sdpsol

A Parser/Solver for Semidefinite Programming and Determinant Maximization Problems with Matrix Structure

User's Guide

Version Beta   May 31, 1996

Shao-Po Wu and Stephen Boyd

The program sdpsol parses semidefinite programming (SDP) and determinant maximization (MAXDET) problems expressed in the sdpsol language, solves them using interior-point algorithms, and reports the results in a convenient form. This user's guide gives a brief introduction to the problems sdpsol solves, the language parsed by sdpsol, and the usage of sdpsol.

1 Introduction

The problems

sdpsol solves the MAXDET optimization problem

$$\begin{array}{ll} \text{minimize} & c^T x + \log\det G(x)^{-1} \\ \text{subject to} & G(x) > 0, \quad F(x) > 0 \\ & Ax = b \end{array} \tag{1}$$

where the optimization variable is the vector $x \in \mathbf{R}^m$. The functions $G : \mathbf{R}^m \to \mathbf{R}^{l \times l}$ and $F : \mathbf{R}^m \to \mathbf{R}^{n \times n}$ are affine:

$$\begin{aligned} G(x) &= G_0 + x_1 G_1 + \cdots + x_m G_m, \\ F(x) &= F_0 + x_1 F_1 + \cdots + x_m F_m, \end{aligned}$$

where $G_i = G_i^T$ and $F_i = F_i^T$ for $i = 0, \ldots, m$. Here $A \in \mathbf{R}^{p \times m}$, $b \in \mathbf{R}^p$; we take $p = 0$ if there are no equality constraints. The inequality signs in (1) denote matrix inequalities, *i.e.*, $G(x)$ and $F(x)$ are positive definite. Since these matrix inequalities depend affinely on x, we call them *linear matrix inequalities* (LMIs). The MAXDET problem reduces to a *semidefinite programming* (SDP) problem when the log determinant term is absent (*i.e.*, $l = 0$)

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & F(x) > 0 \\ & Ax = b. \end{array} \tag{2}$$

The MAXDET problem (including its special case, SDP) is a convex optimization problem, *i.e.*, the objective function is convex and the constraint set is convex. In fact, LMI constraints can represent many common convex constraints, including linear inequalities, convex quadratic inequalities, matrix norm and eigenvalue constraints. Conversely, many common convex optimization problems can be expressed as MAXDET or SDP problems. See [BEFB94], [VB96] and [VBW96] for many examples that arise in control, statistics, computational geometry, information and communication theory, as well as additional references.

What `sdpsol` does

In many MAXDET or SDP problems, the variable x has matrix structure, which makes it tedious in practice to put the problem into form (1) or (2). The purpose of `sdpsol` is to automate this task by allowing the user to specify (and solve) a MAXDET or SDP problem in a format close to its natural mathematical description, in which variables that are matrices are expressed as matrices instead of some vectorized form. The following example will illustrate the idea.

We consider the matrix completion problem with partially-specified inverse [VBW96, §2.4]. We are given a positive definite diagonal matrix $D \in \mathbf{R}^{n\times n}$ and a matrix $C = C^T \in \mathbf{R}^{n\times n}$. The problem is to 'complete' D to a positive definite matrix whose inverse matches C in the off-diagonal entries. Thus we seek a matrix $\Delta = \Delta^T$ such that

$$D + \Delta > 0, \quad \Delta_{ii} = 0,\ i = 1, \ldots, n, \quad (D+\Delta)^{-1}_{ij} = C_{ij} \text{ for } i \neq j.$$

This problem can be solved by solving the following MAXDET problem:

$$\begin{array}{ll} \text{minimize} & \mathbf{Tr}\, C(D+\Delta) + \log\det(D+\Delta)^{-1} \\ \text{subject to} & D + \Delta > 0 \\ & \Delta_{ii} = 0, \quad i = 1, \ldots, n, \end{array}$$

in which the symmetric matrix $\Delta \in \mathbf{R}^{n\times n}$ is the optimization variable. We can express this problem in the form (1) by finding a basis $P_1, \ldots, P_m$ for symmetric $n \times n$ matrices ($m = n(n+1)/2$), then expressing the optimization variable Δ as $\Delta = \sum_{i=1}^m x_i P_i$, etc. Clearly this is straightforward but inconvenient.

However, the problem (or rather, an instance of the problem) can be specified conveniently in the `sdpsol` language as follows:

sdpsol Source 1 positive definite matrix completion

```
D=diag([2.34, 0.57, 1.13, 3.95]);
C=[ 3.46, -1.01,  0.38, -1.47;
   -1.01,  4.44, -1.48,  2.88;
    0.38, -1.48,  2.43,  0.94;
   -1.47,  2.88,  0.94,  3.97];
variable Delta(4,4) symmetric;  % declare 4x4 symmetric variable Delta

% equality constraints
diag(Delta) == zeros(4,1);
% LMI constraint
D+Delta > 0;

% specify the objective
minimize obj_value = Tr(C*(D+Delta))-logdet(D+Delta);
```

This problem specification reveals some important features of `sdpsol`. It is possible to construct matrices from constants, form matrix expressions using operators and functions, make assignments, declare optimization variables, specify LMI and equality constraints, and define an objective.

When `sdpsol` processes this problem specification, it produces the following output:

Results 1 `sdpsol` output of the matrix completion example

```
sdpsol version beta, Thu May 16 23:30:53 1996

*** Problem: mat_c
10 variables
4 linear equality constraints
LMI size: 4-by-4
1 diagonal block
MAXDET problem.

*** Algorithm parameters:
ABSTOL  = 7.99e-08
RELTOL  = 7.99e-08
BIGM    = 7990
NU      = 10
GAMMA   = 100
MAXITER = 100

*** Optimization result
OPTIMAL after 16 iterations,
sdpsol stopped because RELATIVE TOLERANCE was reached.

*** Objective value
obj_value = 11.63

*** Variable
Delta =
[    0.0000,   -0.8277,   -1.2558,    2.6579;
    -0.8277,    0.0000,    0.6525,   -1.3174;
    -1.2558,    0.6525,    0.0000,   -1.8167;
     2.6579,   -1.3174,   -1.8167,    0.0000 ]
```

The output reports some basic information about the problem, the optimization algorithm parameters used, the status of the optimization phase, the optimal values of the variables, and the optimal objective value.

2 The sdpsol language

Comments

`sdpsol` supports two styles of comments: C and Matlab. The former starts with `/*` and ends with `*/`; the latter begins with `%` and includes any following text on the same line. Comments are stripped and ignored by `sdpsol`.

Forming vectors and matrices

The only data type used by `sdpsol` is matrix. In other words, every expression is a matrix with fixed dimension, *i.e.*, number of rows and columns. Matrices with one row and column are considered scalars; matrices with only one column are considered vectors; and matrices with only one row are considered (row) vectors. A matrix is formed from numbers using the symbols `[` and `]` to enclose the matrix, semicolons to delimit rows, and commas to delimit entries within a row. For example, the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

can be expressed as `[1,2,3; 4,5,6; 7,8,9]`. Row and column vectors (which are just matrices) are expressed the same way. For example, `[1, 2, 3]` denotes the row vector

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix},$$

and `[1; 2; 3]` denotes the vector

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

Since `sdpsol` considers 1×1 matrices to be scalars, `[3]` is the same as the scalar 3.

Newlines are completely equivalent to spaces; in particular newlines do *not* delimit the rows of a matrix, as they do in Matlab. Thus `[1 2; 3 4]` is *not* a valid `sdpsol` expression (although it is legal in Matlab), and

```
[1, 2,
 3, 4]
```

represents the row vector $[1, 2, 3, 4]$, and not the 2×2 matrix it represents in Matlab.

So far we have formed matrices (and vectors) from scalar constants. In fact the same constructions can be used to form matrices from other matrices, provided the sizes make sense: the matrices in any given row (delimited by commas) must have the same number of rows, and each row must have the same total number of columns. As a complicated example, `[[1,2,3], 4; [5;9], [6;10], [7,8; [11,12]]]` denotes the matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}.$$

Several special matrices are provided:

- `zeros(i,j)` denotes an $i \times j$ zero matrix.
- `ones(i,j)` denotes an $i \times j$ matrix with all entries one.
- `eye(i,j)` denotes an $i \times j$ matrix with main diagonal one; `eye(i)` denotes an $i \times i$ identity matrix.
- `[a:b]` generates a *row* vector of consecutive integers. If $a \le b$ it generates the (ascending integer) vector $[i, i+1, \dots, j]$, where i is a rounded toward zero and j is b rounded toward zero. If $a > b$ it generates the (descending integer) row vector $[i, i-1, \dots, j]$.

Variables

The matrices in the preceding section are **constants**, since they have specific numerical values. `sdpsol` also supports **variables**, which are the optimization variables in the problem. Variables are explicitly declared with declaration statements. The syntax of variable declaration is

`variable` *variable_name*[(*dimension*)] [,*variable_name*[(*dimension*)] ...][*attribute*];

Arguments enclosed by brackets are optional. *attribute* gives the structure of the variable(s) declared; it can be either **symmetric** or **diagonal**. Variable declaration without attribute indicates that the variables have no structure. *variable_name* serves as the name of the variable declared. A valid name starts with a letter and is followed by letters, digits or underscores. As an example,

```
variable A(5,5) symmetric;
variable b(k,1);
variable c(1,1), d;
```

declares a 5×5 symmetric matrix variable A, a $k \times 1$ vector variable b (k is an internal variable, see page 9), and two scalar variables c and d.

A variable name cannot be the name of any previously declared variable, or a reserved key word:

`include`	`variable`	`constraint`	`initialize`	`minimize`
`maximize`	`symmetric`	`diagonal`	`ones`	`zeros`
`eye`	`toeplitz`	`diag`	`ip`	`Tr`
`reshape`	`sum`	`rows`	`cols`	`what`
`disp`	`logdet`	`sumlog`	`for`	`end`

If a variable name has been used as an internal variable, `sdpsol` simply overwrites its previous assignment by the variable declaration. In this version of `sdpsol`, all variable declarations must precede all constraint or objective statements (see page 9).

It is possible to initialize a variable for the optimization phase, with a statement of the form

`initialize` *variable_name* = *constant_expression* ;

Variables must be declared before they can be initialized. It's important to understand that initializing a variable does not assign it a value; it simply gives the variable a starting value to use in the optimization phase. The only purpose of initialization is to speed up the optimization phase.

Expressions

Constants and variables can be combined using various **operators** (such as addition, multiplication, and transpose) and **functions** (such as trace and inner product) to form **expressions**. For example (and assuming that P has already been declared as a 3×3 matrix variable),

```
[1,2,3; 4,5,6; 7,8,9]' * P + P * [1,2,3; 4,5,6; 7,8,9] + [1;2;3]*[1;2;3]'
```

represents

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T P + P \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}^T .$$

Note that this expression is a 3×3 matrix that is an **affine** function of the variable P. We say this expression is an **affine expression**. If an expression does not depend on any variable, *i.e.*, has a specific numerical value, we say it is a **constant expression**. In the current version of `sdpsol`, except special expressions that are used to form the log determinant objective (see page 8), all expressions are affine (or constant) functions of the variables. This limits the ways you can combine expressions to form others. For example, you cannot multiply two affine expressions, since this would result in an expression that depends quadratically on the variables.

Operators

`sdpsol` provides the following matrix operations, from higher precedence to lower precedence: transpose, unary plus/minus, multiplication (including scalar multiplication/division and componentwise multiplication/division), and addition/subtraction.

- `A'` is the transpose of A.
- `-A` denotes the negative of A; `+A` is just A.
- `A*B`, A multiplied by B, can mean two things:

- *Matrix multiplication.* The number of columns of A equals the number of rows of B. In this case `A*B` means the usual matrix multiplication AB.
- *Scalar/matrix multiplication.* A or B is a scalar, *i.e.*, a 1×1 matrix. In this case `A*B` means scalar multiplication of the matrix by the scalar.

Either A or B must be a constant expression.

- `A/a` denotes scalar multiplication of the matrix A by $1/a$, where a is a non-zero scalar constant expression.
- `A.*B` denotes componentwise multiplication of A and B, *i.e.*, $(\texttt{A.*B})_{ij} = A_{ij}B_{ij}$. A and B must have the same dimension and at least one of them is a constant expression.
- `A./B` denotes componentwise division of A by B, *i.e.*, $(\texttt{A./B})_{ij} = A_{ij}/B_{ij}$. A and B must have the same dimension and B must be a constant expression, with all entries non-zero.
- `A+B` means one of the following (similar rules apply to `A-B`):
  - `A+B` denotes $A + B$, if A and B are matrices of the same dimension.
  - `a+B` denotes $aI + B$, if a is *scalar* and B is *square*. Similarly, `A+b` denotes $A + bI$.
  - `a+B` means $(\texttt{a+B})_i = a + B_i$, if a is *scalar* and B is a (column or row) *vector*. Similarly, `A+b` means $(\texttt{A+b})_i = A_i + b$.

  If one of the operands is scalar and the other is not a vector or a square matrix (including scalar), the operation is invalid. Note that this nonstandard extension of addition/subtraction, in which a scalar can be added or subtracted from vectors and square matrices, is consistent with common mathematical notation and greatly simplifies the description of some SDPs.

Operations are carried out in the order of precedence. For example in the expression `A*B+C'`, `C'` is formed first, then the product `A*B`, and finally the sum. Parentheses, *i.e.*, `(` and `)`, can always be used to force groupings. For example in `A*(B+C)'`, `B+C` is formed, then its transpose, and then the multiplication.

Functions

`sdpsol` provides the following functions:

- `cols(A)` returns the number of columns of A, a positive, integer scalar.
- `rows(A)` returns the number of rows of A, a positive, integer scalar.
- `reshape(A,r,c)` denotes the $r \times c$ matrix expression with its elements taken column-wise from A. The product of the number of rows and columns of A must be rc; `sdpsol` gives an error if A does *not* have rc elements.

- `diag(A)` is used in two ways:
  - *Forming diagonal matrices.* If `A` is a column or row vector, `diag(A)` is a diagonal matrix with entries of A along its diagonal.
  - *Extracting the diagonal of a matrix.* If `A` is a matrix but not a row or column vector, `diag(A)` is a *column* vector made of the main diagonal entries of A.
- `sum(A)` denotes the sum of the elements of a *vector* argument A. If A is a matrix, `sum(A)` is a *row vector* containing the sum over each column of A, that is, `ones(1,rows(A))*A`.
- `Tr(A)` denotes the trace of A, *i.e.*, the sum of the diagonal entries of A. A has to be *square*.
- `ip(A,B)` denotes the inner-product of A and B, *i.e.*, $\mathbf{Tr}\, A^T B$. A and B must have the same dimension, and one must be a constant expression.

`sdpsol` provides the following functions used to specify log determinant objectives:

- `logdet(A)` denotes the log determinant of (the symmetric part of) the matrix A, *i.e.*, $\log\det(A + A^T)/2$. Using this expression automatically adds the constraint $(A + A^T)/2 > 0$ to the problem specification.
- `sumlog(a)` denotes $\sum_i \log(a_i)$, where a is a row or column vector. `sdpsol` automatically adds the constraint $a > 0$.

When the arguments to these two functions are constant expressions, the functions return constant scalar expressions, which can then be used anywhere constant scalar expressions can be used. When the arguments to these two functions are affine expressions, these functions return expressions that (obviously) do not depend affinely on the variables. There are therefore several limitations on how and where they can be used and combined:

- They cannot be used in any constraint (since they are not affine).
- They cannot be multiplied with other expressions, even scalar constant expressions.
- They cannot be used to form matrices.
- They can be added and subtracted with each other and affine expressions. (But the final expression used in the objective must be convex if it is a minimization problem, or concave if it is a maximization problem; see page 10).

Let us consider some examples. Assume X is a square matrix variable and v is a vector variable, and B and C are constant matrices. The following expression is valid:

```
3-sumlog(v)-logdet(B*X+X'*B')+2*Tr(X)/logdet(C)
```

whereas the following expressions are *not* valid:

```
-3*sumlog(v)
[1, 0; 0, sumlog(v)]
Tr(X)/logdet(X)
```

Internal variables and assignments

`sdpsol` allows the user to create **internal variables**, *i.e.*, give names to expressions using **assignments**. The syntax of assignment is

variable_name = *expression* ;

The name must be a valid variable name (see page 5). Consider the example

```
Aplant = 4*[1, 2, 3; 0, 0, 0; 4, 1, 1] ;
Lyap = Aplant'*P + P*Aplant ;
```

In the first assignment, *Aplant* is given the constant value

$$\begin{bmatrix} 4 & 8 & 12 \\ 0 & 0 & 0 \\ 16 & 4 & 4 \end{bmatrix}$$

(overwriting any previous definition of *Aplant* as an internal variable). Once *Aplant* has been assigned it can itself be used in expressions, as in the second assignment, in which *Lyap* becomes an expression (that could be used in subsequent expressions, etc.).

It is possible to refer to individual elements (submatrices) of an internal variable via subscripting. A subscript can be a scalar, a vector or a colon (`:`); colon denotes all of the corresponding row or column. For example, `Aplant(1,2)` refers to the $(1,2)$ entry of *Aplant*, which is the constant scalar 8; `Aplant(3,:)` is the row vector

$$\begin{bmatrix} 12 & 0 & 4 \end{bmatrix}$$

and `Aplant([1,3],[3,3,1])` denotes the matrix

$$\begin{bmatrix} 12 & 12 & 4 \\ 4 & 4 & 16 \end{bmatrix}.$$

No element in a subscript can exceed the dimension of the given internal variable, *e.g.*, `Aplant(1,4)` is *invalid*.

In this version of `sdpsol`, expressions *cannot* be assigned to an internal variable with subscripts. An assignment such as

```
Aplant(2,2) = 3;
```

which is valid in Matlab, is *invalid* in `sdpsol`.

Constraints

Expressions can be combined with **relation operators** to form **constraints**, as in

```
diag(Delta) == zeros(4,1);
D+Delta > 0;
```

which are taken from our example. The syntax of constraint specification is

expression rel_op expression ;

The relation operator *rel_op* can be equality (`==`), matrix inequality (`>`, `<`) or componentwise inequality (`.>`, `.<`). At least one of the two expressions has to be affine; otherwise an error message results. No expression with special functions, such as `logdet()` and `sumlog()` (see page 8), can be used in constraints.

As with operators, we interpret constraints in a convenient way when one of the expressions is a scalar.

- `A == B` means $A = B$, if A and B are matrices of the same dimension. This is the most common usage, but for consistency `sdpsol` extends equality to include vector-scalar and matrix-scalar equality. If a is scalar and B is square, `a == B` (or `B == a`) means $aI = B$. If a is scalar and B is either a column or row vector, `a == B` (or `B == a`) means $a = B_i$. If B is a column vector, for example, `a == B` is the same as `a*ones(1,rows(B)) == B`.

- `A > 0` means $(A+A^T)/2 > 0$ ($(A+A^T)/2$ is positive definite) if A is *square*. Similarly, `A < 0` means $(A+A^T)/2 < 0$. Note that `sdpsol` automatically symmetrizes positive (or negative) definite constraints.

- `A > k` means $(A+A^T)/2 - kI > 0$, if k is a *scalar* and A is *square*. A similar rule applies to `A < k`.

- `A > B` means $A - B > 0$, *i.e.*, $((A-B)+(A-B)^T)/2 > 0$, if `A` and `B` are *square* matrices of the same dimension.

- `A .> k` means the componentwise inequality $A_{ij} > k$, where k is a scalar. Similarly, `A .< k` means $A_{ij} < k$.

- If `A` and `B` are matrices (or vectors) of the same size, `A .> B` (and `B .< A`) means $A_{ij} > B_{ij}$.

Objective

The objective or cost function is given by an objective statement, *e.g.*,

```
minimize obj_value = Tr(C*(D+Delta))-logdet(D+Delta);
```

which is taken from our example. This statement assigns $\mathbf{Tr}\, C(D+\Delta) - \log\det(D+\Delta)$ to the internal variable *obj_value* and tells `sdpsol` to minimize it. The syntax of objective statement is

`minimize` *variable_name* = *scalar_expression* ;
`maximize` *variable_name* = *scalar_expression* ;

where *scalar_expression* is composed of scalar affine expressions and/or log determinant expressions. The objective expression must be convex in the optimization variables to be minimized (or concave in the variables to be maximized).

For example, assume X is a square matrix variable, v is a vector variable, and B is a constant matrix. Then the following objective statements are valid:

```
minimize obj_value = 3+2*Tr(X);
maximize obj_value = 3+2*Tr(X);
minimize obj_value = 3-sumlog(v)-logdet(B*X+X'*B')+2*Tr(X);
```

whereas the following objective statements are *not* valid:

```
maximize obj_value = 3-sumlog(v)-logdet(B*X+X'*B')+2*Tr(X);
minimize obj_value = 3-sumlog(v)+logdet(B*X+X'*B')+2*Tr(X);
```

because the former is convex and the latter is neither convex nor concave in X and v.

The objective statement is optional. If no objective is given (or the objective given is constant), `sdpsol` forms and solves the **feasibility problem** only, *i.e.*, it either finds a solution which satisfies all the constraints or proves that the constraints are infeasible. If there is more than one objective statement, a warning message is issued, and only the last one is used.

Commands

The command

    include("*filename*");

includes two types of files into the workspace of `sdpsol`: Matlab binary data files (.mat files) and `sdpsol` source files. If *filename* ends in `.mat`, `sdpsol` treats the file as a Matlab binary file and loads all real variables in the file as internal variables. In this version of `sdpsol`, all complex and string variables in the .mat file are (for now, silently) ignored. If *filename* does not end with the extension `.mat`, the file is included as `sdpsol` source, *i.e.*, as part of the problem specification. Includes can nest up to a maximum of 10 levels.

`what(`*expression*`);` prints to the standard output the dimension, attribute, and type of the given expression, along with a list of the variables on which the expression depends. If the expression is constant, its value will be printed. Using our example, the command `what(C);` prints

```
4x4  internal variable, constant with value:
[    3.4600,   -1.0100,    0.3800,   -1.4700;
    -1.0100,    4.4400,   -1.4800,    2.8800;
     0.3800,   -1.4800,    2.4300,    0.9400;
    -1.4700,    2.8800,    0.9400,    3.9700 ]
```

and `what(Tr(C*(D+Delta))-logdet(D+Delta));` prints

```
1x1  expression, depends on variable(s):
  Delta
```

Loops

Loops can be used to repeat statements, as in

```
for i=1:K;
  [eye(n),          E*X(:,i)+d;
   (E*X(:,i)+d)',   1           ] > 0;
end;
```

which declares the K LMI constraints

$$\begin{bmatrix} I & Ex_i + d \\ (Ex_i + d)^T & 1 \end{bmatrix} > 0,$$

where $X = [x_1, \ldots, x_K]$ and $i = 1, \ldots, K$.

The syntax of a loop is

> `for` *variable_name* = *scalar_constant_1* [: *scalar_constant_2*] : *scalar_constant_3* ;
> [*statement* ; ... *statement* ;]
> `end` ;

Note that the semicolons used to delimit the `for`-statement and the `end`-statement above *cannot* be omitted. This is different from the syntax of loops in Matlab.

Within a loop, the **loop variable** *variable_name* is assigned an integer value from *scalar_constant_1* with increment *scalar_constant_2* (or decrement if *scalar_constant_2* is negative), one at a time, until it exceeds *scalar_constant_3*. These scalar constants are rounded toward zero to ensure that they have integer values. *scalar_constant_2* is optional and it has a default value 1. For example, `for i=1:10;` loops 10 times with the loop variable $i = 1, 2, \ldots, 10$; `for n=10:-2:1;` loops 5 times with $n = 10, 8, \ldots, 2$.

In this version of `sdpsol`, every loop is executed at least once, even when the loop variable has a starting value exceeds its ending value. As an example,

```
for i=1:0;
  what(i);
end;
```

is executed once (with $i = 1$) and `sdpsol` prints out

```
1x1  internal variable, constant with value:
  1
```

Loops can nest up to a maximum of 10 levels.

Algorithm parameters

Several internal variables are reserved by `sdpsol` to serve as algorithm parameters. See [VB96, VB94, VBW96, WVB96] for the precise meanings of these parameters, including their allowable and recommended values. The value of these reserved internal variables can be reassigned by the user via assignments.

NU — The parameter ν controls the rate of convergence when solving SDP problems (see [VB94]). Default value is 10.

GAMMA — The parameter γ controls the rate of convergence when solving MAXDET problems (see [WVB96]). Default value is 100.

MAXITER — The maximum number of iterations allowed. Default value is 100.

ABSTOL — Absolute tolerance. Default value is $\max\{10^{-8}\kappa, 10^{-8}\}$, where $\kappa \approx \max_{i\in\{1,\ldots,m\}} \|F_i\|$.

RELTOL — Relative tolerance. Default value is $\max\{10^{-8}\kappa, 10^{-8}\}$.

BIGM — The parameter M used in the big-M method (see [VB96, §6.1]). Default value is $10^3\kappa$.

3 Using `sdpsol`

There are two ways to use `sdpsol`: as a stand-alone command under UNIX, or, from within Matlab, via the Matlab script `sdpsol.m`.

Using `sdpsol` from UNIX

Under UNIX, `sdpsol` is invoked as

`your_unix_system% sdpsol` [*options*] *src_filename*

Options are:

–h — Show the usage of `sdpsol`.

–q — Quiet mode. Compilation and run-time log messages are suppressed. However, error messages are still sent to `stderr`. These error messages can be suppressed by invoking `sdpsol` with `stderr` redirected, *e.g.*,

`sdpsol` *src_filename* `>& /dev/null`)

–I *incl_filename* — Include the Matlab binary file *incl_filename* into the workspace of `sdpsol`. This is equivalent to having the command `include("`*incl_filename*`")` at the top of the source file.

–m [*mat_filename*] — Export the results of `sdpsol` to the Matlab binary file *mat_filename*. If *mat_filename* is not given, *src_filename*.mat is used.

–o [*out_filename*] — Redirect `sdpsol`'s (text) output from the standard output (default) to the file *out_filename*. If *out_filename* is not given, *src_filename*.out is used.

For example,

```
your_unix_system% sdpsol -q -m my_result.mat my_source
```

invokes `sdpsol` in quiet mode, solves the problem specified in `my_source`, and exports the results to the Matlab binary file `my_result.mat`.

When the **−m** option is used, an additional Matlab string variable `INFO` is written to the *.mat* file along with the values of all variables and the objective. `INFO` has one of the following values: `'infeasible'`, `'feasible'`, `'optimal'` or `'error'`. Their meanings are summarized in Table 1.

`INFO`	Feasibility Problem	Optimization Problem
`'infeasible'`	problem infeasible	problem infeasible
`'feasible'`	problem feasible	feasible solution found, but MAXITER exceeded during optimization
`'optimal'`	(not applicable)	optimum solution found
`'error'`	feasibility unknown	feasibility unknown

Table 1: Possible values of `INFO` and their meanings.

Using `sdpsol` from within Matlab

We have provided a simple Matlab script `sdpsol.m` that can be used to invoke `sdpsol` from within Matlab. The `sdpsol` source filename is stored in the Matlab string variable `SDPSOL_FILENAME`. For example,

```
>> SDPSOL_FILENAME='my_source';
>> sdpsol
```

invokes `sdpsol` to solve the problem specified in `my_source`.

When `sdpsol` is invoked, all the variables (excluding strings and complex variables) in the Matlab workspace are included by `sdpsol` as internal variables. Therefore, no `include` statement is necessary in the `sdpsol` source. When `sdpsol` processes the problem specification, the results (including `INFO`) are automatically exported back to Matlab's workspace. To give a simple example, consider the following `sdpsol` source `my_source`:

```
% sdpsol source file -- my_source
% WARNING: cannot be run from UNIX, since A and n are not defined.
variable P(n,n) symmetric;
A'*P+P*A < -0.1;
P > 0;
Tr(P) == 1;
```

This `sdpsol` source does *not* work with the stand-alone `sdpsol` invoked from UNIX, because A and n are not defined. However, within Matlab, we might use `sdpsol` in the following way:

```
>> n=2;
>> A=[-1.3628   -0.7566;   -0.7566   -0.5166];
>> SDPSOL_FILENAME='my_source';
>> sdpsol
```

... *some messages from* `sdpsol` *will appear* ...

```
>> P
P =
    0.2744   -0.4034
   -0.4034    0.7256
>> INFO
INFO =
feasible
```

4 Examples

Minimum volume ellipsoid containing given points

Consider the MAXDET problem arises in determining the minimum volume ellipsoid that contains given points $x^1, \ldots, x^K$ in $\mathbf{R}^n$. We describe the ellipsoid as $\mathcal{E} = \{x \mid \|Ax+b\| \le 1\}$, where $A = A^T > 0$, so the volume of $\mathcal{E}$ is proportional to $\det A^{-1}$. Hence the minimum volume ellipsoid that contains the points x^i can be computed by solving the convex problem

$$\begin{array}{ll} \text{minimize} & \log\det A^{-1} \\ \text{subject to} & \|Ax^i + b\| \le 1, \; i = 1, \ldots, K \\ & A = A^T > 0, \end{array} \tag{3}$$

where the variables are $A = A^T \in \mathbf{R}^{n\times n}$ and $b \in \mathbf{R}^n$. The norm constraints $\|Ax^i + b\| \le 1$, which are just convex quadratic inequalities in the variables A and b, can be expressed as LMIs

$$\begin{bmatrix} I & Ax^i + b \\ (Ax^i + b)^T & 1 \end{bmatrix} \ge 0.$$

Thus (3) is a MAXDET problem in the variables A and b, and it can be specified using the `sdpsol` language as shown in Source 2.

Log Chebychev approximation

Suppose we wish to solve $Ax \approx b$ approximately, where $A = [a_1 \ldots a_p]^T \in \mathbf{R}^{p\times k}$ and $b \in \mathbf{R}^p$. In some applications b_i has the dimension of a power or intensity (*e.g.*, filter design), and is typically expressed on a log scale. In such cases we would like to solve a log Chebychev approximation to minimize the $\mathbf{L}_\infty$-norm of the log residual, *i.e.*, we solve

$$\text{minimize} \quad \max_i |\log(a_i^T x) - \log(b_i)| \tag{4}$$

(assuming $b_i > 0$, and interpreting $\log(a_i^T x)$ as $-\infty$ when $a_i^T x \le 0$).

This *log Chebychev approximation* problem can be cast as a semidefinite program. To see this, note that

$$|\log(a_i^T x) - \log(b_i)| = \log\max(a_i^T x/b_i, b_i/a_i^T x)$$

`sdpsol` **Source 2** minimum volume ellipsoid containing given points

```
% sdpsol source minVe_pts -- find minimum volume ellipsoid
%  { x | || A*x+b ||<=1 }
% containing K points in R^n, x_i, i=1,...,K (stored in X, an n-by-K matrix)
%
% maxdet program:
%   minimize    - log det A
%   subject to  || A*x_i+b || <= 1, i=1,...,n
%               A > 0
%
% WARNING: cannot be run from UNIX, since X, n, K are not defined.

variable A(n,n) symmetric;
variable b(n,1);

for i=1:K;
  [eye(n),         A*X(:,i)+b;
   (A*X(:,i)+b)',  1          ] > 0;
end;
A > 0;

minimize obj = -logdet(A);
```

(assuming $a_i^T x > 0$). Problem (4) is therefore equivalent to

$$\begin{array}{ll} \text{minimize} & t \\ \text{subject to} & Ax \leq tb \\ & \begin{bmatrix} a_i^T x & 1 \\ 1 & t/b_i \end{bmatrix} \geq 0, \quad i = 1, \ldots, p \end{array}$$

which is a semidefinite program. This problem can be described using the `sdpsol` language as shown in Source 3.

`sdpsol` **Source 3** log Chebychev approximation

```
% sdpsol source log_chebychev -- find x such that Ax > 0
% and |log(Ax) - log(b)|_\infty is minimized.
%
% WARNING: cannot be run from UNIX, since A, b, p, k are not defined.

variable x(k,1),t;

A*x .< t*b;
for i=1:p;
  [A(i,:)*x, 1;
   1,        t/b(i,1)] > 0;
end;

minimize err_bnd = t;
```

5 Caveats

The most important caveats for this version are:

- it is limited to small problems, and
- it handles only strict LMIs.

We hope that both of these short-comings will be removed in future versions.

The matrix structure of x can be exploited to great benefit in the optimization algorithm; see [VB95, BVG94]. This version of `sdpsol`, however, *does not* exploit such structure, and hence is only appropriate for small and medium-sized problems (say, with total number of optimization variables on the order of a hundred.)

`sdpsol` solves the problem (1) and (2), with strict LMIs. If the problem you want to solve does not have a strictly feasible point, `sdpsol` will, of course, fail to find a strictly feasible point. At the same time it will fail to conclude that the problem is infeasible (`sdpsol` terminates with the message "feasibility cannot be determined" and `INFO` equal to `'error'`). Indeed, this gives a hint that your problem does not have a strictly feasible point.

References

[BEFB94] S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*, volume 15 of *Studies in Applied Mathematics*. SIAM, Philadelphia, PA, June 1994.

[BVG94] S. Boyd, L. Vandenberghe, and M. Grant. Efficient convex optimization for engineering design. In *Proceedings IFAC Symposium on Robust Control Design*, pages 14–23, September 1994.

[VB94] L. Vandenberghe and S. Boyd. SP*: Software for Semidefinite Programming. User's Guide, Beta Version.* K.U. Leuven and Stanford University, October 1994.

[VB95] L. Vandenberghe and S. Boyd. A primal-dual potential reduction method for problems involving matrix inequalities. *Mathematical Programming*, 69(1):205–236, July 1995.

[VB96] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, March 1996.

[VBW96] L. Vandenberghe, S. Boyd, and S.-P. Wu. Determinant maximization with linear matrix inequality constraints. *submitted to SIMAX*, February 1996.

[WVB96] S.-P. Wu, L. Vandenberghe, and S. Boyd. MAXDET*: Software for Determinant Maximization Problems. User's Guide, Alpha Version.* Stanford University, April 1996.