SDPT3 — a MATLAB software package for semidefinite-quadratic-linear programming, version 3.0

R. H. Tütüncü [*], K. C. Toh [†], and M. J. Todd [‡]

August 22, 2001

Abstract

This document describes a new release, version 3.0, of the software SDPT3. This code is designed to solve conic programming problems whose constraint cone is a product of semidefinite cones, second-order cones, and/or nonnegative orthants. It employs a predictor-corrector primal-dual path-following method, with either the HKM or the NT search direction. The basic code is written in Matlab, but key subroutines in Fortran and C are incorporated via Mex files. Routines are provided to read in problems in either SeDuMi or SDPA format. Sparsity and block diagonal structure are exploited, but the latter needs to be given explicitly.

[*]Department of Mathematical Sciences, Carnegie Mellon University, Pittsburgh, PA 15213, USA (`reha+@andrew.cmu.edu`). Research supported in part by NSF through grant CCR-9875559.

[†]Department of Mathematics, National University of Singapore, 10 Kent Ridge Crescent, Singapore 119260. (`mattohkc@math.nus.edu.sg`). Research supported in part by the Singapore-MIT Alliance.

[‡]School of Operations Research and Industrial Engineering, Cornell University, Ithaca, New York 14853, USA (`miketodd@cs.cornell.edu`). Research supported in part by NSF through grant DMS-9805602 and ONR through grant N00014-96-1-0050.

1 Introduction

The current version of SDPT3, version 3.0, can solve conic linear optimization problems with inclusion constraints for the cone of positive semidefinite matrices, the second-order cone, and/or the polyhedral cone of nonnegative vectors. It solves the following standard form of such problems, henceforth called SQLP problems:

$$\begin{array}{llccccccc} (P) & \min & \sum_{j=1}^{n_s} \langle c_j^s,\, x_j^s\rangle & + & \sum_{i=1}^{n_q} \langle c_i^q,\, x_i^q\rangle & + & \langle c^l,\, x^l\rangle & & \\ & \text{s.t.} & \sum_{j=1}^{n_s} (A_j^s)^T \mathbf{svec}(x_j^s) & + & \sum_{i=1}^{n_q} (A_i^q)^T x_i^q & + & (A^l)^T x^l & = & b, \\ & & x_j^s \in K_s^{s_j} \;\forall j, & & x_i^q \in K_q^{q_i} \;\forall i, & & x^l \in K_l^{n_l}. & & \end{array}$$

Here, c_j^s, x_j^s are symmetric matrices of dimension s_j and $K_s^{s_j}$ is the cone of positive semidefinite symmetric matrices of the same dimension. Similarly, c_i^q, x_i^q are vectors in $I\!R^{q_i}$ and $K_q^{q_i}$ is the second-order cone defined by $K_q^{q_i} := \{x \in I\!R^{q_i} : x_1 \geq \|x_{2:q_i}\|\}$. Finally, c^l, x^l are vectors of dimension n_l and $K_l^{n_l}$ is the cone $I\!R_+^{n_l}$. In the notation above, A_j^s denotes the $\bar{s}_j \times m$ matrix with $\bar{s}_j = s_j(s_j+1)/2$ whose columns are obtained using the **svec** operator from m symmetric $s_j \times s_j$ constraint matrices corresponding to the jth semidefinite block x_j^s. For a definition of the vectorization operator **svec** on symmetric matrices, see, e.g., [15]. The matrices A_i^q's are $q_i \times m$ dimensional constraint matrices corresponding to the ith quadratic block x_i^q, and A^l is the $l \times m$ dimensional constraint matrix corresponding to the linear block x^l. The notation $\langle p,\, q\rangle$ denotes the standard inner product in the appropriate space.

The software also solves the dual problem associated with the problem above:

$$\begin{array}{lllllll} (D) & \max & b^T y & & & & \\ & \text{s.t.} & A_j^s y & + & z_j^s & = & c_j^s, \quad j = 1\ldots, n_s \\ & & A_i^q y & + & z_i^q & = & c_i^q, \quad i = 1\ldots, n_q \\ & & A^l y & + & z^l & = & c^l, \\ \multicolumn{7}{c}{z_j^s \in K_s^{s_j} \;\forall j, \quad z_i^q \in K_q^{q_i} \;\forall i, \quad z^l \in K_l^{n_l}.} \end{array}$$

This package is written in MATLAB version 5.3 and is compatible with MATLAB version 6.0. It is available from the internet sites:

```
http://www.math.nus.edu.sg/~mattohkc/index.html
http://www.math.cmu.edu/~reha/sdpt3.html
```

The software package was originally developed to provide researchers in semidefinite programming with a collection of reasonably efficient and robust algorithms that could solve general SDPs with matrices of dimensions of the order of a hundred. The current release, version 3.0, expands the family of problems solvable by the software in two dimensions. First, this version is much faster than the previous release [18], especially on large sparse problems, and consequently can solve much larger problems. Second, the current release can also directly solve problems that have second-order cone constraints — with the previous version it was necessary to convert such constraints to semidefinite cone constraints.

In this paper, the vector 2-norm and Frobenius norm are denoted by $\|\cdot\|$ and $\|\cdot\|_F$, respectively. In the next section, we discuss the algorithm used in the software and several implementation details including the initial iterates generated by our software and its data storage scheme. Section 3 describes the search directions used by our algorithms and explains how they are computed. In Section 4, we provide sample runs and comment on several major differences between the current and earlier versions of our software. In the last section, we present performance results of our software on problems from the SDPLIB and DIMACS libraries.

2 A primal-dual infeasible-interior-point algorithm

The algorithm implemented in SDPT3 is a primal-dual interior-point algorithm that uses the path-following paradigm. In each iteration, we first compute a *predictor* search direction aimed at decreasing the duality gap as much as possible. After that, the algorithm generates a Mehrotra-type corrector step [10] with the intention of keeping the iterates close to the central path. However, we do not impose any neighborhood restrictions on our iterates.[1] Initial iterates need not be feasible — the algorithm tries to achieve feasibility and optimality of its iterates simultaneously. It should be noted that in our implementation, the user has the option to use a primal-dual path-following algorithm that does not use corrector steps.

Following next is a **pseudo-code** for the algorithm we implemented. Note that this description makes references to later sections where details related to the algorithm are explained.

Algorithm IPC. *Suppose we are given an initial iterate (x^0, y^0, z^0) with x^0, z^0 strictly satisfying all the conic constraints. Decide on the type of search direction to use. Set $\gamma^0 = 0.9$. Choose a value for the parameter* expon *used in e.*

For $k = 0, 1, \ldots$

(Let the current and the next iterate be (x, y, z) and (x^+, y^+, z^+) respectively. Also, let the current and the next step-length parameter be denoted by γ and γ^+ respectively.)

- *Set $\mu = \langle x,\, z\rangle / n$, and*

$$\texttt{rel\_gap} \;=\; \frac{\langle x,\, z\rangle}{\max(1, (|\langle c,\, x\rangle| + |b^T y|)/2}, \quad \texttt{infeas\_meas} \;=\; \max\left(\frac{\|r_p\|}{\max(1, \|b\|)}, \frac{\|R_d\|}{\max(1, \|c\|)}\right). \quad (1)$$

  Stop the iteration if the infeasibility measure `infeas_meas` *and the relative duality gap (*`rel_gap`*) are sufficiently small.*

[1] This strategy works well on most of the problems we tested. However, it should be noted that the occasional failure of the software on problems with poorly chosen initial iterates is likely due to the lack of a neighborhood enforcement in the algorithm.

- *(Predictor step)*
  Solve the linear system (9) with $\sigma = 0$ *in the right-side vector (11). Denote the solution of (3) by* $(\delta x, \delta y, \delta z)$. *Let* α_p *and* β_p *be the step-lengths defined as in (29) and (30) with* $\Delta x, \Delta z$ *replaced by* $\delta x, \delta z$, *respectively.*
- *Take* σ *to be*

$$\sigma \;=\; \min\left(1,\; \left[\frac{\langle x + \alpha_p\, \delta x,\; z + \beta_p\, \delta z\rangle}{\langle x,\; z\rangle}\right]^e\right),$$

  where the exponent e *is chosen as follows:*

$$e \;=\; \begin{cases} \max[expon, 3\min(\alpha_p, \beta_p)^2] & \text{if } \mu > 10^{-6}, \\ expon & \text{if } \mu \le 10^{-6}. \end{cases}$$

- *(Corrector step)*
  Solve the linear system (9) with R_c *in the the right-hand side vector (11) replaced by*

$$\begin{aligned} \tilde{R}_c^s &= \mathbf{svec}\left[\sigma\mu I - H_P(\mathbf{smat}(x^s)\mathbf{smat}(z^s)) - H_P(\mathbf{smat}(\delta x^s)\mathbf{smat}(\delta z^s))\right] \\ \tilde{R}_c^q &= \sigma\mu e^q - T_G(x^q, z^q) - T_G(\delta x^q, \delta z^q) \\ \tilde{R}_c^l &= \sigma\mu e^l - diag(x^l)z^l - diag(\delta x^l)\delta z^l. \end{aligned}$$

  Denote the solution of (3) by $(\Delta x, \Delta y, \Delta z)$.
- *Update* (x, y, z) *to* (x^+, y^+, z^+) *by*

$$x^+ \;=\; x + \alpha\,\Delta x, \quad y^+ \;=\; y + \beta\,\Delta y, \quad z^+ \;=\; z + \beta\,\Delta z,$$

  where α *and* β *are computed as in (29) and (30) with* γ *chosen to be* $\gamma = 0.9 + 0.09\min(\alpha_p, \beta_p)$.
- *Update the step-length parameter by*

$$\gamma^+ \;=\; 0.9 + 0.09\min(\alpha, \beta).$$

The main routine that corresponds to the infeasible path-following algorithm just described is `sqlp.m`:

```
[obj,X,y,Z,gaphist,infeashist,info,Xiter,yiter,Ziter] =
sqlp(blk,A,C,b,X0,y0,Z0,OPTIONS).
```

Input arguments.

`blk`: a cell array describing the block structure of the SQLP problem.

`A`, `C`, `b`: SQLP data.

`X0`, `y0`, `Z0`: an initial iterate.

`OPTIONS`: a structure array of parameters.

If the input argument OPTIONS is omitted, default values are used.

Output arguments.

The names chosen for the output arguments explain their contents. The argument info is a 5-vector containing performance information; see [18] for details. The argument (Xiter,yiter,Ziter) is new in this release: it is the last iterate of sqlp.m, and if desired, the user can continue the iteration process with this as the initial iterate. Such an option allows the user to iterate for a certain amount of time, stop to analyze the current solution, and continue if necessary. This can be achieved, for example, by choosing a small value for the maximum number iterations specified in OPTIONS.maxit.

Note that, while (X,y,Z) normally gives approximately optimal solutions, if info(1) is 1 the problem is suspected to be primal infeasible and (y,Z) is an approximate certificate of infeasibility, with $\mathtt{b}^T\mathtt{y} = 1$, Z in the appropriate cone, and $\mathtt{A}^T\mathtt{y} + \mathtt{Z}$ small, while if info(1) is 2 the problem is suspected to be dual infeasible and X is an approximate certificate of infeasibility, with $\langle \mathtt{C},\, \mathtt{X}\rangle = -1$, X in the appropriate cone, and AX small.

A structure array for parameters.

The function sqlp.m uses a number of parameters which are specified in a MATLAB structure array called OPTIONS in the m-file parameters.m. If desired, the user can change the values of these parameters. The meaning of the specified fields in OPTIONS are given in the m-file itself. As an example, if the user does not wish to use corrector steps in Algorithm IPC, then he/she can do so by setting OPTIONS.predcorr = 0. Similarly, if the user wants to use a fixed value, say 0.98, for the step-length parameter γ instead of the adaptive strategy used in the default, he/she can achieve that by setting OPTIONS.gam = 0.98.

Stopping criteria.

The user can set a desired level of accuracy through the parameters OPTIONS.gaptol and OPTIONS.inftol (the default for each is 10^{-8}).. The algorithm is stopped when any of the following cases occur.

1. solutions with the desired accuracy have been obtained, i.e.,

$$\mathtt{rel\_gap} := \frac{\langle x,\, z\rangle}{\max\{1, (|\langle c,\, x\rangle| + |b^T y|)/2\}}$$

   and

$$\mathtt{infeas\_meas} := \max\left[\frac{\|Ax - b\|}{\max\{1, \|b\|\}}, \frac{\|A^T y + z - c\|}{\max\{1, \|c\|\}}\right]$$

   are both below OPTIONS.gaptol.

2. primal infeasibility is suggested because

$$b^T y / \|A^T y + z\| > 1/\mathtt{OPTIONS.inftol};$$

3. dual infeasibility is suggested because

$$-c^T x/\|Ax\| > 1/\texttt{OPTIONS.inftol};$$

4. slow progress is detected, measured by a rather complicated set of tests including

$$x^T z/n < 10^{-4} \quad \text{and} \quad \texttt{rel\_gap} < 5 * \texttt{infeas\_meas};$$

5. numerical problems are encountered, such as the iterates not being positive definite or the Schur complement matrix not being positive definite; or
6. the step sizes fall below 10^{-6}.

Initial iterates.

Our algorithms can start with an infeasible starting point. However, the performance of these algorithms is quite sensitive to the choice of the initial iterate. As observed in [4], it is desirable to choose an initial iterate that at least has the same order of magnitude as an optimal solution of the SQLP. If a feasible starting point is not known, we recommend that the following initial iterate be used:

$$\begin{aligned} y^0 &= 0, \\ (x_j^s)^0 &= \xi_j^s\, I_{s_j}, \quad (z_j^s)^0 = \eta_j^s\, I_{s_j}, \quad j = 1, \ldots, n_s, \\ (x_i^q)^0 &= \xi_i^q\, e_i^q, \quad (z_i^q)^0 = \eta_i^q\, e_i^q, \quad i = 1, \ldots, n_q, \\ (x^l)^0 &= \xi^l\, e^l, \quad (z^l)^0 = \eta^l\, e^l, \end{aligned}$$

where I_{s_j} is the identity matrix of order s_j, e_i^q is the first q_i-dimensional unit vector, e^l is the vector of all ones, and

$$\begin{aligned} \xi_j^s &= \sqrt{s_j}\max\left(1,\ \sqrt{s_j}\max_{1\le k\le m}\frac{1+|b_k|}{1+\|A_j^s(:,k)\|}\right), \\ \eta_j^s &= \sqrt{s_j}\max\left(1,\ \frac{1+\max(\max_k\{\|A_j^s(:,k)\|\},\ \|c_j^s\|_F)}{\sqrt{s_j}}\right), \\ \xi_i^q &= \sqrt{q_i}\max\left(1,\ \max_{1\le k\le m}\frac{1+|b_k|}{1+\|A_i^q(:,k)\|}\right), \\ \eta_i^q &= \sqrt{q_i}\max\left(1,\ \frac{1+\max(\max_k\{\|A_i^q(:,k)\|\},\ \|c_i^q\|)}{\sqrt{q_i}}\right), \\ \xi^l &= \max\left(1,\ \max_{1\le k\le m}\frac{1+|b_k|}{1+\|A^l(:,k)\|}\right), \\ \eta^l &= \max\left(1,\ \frac{1+\max(\max_k\{\|A^l(:,k)\|\},\ \|c^l\|)}{\sqrt{n_l}}\right), \end{aligned}$$

where $A^s_j(:,k)$ denotes the kth column of A^s_j, and $A^q_i(:,k)$ and $A^l_i(:,k)$ are defined similarly.

By multiplying the identity matrix I_{s_i} by the factors ξ^s_i and η^s_i for the semidefinite blocks, and similarly for the quadratic and linear blocks, the initial iterate has a better chance of having the appropriate order of magnitude.

The initial iterate above is set by calling `infeaspt.m`, with initial line

```
function  [X0,y0,Z0] = infeaspt(blk,A,C,b,options,scalefac),
```

where `options = 1 (default)` corresponds to the initial iterate just described, and `options = 2` corresponds to the choice where the blocks of `X0`, `Z0` are `scalefac` times identity matrices or unit vectors, and `y0` is a zero vector.

Cell array representation for problem data.

Our implementation SDPT3 exploits the block structure of the given SQLP problem. In the internal representation of the problem data, we classify each semidefinite block into one of the following two types:

1. a dense or sparse matrix of dimension greater than or equal to 30;
2. a sparse block-diagonal matrix consisting of numerous sub-blocks each of dimension less than 30.

The reason for using the sparse matrix representation to handle the case when we have numerous small diagonal blocks is that it is less efficient for MATLAB to work with a large number of cell array elements compared to working with a single cell array element consisting of a large sparse block-diagonal matrix. Technically, no problem will arise if one chooses to store the small blocks individually instead of grouping them together as a sparse block-diagonal matrix.

For the quadratic part, we typically group all quadratic blocks (small or large) into a single block, though it is not mandatory to do so. If there are a large number of small blocks, it is advisable to group them all together as a single large block consisting of numerous small sub-blocks for the same reason we mentioned before.

Let $L = n_s + n_q + 1$. For each SQLP problem, the block structure of the problem data is described by an $L \times 2$ cell array named `blk`. The content of each of the elements of the cell arrays is given as follows. If the jth block is a semidefinite block consisting of a single block of size $\mathtt{s_j}$, then

$$\mathtt{blk\{j,1\}} = \mathtt{'s'}, \quad \mathtt{blk\{j,2\}} = [\mathtt{s_j}],$$

$$\mathtt{A\{j\}} = [\bar{\mathtt{s}}_\mathtt{j} \, \mathtt{x}\, \mathtt{m} \ \ \mathtt{sparse}],$$

$$\mathtt{C\{j\}},\ \mathtt{X\{j\}},\ \mathtt{Z\{j\}} = [\mathtt{s_j}\,\mathtt{x}\,\mathtt{s_j} \ \ \mathtt{double\ or\ sparse}],$$

where $\bar{\mathtt{s}}_\mathtt{j} = \mathtt{s_j}(\mathtt{s_j}+1)/2$.

If the jth block is a semidefinite block consisting of numerous small sub-blocks, say p of them, of dimensions $\mathtt{s_{j1}}, \mathtt{s_{j2}}, \ldots, \mathtt{s_{jp}}$ such that $\sum_{k=1}^{p} \mathtt{s_{jk}} = \mathtt{s_j}$, then

$$\texttt{blk\{j,1\} = 's'}, \quad \texttt{blk\{j,2\}} = [\mathtt{s_{j1}}\ \ \mathtt{s_{j2}}\ \ \cdots\ \ \mathtt{s_{jp}}],$$

$$\texttt{A\{j\}} = [\bar{\mathtt{s}}_{\mathtt{j}}\ \mathtt{x}\, \mathtt{m}\ \ \texttt{sparse}],$$

$$\texttt{C\{j\}},\ \texttt{X\{j\}},\ \texttt{Z\{j\}} = [\mathtt{s_j}\ \mathtt{x}\ \mathtt{s_j}\ \ \texttt{sparse}]\ ,$$

where $\bar{\mathtt{s}}_{\mathtt{j}} = \sum_{\mathtt{k}=1}^{\mathtt{p}} \mathtt{s_{jk}}(\mathtt{s_{jk}}+1)/2$.

The above storage scheme for the data matrix A_j^s associated with the semidefinite blocks of the SQLP problem represents a departure from earlier versions (version 2.x or earlier) of our implementation, such as the one described in [18]. Previously, the semidefinite part of `A` was represented by an $n_s \times m$ cell array, where `A{j,k}` corresponds to the kth constraint matrix associated with the jth semidefinite block, and it was stored as an individual matrix in either dense or sparse format. Now, we store all the constraint matrices associated with the jth semidefinite block in vectorized form as a single $\bar{s}_j \times m$ matrix where the kth column of this matrix corresponds to the kth constraint matrix. That is, `A{j}(:,k) = svec(kth constraint matrix associated with the jth semidefinite block)`. The data format we used in earlier versions of SDPT3 was more natural, but our current data representation was adopted for the sake of computational efficiency. The reason for such a change is again due to the fact that it is less efficient for MATLAB to work with a single cell array with many cells.

The data storage scheme corresponding to quadratic and linear blocks is rather straightforward. If the ith block is a quadratic block consisting of numerous subblocks, say p of them, of dimensions $\mathtt{q_{i1}}, \mathtt{q_{i2}}, \ldots, \mathtt{q_{ip}}$ such that $\sum_{k=1}^{p} \mathtt{q_{ik}} = \mathtt{q_i}$, then

$$\texttt{blk\{i,1\} = 'q'}, \quad \texttt{blk\{i,2\}} = [\mathtt{q_{i1}}\ \ \mathtt{q_{i2}}\ \ \cdots\ \ \mathtt{q_{ip}}],$$

$$\texttt{A\{i\}} = [\mathtt{q_i}\ \mathtt{x}\, \mathtt{m}\ \texttt{sparse}],$$

$$\texttt{C\{i\}},\ \texttt{X\{i\}},\ \texttt{Z\{i\}} = [\mathtt{q_i}\ \mathtt{x}\ 1\ \ \texttt{double or sparse}].$$

If the ith block is the linear block, then

$$\texttt{blk\{i,1\} = 'l'}, \quad \texttt{blk\{i,2\}} = \mathtt{n_l},$$

$$\texttt{A\{i\}} = [\mathtt{n_l}\ \mathtt{x}\, \mathtt{m}\ \ \texttt{sparse}],$$

$$\texttt{C\{i\}},\ \texttt{X\{i\}},\ \texttt{Z\{i\}} = [\mathtt{n_l}\ \mathtt{x}\ 1\ \ \texttt{double or sparse}].$$

Notice that we associated with each constraint a column in `A{j}` rather than the usual linear programming practice of associating with it a row. The reason is to avoid the need to take the transpose of `A{j}` excessively, which can incur a significant amount of CPU time in MATLAB when `A{j}` is a large sparse matrix.

3 The search direction

To simplify discussion, we introduce the following notation, which is also consistent with the internal data representation in SDPT3:

$$A^s = \begin{bmatrix} A_1^s \\ \vdots \\ A_{n_s}^s \end{bmatrix}, \quad A^q = \begin{bmatrix} A_1^q \\ \vdots \\ A_{n_q}^q \end{bmatrix}.$$

Similarly, we define

$$x^s = \begin{bmatrix} \mathbf{svec}(x_1^s) \\ \vdots \\ \mathbf{svec}(x_{n_s}^s) \end{bmatrix}, \quad x^q = \begin{bmatrix} x_1^q \\ \vdots \\ x_{n_q}^q \end{bmatrix}. \tag{2}$$

The vectors c^s, z^s, c^q, and z^q are defined analogously. We will use corresponding notation for the search directions as well. Finally, let

$$A = \begin{bmatrix} A^s \\ A^q \\ A^l \end{bmatrix}, \ x = \begin{bmatrix} x^s \\ x^q \\ x^l \end{bmatrix}, \ c = \begin{bmatrix} c^s \\ c^q \\ c^l \end{bmatrix}, \ z = \begin{bmatrix} z^s \\ z^q \\ z^l \end{bmatrix},$$

and

$$n = \sum_{j=1}^{n_s} s_j + \sum_{i=1}^{n_q} q_i + n_l.$$

Note that the matrix A above is defined as the transpose of that in the standard literature so as to be consistent with our data representation.

With the notation introduced above, the primal and dual equality constraints can be represented respectively as

$$A^T x = b, \quad Ay + z = c.$$

The primal-dual path-following algorithm we implemented assumes that A has full column rank. But in our software, the presence of (nearly) dependent constraints is detected automatically, and warning messages are displayed if such constraints exist. When this happens, the user has the option of removing these (nearly) dependent constraints by calling a preprocessing routine to remove them by setting `OPTIONS.rmdepconstr = 1`. We should mention that the routine we have coded for removing dependent constraints is a rather primitive one, and it is inefficient for large problems. We hope to improve on this routine in future versions of SDPT3.

The main step at each iteration of our algorithms is the computation of the search direction $(\Delta x, \Delta y, \Delta z)$ from the *symmetrized Newton equation* with respect to an invertible block diagonal scaling matrix P for the semidefinite block and an invertible block diagonal scaling matrix G for the quadratic block. The matrices P and G are usually chosen as functions of the current iterate x, z and we will elaborate on specific choices below. The search direction $(\Delta x, \Delta y, \Delta z)$ is obtained from the following system of equations:

$$
\begin{array}{lcrcccl}
 & A\Delta y & + \quad \Delta z & = & R_d & := & c - z - Ay \\
A^T\Delta x & & & = & r_p & := & b - A^T x \\
\mathcal{E}^s \Delta x^s & & + \quad \mathcal{F}^s \Delta z^s & = & R_c^s & := & \mathbf{svec}\left(\sigma\mu I - H_P\left(\mathbf{smat}(x^s)\mathbf{smat}(z^s)\right)\right) \\
\mathcal{E}^q \Delta x^q & & + \quad \mathcal{F}^q \Delta z^q & = & R_c^q & := & \sigma\mu e^q - T_G(x^q, z^q) \\
\mathcal{E}^l \Delta x^l & & + \quad \mathcal{F}^l \Delta z^l & = & R_c^l & := & \sigma\mu e^l - \mathcal{E}^l \mathcal{F}^l e^l,
\end{array}
\tag{3}
$$

where $\mu = \langle x,\, z\rangle / n$ and σ is the centering parameter. The notation **smat** denotes the inverse map of **svec** and both are to be interpreted as blockwise operators if the argument consists of blocks. Here H_P is the symmetrization operator whose action on the jth semidefinite block is defined by

$$
\begin{aligned}
H_{P_j} : I\!R^{s_j \times s_j} &\longrightarrow I\!R^{s_j \times s_j} \\
H_{P_j}(U) &= \tfrac{1}{2}\left[P_j U P_j^{-1} + P_j^{-T} U^T P_j^T\right],
\end{aligned}
\tag{4}
$$

with P_j the jth block of the block diagonal matrix P and $\mathcal{E}^s$ and $\mathcal{F}^s$ are symmetric block diagonal matrices whose jth blocks are given by

$$
\mathcal{E}_j^s = P_j \circledast P_j^{-T} z_j^s, \qquad \mathcal{F}_j^s = P_j x_j^s \circledast P_j^{-T},
\tag{5}
$$

where $R \circledast T$ is the symmetrized Kronecker product operation described in [15].

In the quadratic block, e^q denotes the blockwise identity vector, i.e.,

$$
e^q = \begin{bmatrix} e_1^q \\ \vdots \\ e_{n_q}^q \end{bmatrix},
$$

where e_i^q is the first unit vector in $I\!R^{q_i}$. Let the arrow operator defined in [2] be denoted by $\mathbf{Arw}(\cdot)$. Then the operator T_G is defined as follows:

$$
T_G(x^q, z^q) = \begin{bmatrix} \mathbf{Arw}\left(G_1 x_1^q\right)\left(G_1^{-1} z_1^q\right) \\ \vdots \\ \mathbf{Arw}\left(G_{n_q} x_{n_q}^q\right)\left(G_{n_q}^{-1} z_{n_q}^q\right) \end{bmatrix},
\tag{6}
$$

where G is a symmetric block diagonal matrix that depends on x, z and G_i is the ith block of G. The matrices $\mathcal{E}^q$ and $\mathcal{F}^q$ are block diagonal matrices whose the ith blocks are given by

$$
\mathcal{E}_i^q = \mathbf{Arw}\left(G_i^{-1} z_i^q\right) G_i, \quad \mathcal{F}_i^q = \mathbf{Arw}\left(G_i x_i^q\right) G_i^{-1}.
\tag{7}
$$

In the linear block, e^l denotes the n_l-dimensional vector of ones, and $\mathcal{E}^l = \text{diag}(x^l)$, $\mathcal{F}^l = \text{diag}(z^l)$.

For future reference, we partition the vectors R_d, Δx, and Δz in a manner analogous to c, x, and z as follows:

$$R_d = \begin{bmatrix} R_d^s \\ R_d^q \\ R_d^l \end{bmatrix}, \quad \Delta x = \begin{bmatrix} \Delta x^s \\ \Delta x^q \\ \Delta x^l \end{bmatrix}, \quad \Delta z = \begin{bmatrix} \Delta z^s \\ \Delta z^q \\ \Delta z^l \end{bmatrix}. \tag{8}$$

Assuming that $m = \mathcal{O}(n)$, we compute the search direction via a Schur complement equation as follows (the reader is referred to [1] and [15] for details). First compute Δy from the Schur complement equation

$$M\Delta y \;=\; h, \tag{9}$$

where

$$M \;=\; (A^s)^T(\mathcal{E}^s)^{-1}\mathcal{F}^s A^s + (A^q)^T(\mathcal{E}^q)^{-1}\mathcal{F}^q A^q + (A^l)^T(\mathcal{E}^l)^{-1}\mathcal{F}^l A^l \tag{10}$$

$$\begin{aligned} h \;=\; & r_p - (A^s)^T(\mathcal{E}^s)^{-1}(R_c^s - \mathcal{F}^s R_d^s) \\ & - (A^q)^T(\mathcal{E}^q)^{-1}(R_c^q - \mathcal{F}^q R_d^q) \;-\; (A^l)^T(\mathcal{E}^l)^{-1}(R_c^l - \mathcal{F}^l R_d^l). \end{aligned} \tag{11}$$

Then compute Δx and Δz from the equations

$$\Delta z \;=\; R_d - A\Delta y \tag{12}$$

$$\Delta x^s \;=\; (\mathcal{E}^s)^{-1}R_c^s - (\mathcal{E}^s)^{-1}\mathcal{F}^s\Delta z^s \tag{13}$$

$$\Delta x^q \;=\; (\mathcal{E}^q)^{-1}R_c^q - (\mathcal{E}^q)^{-1}\mathcal{F}^q\Delta z^q \tag{14}$$

$$\Delta x^l \;=\; (\mathcal{E}^l)^{-1}R_c^l - (\mathcal{E}^l)^{-1}\mathcal{F}^l\Delta z^l. \tag{15}$$

3.1 Two choices of search directions

We start by introducing some notation that we will use in the remainder of this paper. For a given q_i-dimensional vector x_i^q, we let x_i^0 denote its first component and x_i^1 denote its subvector consisting of the remaining entries, i.e.,

$$\begin{bmatrix} x_i^0 \\ x_i^1 \end{bmatrix} = \begin{bmatrix} (x_i^q)_1 \\ (x_i^q)_{2:q_i} \end{bmatrix}. \tag{16}$$

We will use the same convention for z_i^q, Δx_i^q, etc. Also, we define the following function from $K_q^{q_i}$ to $I\!R_+$:

$$\gamma(x_i^q) \;:=\; \sqrt{(x_i^0)^2 - \langle x_i^1, x_i^1\rangle}. \tag{17}$$

Finally, we use X and Z for $\mathbf{smat}(x^s)$ and $\mathbf{smat}(z^s)$, where the operation is applied blockwise to form a block diagonal symmetric matrix of order $\sum_{j=1}^{n_s} s_j$.

In the current release of this package, the user has two choices of scaling operators parametrized by P and G, resulting in two different search directions: the HKM direction [7, 9, 11], and the NT direction [14]. See also Tsuchiya [20] for the second-order case.

(1) **The HKM direction.** This choice uses the scaling matrix $P = Z^{1/2}$ for the semidefinite blocks and a symmetric block diagonal scaling matrix G for the quadratic blocks where the ith block G_i is given by the following equation:

$$G_i \;=\; \begin{bmatrix} z_i^0 & (z_i^1)^T \\ z_i^1 & \gamma(z_i^q)I + \dfrac{z_i^1(z_i^1)^T}{\gamma(z_i^q)+z_i^0} \end{bmatrix}. \tag{18}$$

(2) **The NT direction.** This choice uses the scaling matrix $P = N^{-1}$ for the semidefinite blocks, where N is a matrix such that $D := N^T Z N = N^{-1} X N^{-T}$ is a diagonal matrix [15], and G is a symmetric block diagonal matrix whose ith block G_i is defined as follows. Let

$$\omega_i = \sqrt{\frac{\gamma(z_i^q)}{\gamma(x_i^q)}}, \quad \xi_i = \begin{bmatrix} \xi_i^0 \\ \xi_i^1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\omega_i} z_i^0 + \omega_i x_i^0 \\ \frac{1}{\omega_i} z_i^1 - \omega_i x_i^1 \end{bmatrix}. \tag{19}$$

Then

$$G_i = \omega_i \begin{bmatrix} t_i^0 & (t_i^1)^T \\ t_i^1 & I + \dfrac{t_i^1(t_i^1)^T}{1+t_i^0} \end{bmatrix}, \quad \text{where} \quad \begin{bmatrix} t_i^0 \\ t_i^1 \end{bmatrix} = \frac{1}{\gamma(\xi_i)} \begin{bmatrix} \xi_i^0 \\ \xi_i^1 \end{bmatrix}. \tag{20}$$

3.2 Computation of the search directions

Generally, the most expensive part in each iteration of Algorithm IPC lies in the computation and factorization of the Schur complement matrix M defined in (9). And this depends critically on the size and density of M. Note that the density of this matrix depends on two factors: (i) The density of the constraint coefficient matrices A^s, A^q, and A^l, and (ii) any additional fill-in introduced because of the terms $(\mathcal{E}^s)^{-1}\mathcal{F}^s$, $(\mathcal{E}^q)^{-1}\mathcal{F}^q$, and $(\mathcal{E}^l)^{-1}\mathcal{F}^l$ in (9).

3.2.1 Semidefinite blocks

For problems with semidefinite blocks, the contribution by the jth semidefinite block to M is given by $M_j^s := (A_j^s)^T(\mathcal{E}_j^s)^{-1}\mathcal{F}_j^s A_j^s$. As the matrix $(\mathcal{E}_j^s)^{-1}\mathcal{F}_j^s$ is dense and structure-less for most problems, the matrix M_j^s is generally dense even if A_j^s is sparse. The computation of each entry of M_j^s involves matrix products, which in the case of NT direction has the form

$$(M_j^s)_{\alpha\beta} \;=\; A_j^s(:,\alpha)^T \mathbf{svec}\left(w_j^s\, \mathbf{smat}(A_j^s(:,\beta))\, w_j^s\right),$$

where $A_j^s(:,k)$ denotes the kth column of A_j^s. This computation can be very expensive if it is done naively without properly the exploiting sparsity that is generally present in A_j^s. In our earlier papers [15, 18], we discussed briefly how sparsity of A_j^s is exploited in our implementation by following the ideas presented in [4]. However, as the efficient implementation of these ideas is not spelled out in the current literature, we will provide further details here.

In our implementation, firstly the matrix A_j^s is sorted column-wise in ascending order of the number of non-zero elements in each column. Suppose we denote the sorted matrix by $\tilde{A}_j^s$ and the matrix $(\tilde{A}_j^s)^T(\mathcal{E}_j^s)^{-1}\mathcal{F}_j^s\tilde{A}_j^s$ by $\tilde{M}_j^s$. (We should emphasize that, in order to cut down memory usage, the matrix $\tilde{A}_j^s$ is not created explicitly, but it is accessed through A_j^s via a permutation vector, and similarly for $\tilde{M}_j^s$.) Then a 2-column cell array `nzlistA` is created such that `nzlistA{j,1}` is an m-vector where `nzlistA{j,1}(k)` is the starting row index in the 2-column matrix `nzlistA{j,2}` that stores the row and column indices of the non-zero elements of $\mathbf{smat}(\tilde{A}_j^s(:,k))$. If the number of non-zero elements of $\mathbf{smat}(\tilde{A}_j^s(:,k))$ exceeds a certain threshold, we set `nzlistA{j,1}(k) = inf`, and we do not append the row and column indices of this matrix to `nzlistA{j,2}`. We use the flag "`inf`" to indicate that the density of the matrix is too high for sparse computation to be done efficiently. If J is the largest integer for which `nzlistA{j,1}(k) < inf`, then we compute the upper triangular part of matrix $\tilde{M}_j^s(1:J,1:J)$ through formula $\mathcal{F}$-3 described in [4]. In our software, this part of the computation is done in a C Mex routine `mexschur.mex*`.

As formula $\mathcal{F}$-3 is efficient only for very sparse constraint matrices, say with density below the level of 2%, we also need to handle the case where constraint matrices have a moderate level of density, say 2% − 20%. For such matrices, the strategy is to compute only those elements of the matrix product $U := w_j^s\,\mathbf{smat}(\tilde{A}_j^s(:,\beta))\,w_j^s$ that contribute to an entry of $\tilde{M}_j^s$, that is, those that correspond to a nonzero entry of $\tilde{A}_j^s(:,k)$ for some $k = 1,\ldots,\beta$. Basically, we use formula $\mathcal{F}$-2 described in [4] in this case. In our implementation, we create another 2-column cell array `nzlistAsum` to facilitate such calculations. In this case, `nzlistAsum{j,1}` is a vector such that `nzlistAsum{j,1}(k)` is the starting row index in the 2-column matrix `nzlistAum{j,2}` that stores the row and column indices of the non-zero elements of $\mathbf{smat}(\tilde{A}_j^s(:,k))$ that are not already present in the combined list of non-zero elements from $\sum_{i=1}^{k-1}|\mathbf{smat}(\tilde{A}_j^s(:,i))|$. Again, if the combined list of non-zero elements of $\sum_{i=1}^{k}|\mathbf{smat}(\tilde{A}_j^s(:,i))|$ exceeds a certain threshold, we set `nzlistAsum{j,1}(k) = inf` since formula $\mathcal{F}$-2 in [4] is not efficient when the combined list of non-zeros elements that need to be calculated for U is too large. Suppose L is the largest integer such that `nzlistAsum{j,1}(k) < inf`. Then we compute the upper triangular part of $\tilde{M}_j^s(1:L,J:L)$ through formula $\mathcal{F}$-2. Again, for efficiency, we do the computation in a C Mex routine `mexProd2nz.mex*`.

The remaining columns of $\tilde{M}_j^s$ are calculated by first computing the full matrix U, and then taking the inner product between the appropriate columns of $\tilde{A}_j^s$ and $\mathbf{svec}(U)$.

Finally, we would like to highlight an issue that is often critical in cutting down the computation time in forming M_j^s. In many large SDP problems, the matrix $\mathbf{smat}(A_j^s(:,k))$ is usually sparse, and it is important to store this matrix as a sparse matrix in MATLAB and perform sparse-dense matrix-matrix multiplication whenever possible.

3.2.2 Quadratic and linear blocks

For linear blocks, $(\mathcal{E}^l)^{-1}\mathcal{F}^l$ is a diagonal matrix and it does not introduce any additional fill-in. This matrix does, however, affect the conditioning of the Schur complement matrix and is a popular subject of research in implementations of interior-point methods for linear programming.

From equation (10), it is easily shown that the contribution of the quadratic blocks to the matrix M is given by

$$M^q \;=\; (A^q)^T(\mathcal{E}^q)^{-1}\mathcal{F}^q A^q \;=\; \sum_{i=1}^{n_q} \underbrace{(A_i^q)^T(\mathcal{E}_i^q)^{-1}\mathcal{F}_i^q A_i^q}_{M_i^q}. \tag{21}$$

For the HKM direction, $(\mathcal{E}_i^q)^{-1}\mathcal{F}_i^q$ is a diagonal matrix plus a rank-two symmetric matrix, and we have

$$M_i^q \;=\; \frac{\langle x_i^q, z_i^q\rangle}{\gamma^2(z_i^q)}(A_i^q)^T J_i A_i^q \;+\; u_i^q(v_i^q)^T + v_i^q(u_i^q)^T, \tag{22}$$

where

$$J_i = \begin{bmatrix} -1 & 0 \\ 0 & I \end{bmatrix}, \quad u_i^q = (A_i^q)^T \begin{bmatrix} x_i^0 \\ x_i^1 \end{bmatrix}, \quad v_i^q = (A_i^q)^T \left(\frac{1}{\gamma^2(z_i^q)} \begin{bmatrix} z_i^0 \\ -z_i^1 \end{bmatrix} \right). \tag{23}$$

The appearance of the outer-product terms in the equation above is potentially alarming. If the vectors u_i^q, v_i^q are dense, then even if A_i^q is sparse, the corresponding matrix M_i^q, and hence the Schur complement matrix M, will be dense. A direct factorization of the resulting dense matrix will be very expensive for even moderately high m.

The observed behavior of the density of the matrix M_i^q on test problems depends largely on the particular problem structure. When the problem has many small quadratic blocks, it is often the case that each block appears in only a small fraction of the constraints. In this case, all A_i^q matrices are sparse and the vectors u_i^q and v_i^q turn out to be sparse vectors for each i. Consequently, the matrices M_i^q remain relatively sparse for these problems. As a result, M is also sparse and it can be factorized directly with reasonable cost. The behavior is typical for all `nql` and `qssp` problems from the DIMACS library.

The situation is drastically different for problems where one of the quadratic blocks, say the ith block, is large. For such problems the vectors u_i^q, v_i^q are typically dense, and therefore, M_i^q is likely be a dense matrix even if the data A_i^q is sparse. However, observe that M_i^q is a rank-two perturbation of a sparse matrix when A_i^q is sparse. In such a situation, it may be advantageous to use the Sherman-Morrison-Woodbury update formula [6] when solving the Schur complement equation (9). This is a standard strategy used in linear programming when there are dense columns in the constraint matrix and this is the approach we used in our implementation of SDPT3. This approach helps tremendously on the scheduling problems from the DIMACS library.

To apply the Sherman-Morrison-Woodbury formula, we need to modify the sparse portion of the matrix M_i^q slightly. Since the diagonal matrix J_i has a negative component, the matrix $(A_i^q)^T J_i A_i^q$ need not be a positive definite matrix, and therefore the Cholesky factorization of the sparse portion of M_i^q need not exist. To overcome this problem, we use the following identity:

$$M_i^q \;=\; \frac{\langle x_i^q, z_i^q\rangle}{\gamma^2(z_i^q)}(A_i^q)^T A_i^q \;+\; u_i^q(v_i^q)^T + v_i^q(u_i^q)^T - 2\frac{\langle x_i^q, z_i^q\rangle}{\gamma^2(z_i^q)} k_i k_i^T, \tag{24}$$

where u_i^q and v_i^q are as in (23) and $k_i = (A_i^q)^T e_i^q$. Note that if A_i^q is a large sparse matrix with a few dense rows, we also use the Sherman-Morrison-Woodbury formula to handle the matrix $(A_i^q)^T A_i^q$ in (24).

In the above, we have focused our discussion on the HKM direction, but the same holds true for the NT direction, where the corresponding matrix M_i^q is given by

$$M_i^q \;=\; \frac{1}{\omega_i^2}\left((A_i^q)^T J_i A_i^q \;+\; 2u_i^q(u_i^q)^T\right), \text{ with } u_i^q = (A_i^q)^T \begin{bmatrix} t_i^0 \\ -t_i^1 \end{bmatrix}. \tag{25}$$

Finally, we give a brief description of our implementation of the Sherman-Morrison-Woodbury formula for solving the Schur complement equation when M is a low rank perturbation of a sparse matrix. In such a case, the Schur complement matrix M can be written in the form

$$M \;=\; H + UU^T \tag{26}$$

where H is a sparse symmetric matrix and U has only few columns. If H is nonsingular, then by the Sherman-Morrison-Woodbury formula, the solution of the Schur complement equation is given by

$$\Delta y \;=\; \hat{h} - H^{-1}U\left(I + U^T H^{-1} U\right)^{-1} U^T \hat{h}, \tag{27}$$

where $\hat{h} = H^{-1}h$.

Computing Δy via the Sherman-Morrison-Woodbury update formula above is not always stable, and the computed solution for Δy can be highly inaccurate when H is ill-conditioned. To partially overcome such a difficulty, we combine the Sherman-Morrison-Woodbury update with iterative refinement [8]. It is noted in [8] that iterative refinement is beneficial even if the residuals are computed only at the working precision.

Our numerical experience with the second-order cone problems from the DIMACS library confirmed that iterative refinement very often does improve the accuracy of the computed solution for Δy via the Sherman-Morrison-Woodbury formula. However, we must mention that iterative refinement can occasionally fail to provide any significant improvement. We have not yet incorporated a stable and efficient method for computing Δy when M has the form (26), but note that Goldfarb and Scheinberg [5] discuss a stable product-form Cholesky factorization approach to this problem.

3.3 Step-length computation

Once a direction Δx is computed, a full step will not be allowed if $x + \Delta x$ violates the conic constraints. Thus, the next iterate must take the form $x + \alpha\Delta x$ for an appropriate choice of the step-length α. In this subsection, we discuss an efficient strategy to compute the step-length α.

For semidefinite blocks, it is straightforward to verify that, for the jth block, the maximum allowed step-length that can be taken without violating the positive semidefiniteness of the matrix $x_j^s + \alpha_j^s \Delta x_j^s$ is given as follows:

$$\alpha_j^s = \begin{cases} \dfrac{-1}{\lambda_{\min}((x_j^s)^{-1}\Delta x_j^s)}, & \text{if the minimum eigenvalue } \lambda_{\min} \text{ is negative} \\ \infty & \text{otherwise.} \end{cases} \tag{28}$$

If the computation of eigenvalues necessary in α_j^s above becomes expensive, then we resort to finding an approximation of α_j^s by estimating extreme eigenvalues using Lanczos iterations [17]. This approach is quite accurate in general and represents a good trade-off between the computational effort versus quality of the resulting stepsizes.

For quadratic blocks, the largest step-length α_i^q that keeps the next iterate feasible with respect to the kth quadratic cone can be computed as follows. Let

$$a_i = \gamma^2(\Delta x_i^q), \quad b_i = \langle \Delta x_i^q, -J_i x_i^q \rangle, \quad c_i = \gamma^2(x_i^q), \quad d_i \;=\; b_i^2 - a_i c_i,$$

where J_i is the matrix defined in (23). We want the largest positive $\bar{\alpha}$ for which $a_i\alpha^2 + 2b_i\alpha + c_i > 0$ for all smaller positive α's, which is given by

$$\alpha_i^q = \begin{cases} \dfrac{-b_i - \sqrt{d_i}}{a_i} & \text{if } a_i < 0 \text{ or } b_i < 0, a_i \le b_i^2/c_i \\ \dfrac{-c_i}{2b_i} & \text{if } a_i = 0, b_i < 0 \\ \infty & \text{otherwise.} \end{cases}$$

For the linear block, the maximum allowed step-length α_i^l for the hth component is given by

$$\alpha_h^l = \begin{cases} \dfrac{-x_h^l}{\Delta x_h^l}, & \text{if } \Delta x_h^l < 0 \\ \infty & \text{otherwise.} \end{cases}$$

Finally, an appropriate step-length α that can be taken in order for $x+\alpha\Delta x$ to satisfy all the conic constraints takes the form

$$\alpha \;=\; \min\left(1,\; \gamma \min_{1\le j\le n_s} \alpha_j^s,\; \gamma \min_{1\le i\le n_q} \alpha_i^q,\; \gamma \min_{1\le h\le n_l} \alpha_h^l\right), \tag{29}$$

where γ (known as the step-length parameter) is typically chosen to be a number slightly less than 1, say 0.98, to ensure that the next iterate $x+\alpha\Delta x$ stays strictly in the interior of all the cones.

For the dual direction Δz, we let the analog of α_j^s, α_i^q and α_h^l be β_j^s, β_i^q and β_h^l, respectively. Similar to the primal direction, the step-length that can be taken by the dual direction Δz is given by

$$\beta \;=\; \min\left(1,\; \gamma \min_{1\le j\le n_s} \beta_j^s,\; \gamma \min_{1\le i\le n_q} \beta_i^q,\; \gamma \min_{1\le h\le n_l} \beta_h^l\right). \tag{30}$$

4 Further details

Sample runs.

We will now generate some sample runs to illustrate how our package might be used to solve test problems from the SDPLIB and DIMACS libraries [3, 13]. We provide two m-files, `read_sdpa.m` and `read_sedumi.m`, to convert problem data from these libraries into MATLAB cell arrays described in Section 2. We assume that the current directory is `SDPT3-3.0` and `sdplib` is a subdirectory.

```
>> startup      % set up default parameters in the OPTIONS structure
>> [blk,A,C,b] = read_sdpa('./sdplib/mcp250.1.dat-s');
>> [obj,X,y,Z] = sqlp(blk,A,C,b);

*******************************************************************
         Infeasible path-following algorithms
*******************************************************************
 version  predcorr  gam  expon  scale_data
   HKM      1      0.000   1       0
it  pstep dstep p_infeas d_infeas  gap       obj       cputime
-------------------------------------------------------------------
 0  0.000 0.000 1.8e+02 1.9e+01  7.0e+05 -1.462827e+04
 1  0.981 1.000 3.3e+00 2.0e-15  1.7e+04 -2.429708e+03  0.7
 2  1.000 1.000 4.3e-14 0.0e+00  2.4e+03 -1.352811e+03  2.2
 :    :     :      :       :        :             :      :
13  1.000 0.996 3.9e-13 8.6e-17  2.1e-05 -3.172643e+02  19.2
14  1.000 1.000 4.1e-13 8.9e-17  6.5e-07 -3.172643e+02  20.6

 Stop: max(relative gap, infeasibilities) < 1.00e-08
----------------------------------------------------
 number of iterations   = 14
 gap                    = 6.45e-07
 relative gap           = 2.03e-09
```

```
 primal infeasibilities = 4.13e-13
 dual   infeasibilities = 8.92e-17
 Total CPU time (secs)  = 21.8
 CPU time per iteration = 1.6
 termination code       =  0
-------------------------------------------------------------------
 Percentage of CPU time spent in various parts
-------------------------------------------------------------------
 preproc  Xchol Zchol   pred   pred_steplen   corr   corr_steplen   misc
   5.7    3.6   0.5     33.3   9.5     3.9    25.2   11.1    3.9    3.3
-------------------------------------------------------------------
```

We can solve a DIMACS test problem in a similar manner.

```
>> OPTIONS.vers = 2;  % use NT direction
>> [blk,A,C,b] = read_sedumi('./dimacs/nb.mat');
>> [obj,X,y,Z] = sqlp(blk,A,C,b,[],[],[],OPTIONS);

******************************************************************
        Infeasible path-following algorithms
*******************************************************************
 version  predcorr  gam  expon  scale_data
    NT      1      0.000   1        0
it  pstep dstep p_infeas d_infeas  gap        obj       cputime
-------------------------------------------------------------------
 0  0.000 0.000 1.4e+03 5.8e+02  4.0e+04  0.000000e+00
 1  0.981 0.976 2.6e+01 1.4e+01  7.8e+02 -1.423573e+01  2.8
 2  1.000 0.989 1.2e-14 1.5e-01  2.7e+01 -1.351345e+01  6.4
 :    :     :      :       :        :             :      :
13  0.676 0.778 2.6e-05 1.4e-08  2.4e-04 -5.059624e-02  45.7
14  0.210 0.463 2.6e-04 7.7e-09  1.9e-04 -5.061370e-02  49.3

 Stop: relative gap < 5*infeasibility
----------------------------------------------------
 number of iterations   = 14
 gap                    = 1.89e-04
 relative gap           = 1.89e-04
 primal infeasibilities = 2.57e-04
 dual   infeasibilities = 7.65e-09
 Total CPU time (secs)  = 51.3
 CPU time per iteration = 3.7
 termination code       =  0
-------------------------------------------------------------------
 Percentage of CPU time spent in various parts
-------------------------------------------------------------------
 preproc  Xchol Zchol  pred   pred_steplen   corr   corr_steplen   misc
   4.0    0.2   0.1    90.0   0.2     0.1    2.6    0.1     0.3    2.3
-------------------------------------------------------------------
```

Note that in this example, dual feasibility is almost attained, while primal feasibility was attained at iteration 2 but has since been slowly degrading. The iterations terminate when this degradation overtakes the improvement in the duality gap (this is part of Item 4 in our list of stopping criteria in Section 2).

Mex files used.

Our software uses a number of Mex routines generated from C programs written to carry out certain operations that MATLAB is not efficient at. In particular, operations such as extracting selected elements of a matrix, and performing arithmetic operations on these selected elements are all done in C. As an example, the vectorization operation **svec** is coded in the C program `mexsvec.c`.

Our software also uses a number of Mex routines generated from Fortran programs written by Ng, Peyton, and Liu for computing sparse Cholesky factorizations [12]. These programs are adapted from the LIPSOL software written by Y. Zhang [21].

To generate these Mex routines, the user can run the shell script file `Installmex` in the subdirectory `SDPT3-3.0/Solver/mexsrc` by typing `./Installmex` in that subdirectory.

Cholesky factorization.

Earlier versions of SDPT3 were intended for problems that always have semidefinite cone constraints. For SDP problems, the Schur complement matrix M in (10) is generally a dense matrix after the first iteration. To solve the associated linear system (9), we first find a Cholesky factorization of M and then solve two triangular systems. When M is dense, a reordering of the rows and columns of M does not alter the efficiency of the Cholesky factorization and specialized sparse Cholesky factorization routines are not useful. Therefore, earlier versions of SDPT3 (up to version 1.3) simply used MATLAB's `chol` routine for Cholesky factorizations. For versions 2.1 and 2.2, we introduced our own Cholesky factorization routine `mexchol` that utilizes loop unrolling and provided 2-fold speed-ups on some architectures compared to MATLAB's `chol` routine. However, in newer versions of MATLAB that use numerics libraries based on LAPACK, MATLAB's `chol` routine is more efficient than our Cholesky factorization routine `mexchol` for dense matrices. Thus, in version 3.0, we use MATLAB's `chol` routine whenever M is dense.

For most second-order cone programming problems in the DIMACS library, however, MATLAB's `chol` routine is not competitive. This is largely due to the fact that the Schur complement matrix M is often sparse for SOCPs and LPs, and MATLAB cannot sufficiently take advantage of this sparsity. To solve such problems more efficiently we imported the sparse Cholesky solver in Y. Zhang's LIPSOL [21], an interior-point code for linear programming problems. It should be noted that LIPSOL uses Fortran programs developed by Ng, Peyton, and Liu for Cholesky factorization [12]. When SDPT3 uses LIPSOL's Cholesky solver, it first generates a symbolic factorization of the Schur complement matrix to determine the pivot order by examining the sparsity structure of this matrix carefully. Then, this pivot order is re-used in later iterations to compute the Cholesky factors. Contrary to the case of linear programming, however, the sparsity structure of the Schur complement matrix can change during the iterations for SOCP problems. If this happens, the pivot order has to be recomputed. We detect changes in the sparsity structure by monitoring the nonzero elements of the Schur complement matrix. Since the default initial iterates we use for an SOCP problem are unit vectors but subsequent iterates are not, there is always a change in the sparsity pattern of M after the first iteration. After the second iteration, the sparsity pattern remains unchanged for most problems, and only one more change occurs in a small fraction of the test problems.

The effect of including a sparse Cholesky solver option for SOCP problems was dramatic. We observed speed-ups of up to two orders of magnitude. Version 3.0 of SDPT3 automatically makes a choice between MATLAB's built-in `chol` routine and the sparse Cholesky solver based on the density of the Schur complement matrix. The cutoff density is specified in the parameter `OPTIONS.spdensity`.

Vectorized matrices vs. sparse matrices.

The current release, version 3.0, of the code stores the constraint matrix in "vectorized" form as described in Section 2. In the previous version 2.3, $\tilde{A}$ was a doubly subscripted cell array of symmetric matrices for the semidefinite blocks. The result of the change is that much less storage is required for the constraint matrix, and that we save a considerable amount of time in forming the Schur complement matrix M in (10) by avoiding loops over the index k. An analysis of the amount of time version 3.0 of our code spends on different parts of the algorithm leads to the following observations, which can sometimes be platform dependent: Operations relating to forming and factorizing the Schur complement and hence computing the predictor search direction comprise much of the computational work for most problem classes, ranging from about 25% for `qpG11` up to 99% for the larger `theta` problems, the `control` problems, `copo14`, `hamming-7-5-6`, and the `nb` problems. Other parts of the code that require significant amount of computational time include the computation of the corrector search direction (up to 51% on some `qp` problems) and the computation of step lengths (up to 50% on `truss7`).

While we now store the constraint matrix in vectorized form, the parts of the iterates `X` and `Z` corresponding to semidefinite blocks are still stored as matrices, since that is how the user wants to access them.

Choice of search direction.

The new version of the code allows only two search directions, HKM and NT. Version 2.3 also allowed the AHO direction of Alizadeh, Haeberly, and Overton [1] and the GT (Gu-Toh, see [16]) direction, but these options are not competitive when the problems are of large scale. We intend to keep version 2.3 of the code available for those who wish to experiment with these other search directions, which tend to give more accurate results on smaller problems.

For the two remaining search directions, our computational experience on problems from the SDPLIB and DIMACS libaries is that the HKM direction is almost universally faster than NT on problems with semidefinite blocks, especially for sparse problems with large semidefinite blocks such as the `equalG` and `maxG` problems. The reason that the latter is slower is due to the NT direction's need to compute an eigenvalue decomposition to calculate the NT scaling matrix w_j^s. This computation can dominate the work in each interior-point iteration when the problem is sparse.

The NT direction, however, was faster on SOCP problems such as the `nb`, `nql`, and `sched` problems. The reason for this behavior is not hard to understand. By comparing the formula in (22) for the HKM direction with (25) for the NT direction, it is clear that more computation is required to assemble the Schur complement matrix and more low-rank updating is necessary for the former direction.

Real vs. complex data.

In earlier versions, we allowed SDP problems with complex data, i.e., the constraint matrices are hermitian matrices. However, as problems with complex data rarely occur in practice, and in an effort to simplify the code, we removed this flexibility in the current version. But we intend to keep version 2.3 of the code available for users who wish to solve SDP problems with complex data.

Homogeneous vs. infeasible interior-point methods.

Version 2.3 also allowed the user to employ homogeneous self-dual algorithms instead of the usual infeasible interior-point methods. However, this option almost always took longer than the default choice, and so it has been omitted from the current release. One theoretical advantage of the homogeneous self-dual approach is that it is oriented towards either producing optimal primal and dual solutions or generating a certificate of primal or dual infeasibility, while the infeasible methods strive for optimal solutions only, but detect infeasibility if either the dual or primal iterates diverge. However, we have observed no advantage to the homogeneous methods when applied to infeasible problems. We should mention, however, that our current version does not detect infeasibility in the problem `filtinf1`, but instead stops with a primal near-feasible solution and a dual feasible solution when it encounters numerical problems.

Specifying the block structure of problems.

Our software requires the user to specify the block structure of the SQLP problem. Although no technical difficulty will arise if the user choose to lump a few blocks together and consider it as a single large block, the computational time can be dramatically different. For example, the problem `qpG11` in the SDPLIB library actually has block structure `blk{1,1} = 's', blk{1,2} = 800, blk{2,1} = 'l', blk{2,2}=800`, but the structure specified in the library is `blk{1,1} = 's', blk{1,2} = 1600`. That is, in the former, the linear variables are explicitly identified, rather than being part of a large sparse semidefinite block. The difference in the running time for specifying the block structure differently is dramatic: the former representation is at least six times faster when the HKM direction is used, besides using much less memory space.

It is thus crucial to present problems to the algorithms correctly. We could add our own preprocessor to detect this structure, but believe users are aware of linear variables present in their problems. Unfortunately the versions of `qpG11` (and also `qpG51`) in SDPLIB do not show this structure explicitly. In our software, we provided an m-file, `detect_diag.m`, to detect and correct problems with hidden linear variables. The user can call this m-file after loading the problem data into MATLAB as follows:

```
>> [blk,A,C,b] = read_sdpa('../SDPtest/sdplib/qpG11.dat-s');
>> [blk,A,C,b] = detect_diag(blk,A,C,b);
```

Finally, version 2.3 of SDPT3 included specialized routines to compute the Schur complement matrices directly for certain classes of problem (e.g., maxcut problems). In earlier versions of SDPT3, these specialized routines had produced dramatic decreases in solution times, but for version 2.3, these gains were marginal, since our C Mex routines for exploiting sparsity in computing the Schur complement matrix provided almost as much speedup. We have therefore dropped these routines in version 3.0.

Conversion of problems into standard form.

Here we shall just give an example of how an SDP with linear inequality constraints can be converted into the standard form given in the Introduction. Suppose we have an SDP of the following form:

$$\begin{array}{lll} (P_1) & \min & \langle c^s, x^s \rangle \\ & \text{s.t.} & (A^s)^T \mathbf{svec}(x^s) \;\leq\; b, \\ & & x^s \in K_s^n. \end{array}$$

That is, it has inequality constraint instead of equality constraints. But by introducing a slack variable x^l, we can easily convert (P_1) into the standard form, namely,

$$\begin{array}{llcccl} (P_1^*) & \min & \langle c^s, x^s \rangle & + & \langle c^l, x^l \rangle & \\ & \text{s.t.} & (A^s)^T \mathbf{svec}(x^s) & + & (A^l)^T x^l & = \; b, \\ & & x^s \in K_s^n. & & x^l \in K_l^m, & \end{array}$$

where $c^l = 0$, and $A^l = [e_1 \;\; \cdots \;\; e_m]$. With our use of cell arrays to representation SQLP data, it is easy to take the problem data of (P_1) and use them for the standard form (P_1^*) as follows:

`blk{1,1} = 's'`	`blk{1,2}` $= n$
`A{1}` $= A^s$	`C{1}` $= c^s$
`blk{2,1} = 'l'`	`blk{2,2}` $= m$
`A{2}` $= A^l$	`C{2}` $= c^l$.

Caveats.

The user should be aware that SQLP is more complicated than linear programming. For example, it is possible that both primal and dual problems are feasible, but their optimal values are not equal. Also, either problem may be infeasible without there being a certificate of that fact (so-called weak infeasibility). In such cases, our software package is likely to terminate after some iterations with an indication of short step-length or lack of progress. Also, even if there is a certificate of infeasibility, our infeasible-interior-point methods may not find it. In our very limited testing on strongly infeasible problems, most of our algorithms have been quite successful in detecting infeasibility.

5 Computational results

Here we describe the results of our computational testing of SDPT3, on problems from the SDPLIB collection of Borchers [3] as well as the DIMACS library test problems [13]. In both, we solve a selection of the problems; in the DIMACS problems, these are selected as the more tractable problems, while our subset of the SDPLIB problems is more representative (but we cannot solve the largest two `maxG` problems). Since our algorithm is a primal-dual method storing the primal iterate `X`, it cannot exploit common sparsity in `C` and the constraint matrices as effectively as dual methods or nonlinear-programming based methods. We are therefore unable to solve the largest problems.

The test problems are listed in Tables 1 and 2, along with their dimensions. The results given were obtained on a Pentium III PC (800MHz) with 1G of memory running Linux, using MATLAB 6.0. (We had some difficulties with MATLAB 6 using some of our codes on a Solaris platform, possibly due to bugs in the Solaris version of MATLAB 6.)

Results are given in Tables 3 and 4: Table 3 summarizes the results of our computational experiments on the DIMACS set of problems, while the corresponding results for problems from the SDPLIB library are presented in Table 4. In each table, we list the number of of iterations required, the time in seconds, and four measures of the precision of the computed answer for each problem and for both the HKM and the NT directions. The first of the four accuracy measures is the logarithm (to base 10) of the total complementary slackness; the second is the scaled primal infeasibility $\|Ax - b\|/(1 + \max|b_k|)$, and the third is $\|A^T y + z - c\|/(1 + \max|c|)$, where the norm is subordinate to the inner product and the maximum taken over all components of c; and the last one is the maximum of 0 and $\langle c, x\rangle - b^T y$. Entries like $3 - 13$ mean 3×10^{-13}, etc. In accuracy reporting we followed the guidelines set up for the DIMACS Challenge that took place in November 2000. These set of measures are somewhat inconsistent: the first and the last are *absolute* measures that do not take the solution size into account while the other two measures are *relative* to the sizes of certain input parameters.

Our codes solved most of the problems in the two libraries to reasonable accuracy

Problem	m	semidefinite blocks	second-order blocks	linear block
bm1	883	882	–	–
copo14	1275	[14 x 14]	–	364
copo23	5820	[23 x 23]	–	1771
copo68	154905	[68 x 68]	–	50116
filter48-socp	969	48	49	931
filtinf1	983	49	49	945
minphase	48	48	–	–
hamming-7-5-6	1793	128	–	–
hamming-9-8	2305	512	–	–
hinf12	43	[3, 6, 6, 12]	–	–
hinf13	57	[3, 7, 9, 14]	–	–
nb	123	–	[793 x 3]	4
nb-L1	915	–	[793 x 3]	797
nb-L2	123	–	[1677, 838 x 3]	4
nb-L2-bessel	123	–	[123, 838 x 3]	4
nql30	3680	–	[900 x 3]	3602
nql60	14560	–	[3600 x 3]	14402
nql180	130080	–	[32400 x 3]	129602
nql30old	3601	–	[900 x 3]	5560
nql60old	14401	–	[3600 x 3]	21920
nql180old	129601	–	[32400 x 3]	195360
qssp30	3691	–	[1891 x 4]	2
qssp60	14581	–	[7381 x 4]	2
qssp180	130141	–	[65341 x 4]	2
qssp30old	5674	–	[1891 x 4]	3600
qssp60old	22144	–	[7381 x 4]	14400
qssp180old	196024	–	[65341 x 4]	129600
sched-50-50-orig	2527	–	[2474, 3]	2502
sched-50-50-scaled	2526	–	2475	2502
sched-100-50-orig	4844	–	[4741, 3]	5002
sched-100-50-scaled	4843	–	4742	5002
sched-100-100-orig	8338	–	[8235, 3]	10002
sched-100-100-scaled	8337	–	8236	10002
sched-200-100-orig	18087	–	[17884, 3]	20002
sched-200-100-scaled	18086	–	17885	20002
torusg3-8	512	512	–	–
toruspm3-8-50	512	512	–	–
truss5	208	[33 x 10, 1]	–	–
truss8	496	[33 x 19, 1]	–	–

Table 1: Selected DIMACS library problems. Notation like [33 x 19] indicates that there were 33 semidefinite blocks, each a symmetric matrix of order 19, etc.

Problem	m	semidefinite blocks	linear block
arch8	174	161	174
control7	666	[70, 35]	–
control10	1326	[100, 50]	–
control11	1596	[110, 55]	–
gpp250-4	251	250	–
gpp500-4	501	500	–
hinf15	91	37	–
mcp250-1	250	250	–
mcp500-1	500	500	–
qap9	748	82	–
qap10	1021	101	–
ss30	132	294	132
theta3	1106	150	–
theta4	1949	200	–
theta5	3028	250	–
theta6	4375	300	–
truss7	86	[150 x 2, 1]	–
truss8	496	[33 x 19, 1]	–
equalG11	801	801	–
equalG51	1001	1001	–
equalG32	2001	2001	–
maxG11	800	800	–
maxG51	1000	1000	–
maxG32	2000	2000	–
qpG11	800	1600	–
qpG112	800	800	800
qpG51	1000	2000	–
qpG512	1000	1000	1000
thetaG11	2401	801	–
thetaG11n	1601	800	–
thetaG51	6910	1001	–
thetaG51n	5910	1000	–

Table 2: Selected SDPLIB Problems. Note that `qpG112` is identical to `qpG11` except that the structure of the semidefinite block is exposed as a sparse symmetric matrix of order 800 and a diagonal block of the same order, which can be viewed as a linear block, and similarly for `qpG512`. Also, `thetaG11n` is a more compact formulation of `thetaG11`, and similarly for `thetaG51n`.

— we discuss some of the exceptions. On the DIMACS set of problems, our algorithms terminated with low accuracy solutions (measured by $\langle x, z\rangle$) on the scheduling problems and old versions of the `nql` and `qssp` problems as well as `torusg3-8` and `filtinf1`. The last of these problems, `filtinf1`, is an infeasible problem, but we run into numerical problems before detecting its infeasibility. The optimal values for the `sched*.orig` problems and for `torusg3-8` are above 10^5, so the relative accuracy, which may be considered a better measure of accuracy, is acceptable. Both the old and new versions of the `nql` and `qssp` problems contain duplicated columns coming from splitting free variables. Feasible sets of the duals of these problems have empty interiors and this fact affects the performance of our codes — apparently more so on the older formulations of these problems. Other measures of accuracy were also reasonable for most DIMACS problems, except for, once again, the scheduling problems and some of the `nql` and `qssp` problems.

For the SDPLIB set of problems, we consistently achieve high accuracy solutions, for both the HKM and the NT directions. A few of the smaller problems (`hinf15`, `truss7`, and `truss8`) turn out to be more difficult to solve accurately using either search direction. Interested readers can find detailed discussion of these computational experiments as well as qualitative and quantitative comparisons of different versions of the code and different search directions in a related article by the authors [19].

References

[1] F. Alizadeh, J.-P. A. Haeberly, and M. L. Overton, *Primal-dual interior-point methods for semidefinite programming: convergence results, stability and numerical results*, SIAM J. Optimization, 8 (1998), pp. 746–768.

[2] F. Alizadeh, J.-P. A. Haeberly, M. V. Nayakkankuppam, M. L. Overton, and S. Schmieta, *SDPPACK user's guide*, Technical Report, Computer Science Department, NYU, New York, June 1997.

[3] B. Borchers, *SDPLIB 1.2, a library of semidefinite programming test problems*, Optimization Methods and Software, 11 & 12 (1999), pp. 683–690. Available at `http://www.nmt.edu/~borchers/sdplib.html`.

[4] K. Fujisawa, M. Kojima, and K. Nakata, *Exploiting sparsity in primal-dual interior-point method for semidefinite programming*, Mathematical Programming, 79 (1997), pp. 235–253.

[5] D. Goldfarb and K. Scheinberg, *A product-form Cholesky factorization implementation of an interior-point method for second order cone programming*, preprint.

[6] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 2nd ed., Johns Hopkins University Press, Baltimore, MD, 1989.

	HKM						NT					
Problem	Itn	log $\langle x, z\rangle$	err1	err3	err5	time	Itn	log $\langle x, z\rangle$	err1	err3	err5	time
bm1	18	-6	5-7	3-13	4-6	816	16	-3	4-7	3-13	2-3	2786
copo14	15	-10	1-10	6-15	0	38	13	-9	6-11	6-15	9-9	34
copo23	17	-10	2-9	1-14	8-8	2041	16	-9	8-10	1-14	2-8	1976
filter48-socp	38	-6	1-6	1-13	4-5	54	45	-6	1-6	6-14	5-5	60
filtinf1	27	-1	3-5	1-11	3-1	38	29	-1	1-5	3-11	4-1	44
minphase	32	-7	8-9	3-12	0	5	37	-5	2-8	7-13	0	7
hamming-7-5-6	10	-9	2-15	9-15	7-9	66	10	-9	2-15	9-15	7-9	67
hamming-9-8	11	-6	5-15	9-14	2-6	212	11	-6	5-15	8-14	2-6	422
hinf12	42	-8	2-8	4-10	0	5	39	-8	2-8	2-10	0	5
hinf13	23	-3	9-5	8-13	0	4	22	-4	1-4	9-13	0	4
nb	15	-4	1-5	2-9	2-4	42	14	-4	1-5	1-8	2-4	31
nb-L1	16	-4	7-5	4-9	4-4	75	16	-5	2-4	9-11	1-4	60
nb-L2	12	-8	2-9	1-11	3-8	58	11	-6	4-9	1-8	2-6	45
nb-L2-b	13	-8	8-6	4-12	1-6	40	11	-7	3-7	2-9	7-7	27
nql30	13	-4	6-8	5-9	5-5	11	16	-6	2-6	3-11	0	12
nql60	13	-4	4-7	1-8	1-4	64	15	-5	3-6	2-10	0	57
nql180	15	-3	1-5	3-8	2-4	5686	16	-4	7-5	4-10	0	3264
nql30o	12	-4	5-5	2-8	0	14	12	-4	5-5	2-8	0	12
nql60o	13	-4	1-4	9-9	0	88	13	-3	9-5	5-8	0	76
nql180o	11	-2	2-3	3-6	0	4932	8	0	4-4	1-4	6-1	2221
qssp30	21	-5	7-8	1-9	1-5	24	18	-7	3-7	2-11	0	17
qssp60	21	-4	5-5	2-9	9-4	153	20	-6	3-6	1-11	0	107
qssp180	24	-3	3-4	1-8	1-2	17886	25	-7	3-5	4-12	1-3	9873
qssp30o	11	-1	2-4	4-5	7-1	60	12	-1	4-4	1-5	1-1	62
qssp60o	11	0	3-4	2-4	2-0	390	11	0	2-4	4-4	2-0	359
sched-50-50-orig	28	-1	7-4	3-9	0	21	29	-1	2-4	3-7	0	20
sched-50-50-scaled	23	-4	1-4	4-15	3-4	18	22	-4	6-5	4-15	2-5	16
sched-100-50-orig	39	-1	6-3	3-11	0	63	33	-1	6-3	2-11	3+2	50
sched-100-50-scaled	26	-2	8-4	8-13	2-2	45	22	-2	7-4	1-9	3-2	36
sched-100-100-orig	33	-1	5-2	9-11	0	103	50	+1	1-0	2-8	3+7	141
sched-100-100-scaled	19	-1	4-2	1-14	0	66	27	-1	3-2	2-14	0	56
sched-200-100-orig	41	-1	6-3	3-9	3-2	350	39	0	6-3	1-8	0	313
sched-200-100-scaled	27	-2	3-3	6-9	0	250	25	-2	3-3	7-10	0	218
torusg3-8	15	-2	2-11	8-16	3-2	90	14	-1	2-10	7-16	3-1	405
toruspm3-8-50	14	-6	2-11	6-16	2-6	85	15	-7	4-11	6-16	7-7	429
truss5	16	-5	4-7	7-15	0	9	16	-6	4-7	8-15	0	10
truss8	15	-5	3-6	8-15	0	44	14	-4	2-6	7-15	0	47

Table 3: Computational results on DIMACS library problems using SDPT3-3.0. These were performed on a Pentium III PC (800MHz) with 1G of memory.

	HKM						NT					
Problem	Itn	log $\langle x, z \rangle$	err1	err3	err5	time	Itn	log $\langle x, z \rangle$	err1	err3	err5	time
arch8	21	-8	1-9	5-13	1-8	42	24	-6	2-8	5-13	2-6	55
control7	22	-5	5-7	2-9	3-5	112	22	-5	7-7	2-9	0	131
control10	24	-5	1-6	6-9	0	505	24	-5	1-6	6-9	0	610
control11	24	-5	2-6	6-9	0	768	23	-4	9-7	6-9	0	890
gpp250-4	15	-6	7-8	6-14	0	25	15	-5	6-8	7-14	0	64
gpp500-4	15	-5	6-8	4-14	0	156	17	-5	1-8	5-14	1-5	579
hinf15	23	-4	9-5	2-12	0	6	22	-4	1-4	2-12	0	7
mcp250-1	14	-7	3-12	4-16	6-7	12	15	-7	1-11	4-16	2-7	42
mcp500-1	15	-7	1-11	5-16	7-7	62	16	-7	3-11	5-16	3-7	327
qap9	15	-5	4-8	5-13	0	17	15	-5	5-8	6-13	0	18
qap10	14	-5	4-8	3-13	0	30	13	-4	4-8	5-13	0	30
ss30	21	-7	8-9	3-13	5-7	139	24	-6	1-8	2-13	4-6	245
theta3	15	-7	2-10	2-14	1-7	38	14	-8	2-10	2-14	3-8	40
theta4	15	-7	2-10	3-14	2-7	130	14	-8	3-10	3-14	6-8	135
theta5	15	-7	3-10	4-14	2-7	396	14	-8	4-10	4-14	4-8	402
theta6	14	-7	2-10	5-14	3-7	975	14	-7	6-10	5-14	1-7	1034
truss7	23	-4	3-6	1-13	0	4	21	-4	2-6	2-13	0	5
truss8	15	-5	3-6	7-15	0	45	14	-4	2-6	1-14	1-4	47
equalG11	17	-6	3-10	3-16	2-6	606	18	-5	7-11	7-15	2-5	2371
equalG51	20	-6	2-8	5-16	4-6	1358	20	-6	2-9	1-15	4-6	5116
equalG32	19	-6	2-10	1-14	6-6	8839	19	-6	2-10	4-15	1-6	37419
maxG11	15	-6	9-12	7-16	6-6	192	15	-6	4-11	7-16	1-6	1360
maxG51	17	-6	3-12	5-16	4-6	617	16	-5	2-10	3-16	1-5	3071
maxG32	16	-5	1-10	1-15	1-5	2441	16	-6	2-10	1-15	2-6	21999
qpG11	16	-7	2-11	0	9-7	1498	15	-5	1-10	0	2-5	4487
qpG112	18	-6	2-11	0	1-6	222	17	-5	5-11	0	2-5	1529
qpG51	17	-5	2-10	0	6-5	3157	25	-5	8-10	0	9-5	16548
qpG512	19	-5	8-10	0	1-5	635	29	-5	6-10	0	8-5	5688
thetaG11	19	-6	4-9	4-14	2-6	817	20	-7	2-9	5-14	8-8	2311
thetaG11n	15	-7	1-12	2-13	4-7	460	15	-7	1-12	2-13	4-7	1581
thetaG51	38	-7	1-8	3-13	7-7	17582	30	-5	2-8	1-12	2-5	18659
thetaG51n	19	-6	2-9	5-13	0	3908	23	-7	3-9	5-13	0	8479

Table 4: Computational results on SDPLIB problems using SDPT3-3.0. These were performed on a Pentium III PC (800MHz) with 1G of memory.

[7] C. Helmberg, F. Rendl, R. Vanderbei and H. Wolkowicz, *An interior-point method for semidefinite programming*, SIAM Journal on Optimization, 6 (1996), pp. 342–361.

[8] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996.

[9] M. Kojima, S. Shindoh, and S. Hara, *Interior-point methods for the monotone linear complementarity problem in symmetric matrices*, SIAM J. Optimization, 7 (1997), pp. 86–125.

[10] S. Mehrotra, *On the implementation of a primal-dual interior point method*, SIAM J. Optimization, 2 (1992), pp. 575–601.

[11] R. D. C. Monteiro, *Primal-dual path-following algorithms for semidefinite programming*, SIAM J. Optimization, 7 (1997), pp. 663–678.

[12] J. W. Liu, E. G. Ng, and B. W. Peyton, *On finding supernodes for sparse matrix computations*, SIAM J. Matrix Anal. Appl., 1 (1993), pp. 242–252.

[13] G. Pataki and S. Schmieta, *The DIMACS library of mixed semidefinite-quadratic-linear programs.*
Available at `http://dimacs.rutgers.edu/Challenges/Seventh/Instances`

[14] Yu. E. Nesterov and M. J. Todd, *Self-scaled barriers and interior-point methods in convex programming*, Math. Oper. Res., 22 (1997), pp. 1–42.

[15] M. J. Todd, K. C. Toh, and R. H. Tütüncü, *On the Nesterov-Todd direction in semidefinite programming*, SIAM J. Optimization, 8 (1998), pp. 769–796.

[16] K. C. Toh, *Some new search directions for primal-dual interior point methods in semidefinite programming*, SIAM J. Optimization, 11 (2000), pp. 223–242.

[17] K. C. Toh, *A note on the calculation of step-lengths in interior-point methods for semidefinite programming*, submitted.

[18] K. C. Toh, M. J. Todd, R. H. Tütüncü, *SDPT3 — a Matlab software package for semidefinite programming*, Optimization Methods and Software, 11/12 (1999), pp. 545-581.

[19] R. H. Tütüncü, K. C. Toh, M. J. Todd, *Solving semidefinite-quadratic-linear programs using SDPT3,* March 2001. Submitted to Mathematical Programming.

[20] T. Tsuchiya, *A convergence analysis of the scaling-invariant primal-dual path-following algorithms for second-order cone programming*, Optimization Methods and Software, 11/12 (1999), pp. 141–182.

[21] Y. Zhang, *Solving large-scale linear programs by interior-point methods under the* MATLAB *environment*, Optimization Methods and Software, 10 (1998), pp. 1–31.