & \texttt{val2} = [\ 3 \quad 2 \quad 6\ ] \\ \texttt{ind3} = [\ 7 \quad 3 \quad 5 \quad 4\ ] & \texttt{val3} = [\ 6 \quad 3 \quad 0 \quad 2\ ]. \end{array}$$

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 $\texttt{A}_{\texttt{i,j}}$ and $\texttt{C}_{\texttt{j}}$ 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 subroutine

```
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 subroutine `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 using the command

```
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,

```
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 Xj are stored internally in `SDPCone` object. The subroutine `SDPConeMatrixView(SDPCone sdpcone, int blockj);` can be used to print this matrix to standard output.

The dimension of each block can be found using the routine

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

where the second arguments is the block number and the third arguments are the address of a variable.

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

```
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, the fourth argument is an array containing the variables y. The second argument is the multiple of the matrix 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 whose length is given in the eighth 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 subroutine

```
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 `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.

`DSDP` also supports the **symmetric 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

$$[\ a_{1,1} \quad 0 \quad \ldots \quad 0 \quad a_{2,1} \quad a_{2,2} \quad 0 \quad \ldots \quad 0 \qquad \ldots \qquad a_{n,1} \quad \ldots, \quad a_{n,n}\ ]. \tag{11}$$

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 its more convenient for some applications. The routines

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

set and get the storage format for a block. The second argument specifies the block number and the third argument should be `'P'` for packed storage format and `'U'` for full storage format. The default value is `'P'`. These storage formats correspond 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 $n \times (n+1)/2$. Using 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 (10) can be set in the block j and variable i of the semidefinite cone using one of the routines

```
SDPConeSetSparseVecMat(sdpcone,j,i,3,0,ind1,val1,3);
```

where

$$\texttt{ind1} = \begin{bmatrix} 0 & 3 & 7 \end{bmatrix} \quad \texttt{val1} = \begin{bmatrix} 3 & 2 & 6 \end{bmatrix} .$$

7.3 LP Cone

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

```
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 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` solver, 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 use this `LPCone` variable in the first argument.

A list of n linear inequalities are 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 is represented in sparse row format.)

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

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

In this case, the integer array `nnzin` has length $m+2$, begins with 0, and $\texttt{nnzin}[\texttt{i}+1] - \texttt{nnzin}[\texttt{i}]$ 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, the following problems in the form of (D):

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

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

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

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

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

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

This input is also sparse column input, but the first column corr 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

```
LPConeView(LPCone lpcone);
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 purposes, 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 using the subroutines

```
LPConeGetXArray(LPCone lpcone,double *xout[], int *n);
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 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 solution x of (P). This array should be passed to the cone using the following subroutine

`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.

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 using the subroutine

`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, 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 using the subroutine `DSDPFixVariable(DSDP dsdp, int vari, double val)`.

7.4 Applying the Solver

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

`DSDPSetup(DSDP dsdp);`

This subroutine identifies 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` solver. On very large problems, insufficient memory on the computer may be encountered in this subroutine, so the error code should be checked.

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. If lower and upper bounds on these variables can be determined beforehand, they may be set using the subroutine

`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. These bounds should not be tight. Bounds of negative and positive `1.0e6` can significantly improve the robustness of the solver while usually not affecting the solution. Even if the solution set for y is bounded and bounds have already been incorporated into the model, the developers suggest setting bounds using this routine. If one of the variables nearly equals the bound at the solution, the solver will return a termination code saying (D) is unbounded.

The subroutine

`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 `info=DSDPLogInfoAllow(int,0);` will print even more information if the first argument is positive.

The subroutine

`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 using different initial points.

The subroutine

`DSDPComputeX(DSDP dsdp)`

can be called after `DSDPSolve()` to compute the variables X in (P). These computations are not perfomed within the solver because these variables are not needed to compute the step direction. Infeasibility in either (P) or (D) can be determined using the command

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

This command sets the second argument to an enumerated type. There are four types for `DSDPSolutionType`. The type `DSDP_UNBOUNDED` means that (D) is unbounded and (P) is infeasible. The type `DSDP_INFEASIBLE` means that (D) is infeasible and (P) is unbounded. The type `DSDP_PDFEASIBLE` means that both (D) and (P) are feasible and their objective values are bounded, and the type `DSDP_PDUNKNOWN` means `DSDP` was unable to determine boundedness or feasibility in solutions. The latter type applies when the initial point for (DD) was infeasible.

Each solver created should be destroyed with the command

`DSDPDestroy(DSDP dsdp);`

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

7.5 Convergence Criteria

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

`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

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

specifies the maximum number of iterations. The subroutine `DSDPSetRTolerance(DSDP, double)` specifies how small the constant r representing the infeasibility in (D) must be to be an approximate solution, and the subroutine `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.

7.6 Solutions and Statistics

The objective values in (P) and (D) can be retrieved using the commands

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

```
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

```
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 (D) and (D) satisfy the convergence criteria.
DSDP_MAX_IT	The solver applied the maximum number of iterations without finding solution.
DSDP_INFEASIBLE_START	The initial point in (DD) 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.
DSDP_NUMERICAL_ERROR	Numerical issues prevented further progress.

The subroutines

```
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 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

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

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

`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

`DSDPEventLogSummary(void)`

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

7.7 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 `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

`DSDPSetR0(DSDP dsdp, double r0)`

will set the initial value of r in (DD). 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 default value is usually very large ($1e10$), but smaller values can *significantly* improve performance. The subroutine `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).

The subroutine `DSDPSetPotentialParameter(DSDP dsdp, double rho);` sets the potential parameter ρ. This parameter be 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 using `DSDPSetPenaltyParameter(DSDP dsdp, double M)`, where M is the large positive penalty parameter. This parameter must exceed the trace of the solution X in order to return a feasible solution from an infeasible starting point. The subroutine `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 `M`. The subroutine `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` reuses the 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 or larger than the number of variables y. The subroutine

`DSDPReuseMatrix(DSDP dsdp, int reuse);`

can set a maximum on the number of times the Schur complement matrix is reused. Applications that use a low number of iterations (60 or fewer) should consider setting this number to 0. The default value is 4.

7.8 Iteration Monitor

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, 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 using the commands in the section 7.6. To set this subroutine, use the command

```
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 cam also be used to print the time, duality gap, 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.

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[9]. 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 readsdpa.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 `dsdpplapack.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/readsdpa.c`. Scalability of medium and large-scale problems has been achieved on up to 64 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.

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[6].

1999 DSDP version 2 increased functionality to address semidefinite cones with rank-one constraint matrices and LP constraints [5]. It was used specifically for combinatorial problems such as graph coloring, stable sets[2], 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 [7]. DSDP 3 also featured the initial release of PDSDP[1], 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[4]. 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 [3].

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 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 Johan Lofberg for suggesting many improvements to the software and Hans Mittelmann[8] 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 subprogram 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] S. J. Benson. Parallel computing on semidefinite programs. Technical Report ANL/MCS-P939-0302, Mathematics and Computer Science Division, Argonne National Laboratory, March 2003.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] DIMACS. The Seventh DIMACS Implementation Challenge: 1999-2000. `http://dimacs.rutgers.edu/Challenges/Seventh/`, 1999.

[8] Hans D. Mittelmann. SDPLIB benchmarks. `ftp://plato.asu.edu/pub/sdplib.txt`, 2003.

[9] 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 Copyright

Index