sdpsol

A Parser/Solver for Semidefinite Programs with Matrix Structure

User's Guide

Version Alpha
November 9, 1995

Stephen Boyd and Shaopo Wu

The program `sdpsol` parses semidefinite programming problems expressed in the `sdpsol` language, solves them using a primal-dual interior-point algorithm, 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`. Many example `sdpsol` source files can be found in the directory `examples`. A more detailed reference manual will be available soon.

1 Introduction

A *linear matrix inequality* (LMI) is an inequality of the form

$$F(x) = F_0 + \sum_{i=1}^{m} x_i F_i \geq 0$$

where $F_0, \ldots, F_m$ are given symmetric matrices, and the inequality means that F is positive semidefinite. An LMI is a convex constraint on x. Many common convex constraints, *e.g.*, linear and quadratic inequalities, eigenvalue and matrix norm inequalities, can be expressed as LMIs with appropriate F_i. In *semidefinite programming* (SDP) we minimize a linear function subject to an LMI constraint:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & F(x) \geq 0 \end{array} \tag{1}$$

where $x \in \mathbf{R}^m$ is the optimization variable. SDPs are convex optimization problems; conversely, many convex optimization problems can be expressed as SDPs. The survey paper [VB94a] gives an overview of semidefinite programming and applications.

Note that equality constraints on x can be incorporated into the LMI constraint. For example, we can incorporate $Ax = b$ by adding two more block diagonal LMIs, $\mathbf{diag}(Ax - b) \geq 0$ and $-\mathbf{diag}(Ax - b) \geq 0$, to the original $F(x)$.

`sdpsol` solves a variation on the basic SDP, with the special form

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

(This problem might be more accurately called a positive-definite programm with equality constraints, but we will continue to use the term SDP.) Note that in this form, the equality constraints are explicit; it is not

possible to express equality constraints in the LMI, which is strict. Of course the solution of (2) lies on the boundary of the LMI, *i.e.*, at a point $x_{\rm opt}$ with $F(x_{\rm opt})$ positive semidefinite but not positive definite.

SDPs arise in many applications, *e.g.*, control theory and combinatorial optimization. In many problems the variable x has matrix structure, which makes it tedious in practice to put the problem into our standard SDP form (2). The purpose of `sdpsol` is to automate this task by allowing the user to specify (and solve) an SDP in a format close to the natural mathematical description of the problem.

The following simple example will illustate the idea. Consider the problem

$$\begin{array}{ll} \text{minimize} & \mathbf{Tr}\, P \\ \text{subject to} & P = P^T > 0, \quad D \text{ diagonal}, \quad D > 0, \quad \mathbf{Tr}\, D = 1 \\ & A_1^T P + P A_1 + B D B^T < 0, \\ & A_2^T P + P A_2 + B D B^T < 0 \end{array}$$

in which the matrices $P \in \mathbf{R}^{p\times p}$ and $D \in \mathbf{R}^{q\times q}$ are the variables, and the matrices A_1, A_2, and B are given. (See [BEFB94] for *many* examples similar to this one.) We can express this problem in the form (2) by taking a basis for symmetric $p \times p$ matrices and diagonal $q \times q$ matrices, then expressing the variable matrices P and D as linear functions of $x \in \mathbf{R}^m$, etc. Clearly this is straightforward but inconvenient.

This problem (or really, a specific instance of the problem) can be specified in the `sdpsol` language as follows:

SDP specification 1.1 sdp-example

```
/* give A1, A2, and B */
A1 = [-1, 0, 1; 0, -2, 1; 0, 0, -1];
A2 = A1 + 0.1*A1';
B = [1, 2; 3, 4; 5, 6];
/* variable declarations */
variable P(3,3) symmetric;   /* P is a 3x3 symmetric matrix */
variable D(2,2) diagonal;    /* D is a 2x2 diagonal matrix */
/* constraints */
A1'*P+P*A1 + B*D*B' < 0;
A2'*P+P*A2 + B*D*B' < 0;
P > 0;
D > 0;
Tr(D) == 1;
/* objective */
minimize trace = Tr(P);
```

This problem specification (which can be found in the file `sdp-example` in the `examples` directory) reveals some important features. 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 source file, it produces the following output (in the file `sdp-example.out`):

Results 1.1 sdp-example.out

```
sdpsol version alpha, Mon Mar  6 15:49:34 1995

*** Problem: sdp-example
8 variables
1 linear equality constraint
LMI size: 11-by-11
5 diagonal blocks
OPTIMIZATION problem.

*** SP parameters:
ABSTOL  = 1.59e-06
RELTOL  = 1.59e-06
BIGM    = 159000
NU      = 10
MAXITER = 50

*** Optimization result
OPTIMAL after 13 iterations,
sdpsol stopped because RELATIVE TOLERANCE was reached.

*** Objective value
trace = 25.6966

*** Variables
P =
[    1.0747,    1.2435,    3.5758;
     1.2435,    2.3661,    6.1679;
     3.5758,    6.1679,   22.2558 ]
D =
[    1.0000,    0.0000;
     0.0000,    0.0000 ]
```

The output file reports some basic information about the problem, the optimization parameters used by the primal-dual interior point solver `sp`, the status of the optimization phase, and the optimal values of the variables.

2 The `sdpsol` language

Comments start with `/*` and end with `*/`; nested comments are not allowed. Comments are stripped and ignored by `sdpsol`.

2.1 Forming vectors and matrices

The only data type in `sdpsol` is the (two-dimensional) matrix. In other words, every variable, constant, or expression is a matrix with some 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 (scalar) 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 (column) 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 and vectors are provided for convenience.

- `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 $a \leq b$ it generates the (ascending integer) vector $[i, i+1, \ldots, j]$, where i is a rounded toward zero and j is b rounded toward zero. If $a > b$ it generates the (descending integer) row vector $[i, i-1, \ldots, j]$.

2.2 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*[(*dimensions*)] [,*variable_name*[(*dimensions*)] ...][*attribute*] ;

Arguments enclosed by brackets are, of course, optional. *attribute* gives the structure of the variable(s) declared; it can be **symmetric** or **diagonal**. *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. A variable name cannot be the name of any internal variable (see §2.3), a previously declared variable, or a reserved key word:

```
include    variable    constraint   initialize   minimize
maximize   symmetric   diagonal       ones        zeros
  eye        diag         ip           Tr        reshape
  sum        rows        cols          what
```

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), and two scalar variables `c` and `d`. In this version of `sdpsol`, variable declarations are *not* allowed after the first constraint statement (see §2.5) in the `sdpsol` source file. In other words, all variable declarations must precede all constraint statements.

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. Initialization is only used to speed up the optimization phase.

2.3 Expressions, assignments and internal variables

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]'
```

which 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`, all expressions are affine (or constant) functions of the variables. This limits the ways you can combine expressions to form others. Thus you can multiply two constant matrix expressions (of appropriate dimensions), a constant matrix expression and an affine matrix expression, but *not* two affine matrix expressions, since this would result in a matrix expression that depends quadratically on the variables.

`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` is just A; `-A` denotes the negative of 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.

Since all expressions are affine functions of the variables, we have the further condition that either A or B must be a constant.

- `A/a` denotes scalar multiplication of the matrix A by $1/a$, where a is a scalar constant expression with nonzero value.
- `A.*B` denotes componentwise (Schur) multiplication of A and B, *i.e.*, $(\texttt{A.*B})_{ij} = A_{ij}B_{ij}$. A and B must have the same dimensions and at least one of them has to be 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 dimensions and B must be a constant expression, with all entries having nonzero value.
- `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 dimensions.
  - `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 either a column or a 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, 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.

`sdpsol` provides the following functions:

- `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 main diagonal.
  - *Extracting the diagonal of a matrix.* If `A` is a matrix but not a row or column vector, `diag(A)` means a *column* vector made of the main diagonal entries of A.
- `Tr(A)` denotes the trace of A, *i.e.*, the sum of the diagonal entries of A. A has to be *square*; `Tr(A)` is a scalar.
- `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 dimensions, and one must be a constant expression. `ip(A,B)` is a scalar.
- `cols(A)` gives the number of columns of A. `cols(A)` is a (positive, integer) scalar.
- `rows(A)` gives the number of rows of A. `rows(A)` is a (positive, integer) scalar.
- `reshape(A,r,c)` denotes the $r \times c$ matrix expression with its elements taken columnwise 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.
- `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, *i.e.*, `ones(1,rows(A))*A`.

`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 §2.2). 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*). 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.).

2.4 Optimization parameters

Several internal variables are reserved by `sdpsol` to serve as optimization parameters. See [VB94a, VB94b] for the precise meanings of these parameters, and recommended ranges of values. The value of these reserved internal variables can be reassigned by the user via assignments. The meanings and default values are:

`NU` The parameter ν controls the rate of convergence of `sp`. Default value is 10.

`MAXITER` The maximum number of iterations allowed. Default value is 50.

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

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

`BIGM` The parameter M used in the pashe I big-M method. Default value is $10^3\kappa$.

2.5 Constraints and objective

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

```
A1'*P+P*A1 + B*D*B' < 0;
Tr(D)==1;
```

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. 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 dimensions. 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* expression and A is *square*. A similar rule applies to `A < k`. Thus the `sdpsol` constraint `A > 0` follows standard mathematical notation. (The same constraint can also be written as `A > 1-1`, which doesn't look very standard!)

`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 dimensions.

`A .> k` means the componentwise inequality $A_{ij} > k$, where k is 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 $A_{ij} > B_{ij}$.

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

```
minimize trace = Tr(P);
```

which is taken from our example. This statement assigns the internal variable *trace* to be the trace of P and tells `sdpsol` to minimize it. The syntax is

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

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.

2.6 Commands

The command statement `include("`*filename*`")`; includes two types of files into the workspace of `sdpsol`: Matlab binary data files (.mat files) and `sdpsol` source files. If the *filename* ends in *.mat*, `sdpsol` treats the file as a Matlab binary file and loads all the data as internal variables. Otherwise the file is included as part of the SDP specification file. Includes can nest up to a maximum of 10 layers.

`what(`*expression*`)`; prints to the standard output the dimensions, attributes, 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(B);` prints

```
3x2  internal variable, constant with value:
[    1.0000,    2.0000;
     3.0000,    4.0000;
     5.0000,    6.0000 ]
```

and `what(A1'*P+P*A1 + B*D*B');` prints

```
3x3  expression, affine in:
  P
  D
```

2.7 Dual SDP and alternative constraint specifications

`sdpsol` simultaneously solves the SDP (2) and its dual

$$\begin{array}{ll} \text{maximize} & -\mathbf{Tr}\, F_0 Z + b^T w \\ \text{subject to} & Z = Z^T > 0 \\ & \mathbf{Tr}\, F_i Z + (A^T w)_i = c_i, \qquad i = 1, \ldots, m, \end{array} \tag{3}$$

where Z and w are the variables. The dual variables Z and w also have matrix structure: we can associate with each LMI constraint a dual variable that is a symmetric matrix; with each equality constraint we can associate a dual variable of the same dimensions as the expressions in the equality constraint.

`sdpsol` allows the user to initialize dual variables, and will report the optimal values of dual variables, provided the constraints are named. The syntax of a named constraint is:

> **`constraint`** *constraint_name expression rel_op expression* ;

The additional name on the constraint has several affects. First, it assigns an associated **residual**, which is an internal variable with the name *constraint_name*_res. (*constraint_name* postfixed by "_res"). Given expressions `A` and `B`, the residual of the constraint `A==B`, `A.>B`, `B.<A`, `A>B` or `B<A` is defined to be `A-B`. After optimization `sdpsol` will report the values of all residuals.

Naming the constraint also creates an associated **dual variable**, which has the name *constraint_name*_dual (*constraint_name* postfixed by "_dual") and the same dimensions as the associated residual. Dual variables can be initialized using initialization statements after they are declared. For example, the statement

```
constraint LYAP   A'*P + P*A < 0 ;
initialize LYAP_dual = Qhat;
```

assigns the name `LYAP_res` to $-(A^TP + PA)$, and initializes the dual variable `LYAP_dual` to *Qhat* for the optimization phase. (We assume here that *Qhat* and A are constant, and P is a variable.) After optimization `sdpsol` will report the values of dual variables if *all* the constraints are named.

3 Usage

`sdpsol` is invoked as

`sdpsol` [*options*] *src_filename*

Options are Options are

−h Show the usage of `sdpsol`.

−q Quiet mode. Suppress compilation and run-time log to standard output.

−m [*filename*] Export the results of `sdpsol` to the Matlab binary file *filename1*. If *filename* is not given, *filename*.mat is used by default.

−o [*filename*] Redirect the `sdpsol` result report from the standard output (default) to the text file *filename*. If *filename* is not given, *src_filename*.out is used.

When the **−m** option is specified, a 2×1 matrix `SDPINFO` is written to the *.mat* file along with all the variables and objective. The possible values of `SDPINFO` and their meanings are given in Table 1.

SDPINFO(1,1)	SDPINFO(2,1)	prob. type	result
	0	feas	initial value of variables *leads to* a feasible solution
	1	feas/opt	maximum number of iterations reached
	2	feas/opt	absolute tolerance was reached
	3	opt	relative tolerance was reached
	4	feas	feasible solution was found
	5	feas/opt	proof of infeasibility was found
	6	feas/opt	the only possible solution was checked
	7	feas/opt	strictly primal feasibility can't be insured
-1	5,6	feas/opt	problem infeasible
0	7	feas/opt	feasibility can *not* be determined
1	0,2,4,6	feas	problem feasible
2	1	opt	maximum number of iterations reached
2	2,3,6	opt	optimal solution found
feas: feasibility problem opt: optimization problem			

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

4 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.) Future versions of `sdpsol` will surely exploit the matrix structure in the optimization phase.

`sdpsol` solves the problem (2), with strict LMI. 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. Instead it will stop in phase I after reaching the absolute tolerance or the maximum number of iterations. Indeed, this stopping condition gives a hint that your problem does not have a strictly feasible point. You can run `sdpsol` on the example file `not-stricly-feas` to see what happens in such a case.

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.

[VB94a] L. Vandenberghe and S. Boyd. Semidefinite programming. Submitted to *SIAM Review*, July 1994.

[VB94b] 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. Primal-dual potential reduction method for problems involving matrix inequalities. To be published in *Math. Programming*, 1995.