Math. Program., Ser. B 95: 189–217 (2003)

Digital Object Identifier (DOI) 10.1007/s10107-002-0347-5

R. H. Tütüncü · K. C. Toh · M. J. Todd

Solving semidefinite-quadratic-linear programs using SDPT3

Received: March 19, 2001 / Accepted: January 18, 2002
Published online: October 9, 2002 –

Abstract. This paper discusses computational experiments with linear optimization problems involving semidefinite, quadratic, and linear cone constraints (SQLPs). Many test problems of this type are solved using a new release of SDPT3, a MATLAB implementation of infeasible primal-dual path-following algorithms. The software developed by the authors uses Mehrotra-type predictor-corrector variants of interior-point methods and two types of search directions: the HKM and NT directions. A discussion of implementation details is provided and computational results on problems from the SDPLIB and DIMACS Challenge collections are reported.

1. Introduction

Conic linear optimization problems can be expressed in the following standard form:

$$\begin{aligned} &\min \ \langle c,\ x\rangle \\ &\text{s.t. } \langle a_k,\ x\rangle = b_k, \quad k = 1, \ldots, m, \\ &\qquad x \in K \end{aligned} \tag{1}$$

where K is a closed, convex pointed cone in a finite dimensional inner product space endowed with an inner product $\langle \cdot,\ \cdot \rangle$. By choosing K to be the semidefinite, quadratic (second-order), and linear cones respectively, one obtains the well-known special cases of semidefinite, second-order cone, and linear programming problems. Recent years have seen a dramatic increase in the number of subclasses of conic optimization problems that can be solved efficiently by interior-point methods. In addition to the ongoing theoretical work that derived convergence guarantees and convergence rates for such algorithms, many groups of researchers have also implemented these algorithms and developed public domain software packages that are capable of solving conic optimization problems of ever increasing size and diversity. This paper discusses the authors' contribution to this effort through the development of the software SDPT3. Our earlier work on SDPT3 is presented in [22, 25].

R. H. Tütüncü: Department of Mathematical Sciences, Carnegie Mellon University, Pittsburgh, PA 15213, USA. e-mail: reha@andrew.cmu.edu. Research supported in part by NSF through grant CCR-9875559.

K. C. Toh: Department of Mathematics, National University of Singapore, 10 Kent Ridge Crescent, Singapore 119260. e-mail: mattohkc@math.nus.edu.sg. Research supported in part by the Singapore-MIT Alliance.

M. J. Todd: School of Operations Research and Industrial Engineering, Cornell University, Ithaca, New York 14853, USA. e-mail: miketodd@cs.cornell.edu. Research supported in part by NSF through grant DMS-9805602 and ONR through grant N00014-96-1-0050.

Mathematics Subject Classification (2000): 90C05, 90C22

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. In other words, we allow K in (1) to be a Cartesian product of cones of positive semidefinite matrices, second-order cones, and the nonnegative orthant. We use the following standard form of such problems, henceforth called SQLP problems:

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

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 $\mathbb{R}^{q_i}$ and $K_q^{q_i}$ is the second-order cone defined by $K_q^{q_i} := \{x \in \mathbb{R}^{q_i} : x_1 \geq \|x_{2:q_i}\|\}$. Finally, c^l, x^l are vectors of dimension n_l and $K_l^{n_l}$ is the cone $\mathbb{R}_+^{n_l}$. 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. (Here, for a symmetric matrix x of order s, $\mathbf{svec}(x) := (x_{11}, \sqrt{2}x_{12}, x_{22}, \sqrt{2}x_{13}, \sqrt{2}x_{23}, x_{33}, \ldots)^T \in \mathbb{R}^{s(s+1)/2}$, where the $\sqrt{2}$ is to make the operation an isometry.) 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:

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

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`

This software package was originally developed to provide researchers in semidefinite programming with a collection of reasonably efficient and robust algorithms that can 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 [25], 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 computational details. Section 3 describes the initial iterates generated by our software while Section 4 briefly describes its options, some implementation details, and its data storage scheme. In Section 5, we present and comment on the results of our computational experiments with our software on problems from the SDPLIB and DIMACS libraries. Section 6 contains a short conclusion.

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 [14] 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 our implementation allows the user to switch to a primal-dual path-following algorithm that does not use corrector steps and sets a centering parameter to be used in such a framework. The choices we make on the parameters used by the algorithm are based on minimizing either the number of iterations or the CPU time of the linear algebra involved in computing the Schur complement matrix and its Cholesky factorization. What follows is a pseudo-code for the algorithm we implemented. Note that this description makes references to later parts of this section where many 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*

$$\texttt{relgap} = \frac{\langle x, z\rangle}{1 + \max(|\langle c, x\rangle|, |b^T y|)}, \quad \phi = \max\left(\frac{\|r_p\|}{1+\|b\|}, \frac{\|R_d\|}{1+\|c\|}\right). \quad (2)$$

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

[1] This strategy works well on most 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 (10), with $\sigma = 0$ *in the right-side vector (12). Denote the solution of (4) by* $(\delta x, \delta y, \delta z)$. *Let* α_p *and* β_p *be the step-lengths defined as in (33) and (34) with* Δx, Δz *replaced by* δx, δz, *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 (10) with R_c *in the the right-hand side vector (12) replaced by*

$$\tilde{R}_c^s = \mathbf{svec}\left[\sigma\mu I - H_P(\mathbf{smat}(x^s)\mathbf{smat}(z^s)) - H_P(\mathbf{smat}(\delta x^s)\mathbf{smat}(\delta z^s))\right]$$

$$\tilde{R}_c^q = \sigma\mu e^q - T_G(x^q, z^q) - T_G(\delta x^q, \delta z^q)$$

$$\tilde{R}_c^l = \sigma\mu e^l - diag(x^l)z^l - diag(\delta x^l)\delta z^l.$$

  Denote the solution of (4) 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 (33) and (34) with* γ *chosen to be* $\gamma = 0.9 + 0.09\min(\alpha_p, \beta_p)$.
- *Update the step-length parameter by*

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

2.1. 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

$$x^s = \begin{bmatrix} \mathbf{svec}(x_1^s) \\ \vdots \\ \mathbf{svec}(x_{n_s}^s) \end{bmatrix}, \quad x^q = \begin{bmatrix} x_1^q \\ \vdots \\ x_{n_q}^q \end{bmatrix}. \tag{3}$$

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

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

and

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

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

$$Ax = b, \quad A^T y + z = c.$$

In this paper, we assume that A has full row rank. However, the preprocess option, when it is turned on, will correctly detect and remove dependent constraints.

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 a block scaling matrix G for the quadratic block. The matrices P and G are usually chosen as a function 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:

$$\begin{aligned} A^T \Delta y + \Delta z &= R_d := c - z - A^T y \\ A\Delta x &= r_p := b - Ax \\ \mathcal{E}^s \Delta x^s + \mathcal{F}^s \Delta z^s &= R_c^s := \mathbf{svec}\,(\sigma\mu I - H_P\,(\mathbf{smat}(x^s)\mathbf{smat}(z^s))) \\ \mathcal{E}^q \Delta x^q + \mathcal{F}^q \Delta z^q &= R_c^q := \sigma\mu e^q - T_G(x^q, z^q) \\ \mathcal{E}^l \Delta x^l + \mathcal{F}^l \Delta z^l &= R_c^l := \sigma\mu e^l - \mathcal{E}^l \mathcal{F}^l e^l, \end{aligned} \tag{4}$$

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

$$\begin{aligned} H_{P_j} &: \mathbb{R}^{s_j \times s_j} \longrightarrow \mathbb{R}^{s_j \times s_j} \\ H_{P_j}(U) &= \tfrac{1}{2}\left[P_j U P_j^{-1} + P_j^{-T} U^T P_j^T\right], \end{aligned} \tag{5}$$

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

$$\mathcal{E}_j^s \;=\; P_j \circledast P_j^{-T} z_j^s, \qquad \mathcal{F}_j^s \;=\; P_j x_j^s \circledast P_j^{-T}, \tag{6}$$

where $R \circledast T$ is the symmetrized Kronecker product operation described in [22].

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_j^q is the first unit vector in $\mathbb{R}^{q_j}$. Let the arrow operator defined in [3] be denoted by $\mathbf{Arw}(\cdot)$. Thus $\mathbf{Arw}(x)$ is a block diagonal matrix whose ith block is

$$\mathbf{Arw}\left(x^i\right) := \mathbf{Arw}\begin{pmatrix} x_0^i \\ x_1^i \end{pmatrix} := \begin{pmatrix} x_0^i & x_1^{iT} \\ x_1^i & x_0^i I_{n_i} \end{pmatrix}.$$

Then the operator $T_G(x^q, z^q)$ is defined as follows:

$$T_G(x^q, z^q) = \begin{bmatrix} \mathbf{Arw}\left(G_1 x_1^q\right)\left(G_1^{-1} z_1^q\right) \\ \vdots \\ \mathbf{Arw}\left(G_{n_q} x_{n_q}^q\right)\left(G_{n_q}^{-1} z_{n_q}^q\right) \end{bmatrix}, \tag{7}$$

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

$$\mathcal{E}_i^q \;=\; \mathbf{Arw}\left(G_i^{-1} z_i^q\right) G_i, \quad \mathcal{F}_i^q \;=\; \mathbf{Arw}\left(G_i x_i^q\right) G_i^{-1}. \tag{8}$$

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

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

$$R_d = \begin{bmatrix} R_d^s \\ R_d^q \\ R_d^l \end{bmatrix}, \; \Delta x = \begin{bmatrix} \Delta x^s \\ \Delta x^q \\ \Delta x^l \end{bmatrix}, \; \Delta z = \begin{bmatrix} \Delta z^s \\ \Delta z^q \\ \Delta z^l \end{bmatrix}. \tag{9}$$

Assuming that $m = \mathcal{O}(n)$, we compute the search direction via a Schur complement equation as follows (the reader is referred to [2] and [22] for details). First compute Δy from the Schur complement equation

$$M\Delta y = h, \tag{10}$$

where

$$M = (A^s)^T(\mathcal{E}^s)^{-1}\mathcal{F}^s A^s + (A^q)^T(\mathcal{E}^q)^{-1}\mathcal{F}^q A^q + (A^l)^T(\mathcal{E}^l)^{-1}\mathcal{F}^l A^l \tag{11}$$

$$\begin{aligned} h = r_p \;&-\; (A^s)^T(\mathcal{E}^s)^{-1}(R_c^s - \mathcal{F}^s R_d^s) \\ &- (A^q)^T(\mathcal{E}^q)^{-1}(R_c^q - \mathcal{F}^q R_d^q) \;-\; (A^l)^T(\mathcal{E}^l)^{-1}(R_c^l - \mathcal{F}^l R_d^l). \end{aligned} \tag{12}$$

Then compute Δx and Δz from the equations

$$\Delta z = R_d - A^T \Delta y \tag{13}$$

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

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

$$\Delta x^l = (\mathcal{E}^l)^{-1} R_c^l - (\mathcal{E}^l)^{-1} \mathcal{F}^l \Delta z^l. \tag{16}$$

2.2. *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.,

$$\begin{bmatrix} x_i^0 \\ x_i^1 \end{bmatrix} = \begin{bmatrix} (x_i^q)_1 \\ (x_i^q)_{2:q_i} \end{bmatrix}. \tag{17}$$

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 $\mathbb{R}_+$:

$$\gamma(x_i^q) := \sqrt{(x_i^0)^2 - \langle x_i^1, x_i^1 \rangle}. \tag{18}$$

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 [10, 12, 16], and the NT direction [17]. See also Tsuchiya [26, 27] 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:

$$G_i = \begin{bmatrix} z_i^0 & (z_i^1)^T \\ z_i^1 & \gamma(z_i^q) I + \dfrac{z_i^1 (z_i^1)^T}{\gamma(z_i^q) + z_i^0} \end{bmatrix}. \tag{19}$$

(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 [22], and G is a symmetric block diagonal matrix whose ith block G_i is defined as follows. Let

$$\omega_i = \sqrt{\frac{\gamma(z_i^q)}{\gamma(x_i^q)}}, \quad \xi_i = \begin{bmatrix} \xi_i^0 \\ \xi_i^1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\omega_i} z_i^0 + \omega_i x_i^0 \\ \frac{1}{\omega_i} z_i^1 - \omega_i x_i^1 \end{bmatrix}. \tag{20}$$

Then

$$G_i = \omega_i \begin{bmatrix} t_i^0 & (t_i^1)^T \\ t_i^1 & I + \dfrac{t_i^1 (t_i^1)^T}{1 + t_i^0} \end{bmatrix}, \quad \text{where} \quad \begin{bmatrix} t_i^0 \\ t_i^1 \end{bmatrix} = \frac{1}{\gamma(\xi_i)} \begin{bmatrix} \xi_i^0 \\ \xi_i^1 \end{bmatrix}. \tag{21}$$

2.3. Computation of the search directions

The size and the density of the Schur complement matrix M defined in (10) is the main determinant of the cost of each iteration in our algorithm. 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 (10).

2.3.1. Semidefinite blocks For problems with semidefinite blocks, it appears that there is not much one can do about additional fill-in, since $(\mathcal{E}^s)^{-1}\mathcal{F}^s$ is dense and structure-less for most problems. One can take advantage of sparsity in A^s in related computations, however, and we discussed some of these issues, such as blockwise computations, in our earlier papers [22, 25].

The way we exploit sparsity of A_j^s in the computation of $M_j^s := (A_j^s)^T (\mathcal{E}_j^s)^{-1}\mathcal{F}_j^s A_j^s$ basically follows the approach in [7]. We will not go into the details here but just briefly highlight one issue that is often critical in cutting down the computation time in forming M_j^s. Let $A_j^s(:,k)$ be the kth column of A_j^s. In computing the kth column of M_j^s, typically a matrix product of the form $x_j^s\, \mathbf{smat}(A_j^s(:,k))\, (z_j^s)^{-1}$ or $w_j^s\, \mathbf{smat}(A_j^s(:,k))\, w_j^s$ is required for the HKM direction or NT direction, respectively. In many large SDP problems, the matrix $\mathbf{smat}(A_j^s(:,k))$ is usually very sparse, and it is important to store this matrix as a sparse matrix in MATLAB and perform sparse-dense matrix-matrix multiplication in the matrix products just mentioned whenever possible. Also, entries of this product only need to be computed if they contribute to an entry of M, i.e., if they correspond to a nonzero entry of $A_j^s(:,k')$ for some k'.

2.3.2. 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 (11), it is easily shown that the contribution of the quadratic blocks to the matrix M is given by

$$M^q \;=\; (A^q)^T (\mathcal{E}^q)^{-1}\mathcal{F}^q A^q \;=\; \sum_{i=1}^{n_q} \underbrace{(A_i^q)^T (\mathcal{E}_i^q)^{-1}\mathcal{F}_i^q A_i^q}_{M_i^q}. \tag{22}$$

For the HKM direction, $(\mathcal{E}^q)^{-1}\mathcal{F}^q$ is a block diagonal matrix whose ith block is given by

$$(\mathcal{E}_i^q)^{-1}\mathcal{F}_i^q \;=\; G_i^{-1}\mathbf{Arw}\left(G_i x_i^q\right) G_i^{-1}$$

$$= \frac{1}{\gamma^2(z_i^q)} \left(\langle x_i^q,\, z_i^q \rangle \begin{bmatrix} -1 & 0 \\ 0 & I \end{bmatrix} + \begin{bmatrix} x_i^0 \\ x_i^1 \end{bmatrix} \begin{bmatrix} z_i^0 \\ -z_i^1 \end{bmatrix}^T + \begin{bmatrix} z_i^0 \\ -z_i^1 \end{bmatrix} \begin{bmatrix} x_i^0 \\ x_i^1 \end{bmatrix}^T \right). \quad (23)$$

(Note that $\mathbf{Arw}\left(G_i^{-1} z_i^q\right) = I$.) Thus, we see that matrix $(\mathcal{E}_i^q)^{-1}\mathcal{F}_i^q$ in M_i^q is the sum of a diagonal matrix and a rank-two symmetric matrix. Hence

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

where

$$J_i = \begin{bmatrix} -1 & 0 \\ 0 & I \end{bmatrix}, \quad u_i^q = \left(A_i^q\right)^T \begin{bmatrix} x_i^0 \\ x_i^1 \end{bmatrix}, \quad v_i^q = \left(A_i^q\right)^T \left(\frac{1}{\gamma^2(z_i^q)} \begin{bmatrix} z_i^0 \\ -z_i^1 \end{bmatrix} \right). \quad (25)$$

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 Schur complement matrix 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 Schur complement matrix remains relatively sparse for these problems and it can be factorized directly and cheaply. In Figure 1, the density structures of the Schur complement matrices in the first and later iterations of our algorithm applied to the problem `nql30` depict the situation and are typical for all `nql` and `qssp` problems. Since we initially choose multiples of unit vectors for our variables, all the nonzero elements of the Schur complement matrix in the first iteration come from the nonzero elements of the constraint matrices. Later iterations introduce fewer than 3% new nonzero elements.

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 [9] when solving the Schur complement equation (10). 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 Challenge set. Figure 2 depicts the Schur complement matrix M in the first iteration and its sparse portion in the following iterations. While these two matrices have almost identical sparsity patterns, the complete Schur complement matrix becomes completely dense after the first iteration.

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

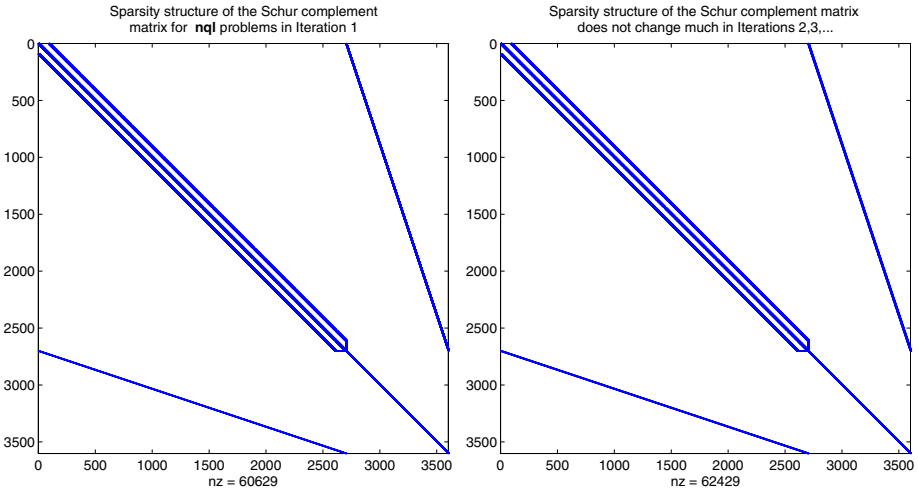

Fig. 1. The output of the `spy` function in MATLAB on the Schur complement matrix for `nql30`. Later iterations introduce less than 3% new nonzero elements

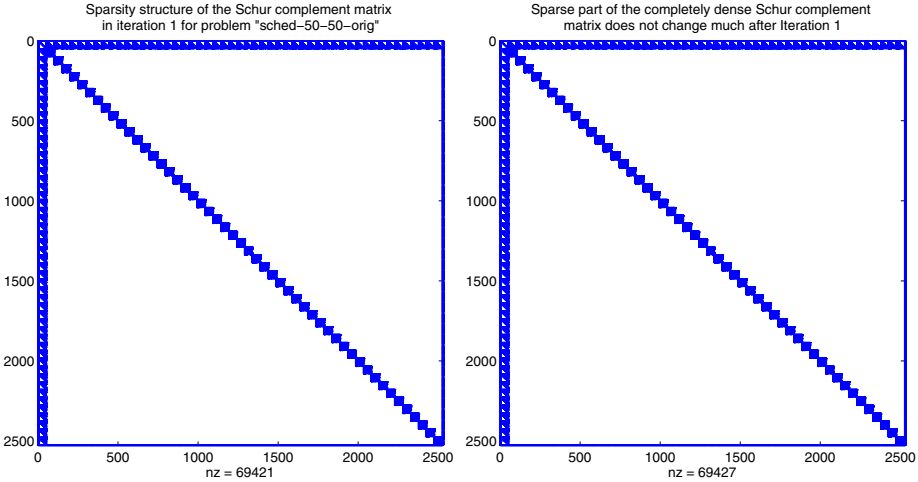

Fig. 2. The output of the `spy` function in MATLAB for problem `sched-50-50-orig` on (i) the complete Schur complement matrix in the first iteration, (ii) the sparse portion of the Schur complement matrix in the following iterations

factorization of the sparse portion of M_i^q need not exist. To overcome this problem, we use the following identity:

$$M_i^q \;=\; \frac{\langle x_i^q,\, z_i^q\rangle}{\gamma^2(z_i^q)}(A_i^q)^T A_i^q \;+\; u_i^q(v_i^q)^T + v_i^q(u_i^q)^T \;-\; 2\frac{\langle x_i^q,\, z_i^q\rangle}{\gamma^2(z_i^q)}k_i k_i^T\,, \tag{26}$$

where u_i^q and v_i^q are as in (25) and

$$k_i = \left(A_i^q\right)^T e_i^q\,. \tag{27}$$

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 (26).

We end our discussion on the computation of the HKM direction with the following formula that is needed in the computation of the right-hand-side vector (12):

$$(\mathcal{E}_i^q)^{-1}(R_c^q)_i \;=\; \frac{\sigma\mu}{\gamma^2(z_i^q)}\begin{bmatrix} z_i^0 \\ -z_i^1 \end{bmatrix} - x_i^q. \tag{28}$$

Just as for the HKM direction, we can obtain a very simple formula for $(\mathcal{E}_i^q)^{-1}\mathcal{F}_i^q$ for the NT direction. By noting that $G_i x_i^q = G_i^{-1} z_i^q$, it is easy to see that the ith block $(\mathcal{E}_i^q)^{-1}\mathcal{F}_i^q = G_i^{-2}$, and a rather straightforward algebraic manipulation gives the following identity:

$$(\mathcal{E}_i^q)^{-1}\mathcal{F}_i^q \;=\; G_i^{-2} \;=\; \frac{1}{\omega_i^2}\left(\begin{bmatrix} -1 & 0 \\ 0 & I \end{bmatrix} + 2\begin{bmatrix} t_i^0 \\ -t_i^1 \end{bmatrix}\begin{bmatrix} t_i^0 \\ -t_i^1 \end{bmatrix}^T\right). \tag{29}$$

For the NT direction, the formula in (28) also holds and we have:

$$M_i^q \;=\; \frac{1}{\omega_i^2}\left((A_i^q)^T J_i A_i^q \;+\; 2u_i^q (u_i^q)^T\right), \text{ with } u_i^q = \left(A_i^q\right)^T \begin{bmatrix} t_i^0 \\ -t_i^1 \end{bmatrix}. \tag{30}$$

We note that the identity (30) describing the NT direction was observed by other authors — see, e.g., [8]. The identities (23) and (24), however, appear to be new in the literature. It is straightforward, if a bit tedious, to verify these formulas. In addition to simplifying the search direction computation, these identities can be used to provide a simple proof of the scale-invariance of the HKM search direction in second-order cone programming. In [27], Tsuchiya proves this result and the scale-invariance of the NT direction using two-page arguments for each proof. We refer the reader to [22] for a description of scale-invariance and provide the following simple and instructive proof:

Proposition 1. *Consider a pure second-order cone programming problem ($n_s = 0$ and n_l=0). The HKM and NT directions for this problem are scale-invariant.*

Proof. The scaled problem is constructed as follows: Let $F_i \in \mathcal{G}_i$ denote a scaling matrix for block i where $\mathcal{G}_i$ is the automorphism group of the cone $K_q^{q_i}$. For future reference, note that we have

$$F_i^T \bar{J}_i F_i = \bar{J}_i, \quad F_i^T \bar{J}_i = \bar{J}_i F_i^{-1}, \quad \bar{J}_i F_i = F_i^{-T} \bar{J}_i, \quad \bar{J}_i F_i^T = F_i^{-1} \bar{J}_i, \tag{31}$$

where $\bar{J}_i = -J_i$, and J_i is as in (25). Let $F = \mathrm{diag}[F_1, \ldots, F_{n_q}]$ and define the scaled quantities as follows:

$$\hat{A}^q = F^{-1}A^q, \quad \hat{b} = b, \quad \hat{c}^q = F^{-1}c^q, \quad \hat{x}^q = F^T x^q, \quad \hat{y} = y, \quad \hat{z}^q = F^{-1}z^q.$$

Note that $\hat{r}_p = r_p$ and $\hat{R}_d = F^{-1}R_d$. First, we consider the HKM direction. We observe that $\gamma^2(\hat{z}_i^q) = (\hat{z}_i^q)^T \bar{J}_i \hat{z}_i^q = (z_i^q)^T F_i^{-T} \bar{J}_i F_i^{-1} z_i^q = \gamma^2(z_i^q)$. Now, from equation (24) and (31) it follows that each M_i^q, and therefore M^q, is invariant with respect to this

automorphic scaling. Using (28), we see that h in (12) is also invariant. Now, if we denote the HKM search direction for the scaled problem by $(\widehat{\Delta x}^q, \widehat{\Delta y}, \widehat{\Delta z}^q)$ and the corresponding direction for the unscaled problem by $(\Delta x^q, \Delta y, \Delta z^q)$ we immediately obtain $\widehat{\Delta y} = \Delta y$ from equation (10), $\widehat{\Delta z}^q = F^{-1}\Delta z^q$ from (13) and $\widehat{\Delta x}^q = F^T \Delta x^q$ from (15) and (28). Thus, the HKM direction for SOCP is scale-invariant.

To prove the result for the NT direction, we first observe that $\gamma(\hat{x}_i^q) = \gamma(x_i^q)$ and that ω_i defined in (20) remains unchanged after scaling. The scaled equivalent of ξ_i defined in (20) is $\hat{\xi}_i = \frac{1}{\omega_i}\hat{z}_i^q + \omega_i \bar{J}_i \hat{x}_i^q = F_i^{-1}(\frac{1}{\omega_i} z_i^q + \omega_i \bar{J}_i x_i^q) = F_i^{-1}\xi_i$. Thus, with the scaled quantities, we obtain

$$\hat{t}_i = \begin{bmatrix} \hat{t}_i^0 \\ \hat{t}_i^1 \end{bmatrix} = F_i^{-1} \begin{bmatrix} t_i^0 \\ t_i^1 \end{bmatrix}.$$

Now, from equation (30) and (31) it follows that each M_i^q, and therefore M^q, is invariant with respect to this automorphic scaling. Continuing as above, we conclude that the NT direction for SOCP must be scale-invariant as well. □

2.4. 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:

$$\alpha_j^s = \begin{cases} \dfrac{-1}{\lambda_{\min}((x_j^s)^{-1}\Delta x_j^s)}, & \text{if the minimum eigenvalue } \lambda_{\min} \text{ is negative} \\ \infty & \text{otherwise.} \end{cases} \tag{32}$$

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 [24]. This approach is quite accurate in general and represents a good trade-off between the 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

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

where J_i is the matrix defined in (25) and let

$$d_i = b_i^2 - a_i c_i.$$

We want the largest α with $a_i\alpha^2 + 2b_i\alpha + c_i > 0$ for all smaller positive values. This 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

$$\alpha \;=\; \min\left(1,\; \gamma \min_{1\le j\le n_s} \alpha_j^s,\; \gamma \min_{1\le i\le n_q} \alpha_i^q,\; \gamma \min_{1\le h\le n_l} \alpha_h^l\right), \tag{33}$$

where γ (known as the step-length parameter) is typically chosen to be a number slightly less than 1, for example as in the adaptive scheme shown in Algorithm IPC, 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

$$\beta \;=\; \min\left(1,\; \gamma \min_{1\le j\le n_s} \beta_j^s,\; \gamma \min_{1\le i\le n_q} \beta_i^q,\; \gamma \min_{1\le h\le n_l} \beta_h^l\right). \tag{34}$$

2.5. Sherman-Morrison-Woodbury formula and iterative refinement

In this subsection, we discuss how we solve the Schur complement equation when M is a low rank perturbation of a sparse matrix. As discussed in Section 2.3 such situations arise when the SQLP does not have a semidefinite block, but has large quadratic blocks or the constraint matrices A_i^q, A^l have a small number of dense rows. In such a case, the Schur complement matrix M can be written in the form

$$M \;=\; H + UV^T \tag{35}$$

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

$$\Delta y \;=\; \hat{h} - H^{-1}U\left(I + V^T H^{-1} U\right)^{-1} V^T \hat{h}, \tag{36}$$

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 overcome such a difficulty, we combine the Sherman-Morrison-Woodbury update with iterative refinement [11]. It is noted in [11] that iterative refinement is beneficial even if the residuals are computed only at the working precision. Our numerical experience with the SQLP problems from the DIMACS Challenge set confirmed that iterative refinement very often does greatly 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 (35), but note that Goldfarb and Scheinberg [8] discuss a stable product-form Cholesky factorization approach to this problem.

3. 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 [7], 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, and

$$
\xi_j^s = s_j \max_{1\le k\le m} \frac{1+|b_k|}{1+\|A_j^s(:,k)\|}, \quad \eta_j^s = \frac{1}{\sqrt{s_j}}\left[1+\max(\max_k\{\|A_j^s(:,k)\|\},\ \|c_j^s\|_F)\right],
$$

$$
\xi_i^q = \sqrt{q_i}\max_{1\le k\le m} \frac{1+|b_k|}{1+\|A_i^q(:,k)\|}, \quad \eta_i^q = \sqrt{q_i}\,[1+\max(\max_k\{\|A_i^q(:,k)\|\},\ \|c_i^q\|)],
$$

$$
\xi^l = \max_{1\le k\le m} \frac{1+|b_k|}{1+\|A^l(:,k)\|}, \quad \eta^l = 1+\max(\max_k\{\|A^l(:,k)\|\},\ \|c^l\|),
$$

where $A_j^s(:,k)$ denotes the kth column of A_j^s, and $A_i^q(:,k)$ and $A_i^l(:,k)$ are defined similarly.

By multiplying the identity matrix I_{s_i} by the factors ξ_i^s and η_i^s 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 above iterate is the default in SDPT3, but other options are also available.

4. Some implementation details

SDPT3 version 3.0 (henceforth denoted SDPT3-3.0) implements the infeasible path-following algorithms described in Section 2. It is designed to be fairly flexible in the strategies used, allowing either the HKM or the NT search direction, switching on or off scaling of the problem and/or the predictor-corrector scheme, giving a choice of step-length determination, etc. (However, as we describe in the next section, the still available version 2.3 (denoted SDPT3-2.3) is more flexible, allowing two more search directions as well as homogeneous self-dual algorithms: we discuss below why these possibilities have been removed from the current version.) Details of these options can be found in the user's guide [28], available from the web sites named in the introduction. The computational results given in the next section were all obtained with default settings, except that we tested both search directions.

The output of SDPT3 is also flexible. Generally the output variables `(X,y,Z)` provide approximately optimal solutions, but if the output variable `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 `A X` small. In the case that an indication of infeasibility is given, the final iterates are still available to the user.

C Mex files used.

Our software uses a number of Mex routines generated from C programs written to carry out certain operations for which MATLAB is not efficient. 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`.

We also use a number of Mex routines generated from the Fortran programs for sparse Cholesky factorization discussed in Section 5.1.

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

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

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

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

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

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

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

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

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

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

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 of our implementation, such as the one described in [25] and SDPT3-2.3. 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. The data format we used in earlier versions of SDPT3 was more natural but, for the sake of computational efficiency, we adopted our current data representation. 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. We also avoid explicit loops over the index k. In the next section, we will discuss the consequence of this modification in our storage scheme.

If the ith block is a quadratic block consisting of numerous sub-blocks, 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

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

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

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

If the ith block is the linear block, then

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

`A{i}` = [`n_l x m sparse`]

`C{i}, X{i}, Z{i}` = [`n_l x 1 double or sparse`].

Caveats.

We should mention that "solving" SQLPs is more subtle 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. (However, in our limited testing on randomly generated strongly infeasible problems, our algorithms have been quite successful in detecting infeasibility.)

5. Computational experiments

Here we describe the results of our computational testing of SDPT3, on problems from the SDPLIB collection of Borchers [4] as well as the DIMACS Challenge test problems [19]. 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 matrix as well as dual methods or nonlinear-programming based methods. We are therefore unable to solve the largest problems.

All results given below were obtained on a Pentium III PC (800MHz) with 1G of memory running Linux, using MATLAB 6.0. The test problems are listed in Tables 1 and 2, along with their dimensions. We also list optimal objective values of these problems that are reported in [19] and [4].

5.1. Cholesky factorization

Earlier versions of SDPT3 were intended for problems that always have semidefinite cone constraints. As we indicated above, for such problems, the Schur complement matrix M in (11) is a dense matrix after the first iteration. To solve the associated linear system (10), 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

Table 1. Selected DIMACS Challenge Problems. SD, SO, and L stand for semidefinite, second-order, and linear blocks, respectively. Notation like [33 x 19] indicates that there were 33 semidefinite blocks, each a symmetric matrix of order 19, etc.

Problem	m	SD	SO	L	opt. obj. value
bm1	883	882	–	–	23.43982
copo14	1275	[14 x 14]	–	364	0
copo23	5820	[23 x 23]	–	1771	0
filter48-socp	969	48	49	931	1.41612901
filtinf1	983	49	49	945	primal inf.
hamming-7-5-6	1793	128	–	–	$42\frac{2}{3}$
hamming-9-8	2305	512	–	–	224
hinf12	43	24	–	–	−0.0231 (?)
hinf13	57	30	–	–	−44.38 (?)
minphase	48	48	–	–	5.98
nb	123	–	[793 x 3]	4	−0.05070309
nb-L1	915	–	[793 x 3]	797	−13.01227
nb-L2	123	–	[1637, 838 x 3]	4	−1.628972
nb-L2-bessel	123	–	[123, 838 x 3]	4	−0.102571
nql30	3680	–	[900 x 3]	3602	−0.9460
nql60	14560	–	[3600 x 3]	14402	−0.935423
nql180	130080	–	[32400 x 3]	129602	−0.927717
nql30old	3601	–	[900 x 3]	5560	−0.9460
nql60old	14401	–	[3600 x 3]	21920	−0.935423
nql180old	129601	–	[32400 x 3]	195360	−0.927717
qssp30	3691	–	[1891 x 4]	2	−6.496675
qssp60	14581	–	[7381 x 4]	2	−6.562696
qssp180	130141	–	[65341 x 4]	2	−6.639527
qssp30old	5674	–	[1891 x 4]	3600	−6.496675
qssp60old	22144	–	[7381 x 4]	14400	−6.562696
qssp180old	196024	–	[65341 x 4]	129600	−6.639527
sched-50-50-orig	2527	–	[2474, 3]	2502	26,673
sched-50-50-scaled	2526	–	2475	2502	7.852038
sched-100-50-orig	4844	–	[4741, 3]	5002	181,889
sched-100-50-scaled	4843	–	4742	5002	67.166281
sched-100-100-orig	8338	–	[8235, 3]	10002	717,367
sched-100-100-scaled	8337	–	8236	10002	27.331457
sched-200-100-orig	18087	–	[17884, 3]	20002	141,360
sched-200-100-scaled	18086	–	17885	20002	51.812471
torusg3-8	512	512	–	–	457.358179
toruspm3-8-50	512	512	–	–	527.808663
truss5	208	[33 x 10, 1]	–	–	132.6356779
truss8	496	[33 x 19, 1]	–	–	133.1145891

on LAPACK, MATLAB's `chol` routine is more efficient than our Cholesky factorization routine `mexchol` for dense matrices. Thus, in SDPT3-3.0, we use MATLAB's `chol` routine whenever M is dense. We also use MATLAB's `chol` in the updated version SDPT3-2.3 of our matrix-based code.

For the solution of most second-order cone programming problems in DIMACS test set, 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 Yin Zhang's LIPSOL [31], an interior-point code for linear programming problems. It should be noted that LIPSOL uses Fortran programs developed by Esmond Ng and Barry Peyton for sparse Cholesky factorization [18]. When SDPT3 uses LIPSOL's Cholesky solver, it first generates

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`

Problem	m	semidefinite blocks	linear block	opt. obj. value
arch8	174	161	174	7.05698
control7	666	[70, 35]	–	20.6251
control10	1326	[100, 50]	–	38.533
control11	1596	[110, 55]	–	31.959
gpp250-4	251	250	–	−747.3
gpp500-4	501	500	–	−1567.02
hinf15	91	37	–	25
mcp250-1	250	250	–	317.2643
mcp500-1	500	500	–	598.1485
qap9	748	82	–	−1410†
qap10	1021	101	–	−1093†
ss30	132	294	132	20.2395
theta3	1106	150	–	42.16698
theta4	1949	200	–	50.32122
theta5	3028	250	–	57.23231
theta6	4375	300	–	63.47709
truss7	86	[150 x 2, 1]	–	−90.0001
truss8	496	[33 x 19, 1]	–	−133.1146
equalG11	801	801	–	629.1553
equalG51	1001	1001	–	4005.601
equalG32	2001	2001	–	N/A
maxG11	800	800	–	629.1648
maxG51	1000	1000	–	4003.809†
maxG32	2000	2000	–	1567.640
qpG11	800	1600	–	2448.659
qpG112	800	800	800	2448.659
qpG51	1000	2000	–	1181.000†
qpG512	1000	1000	1000	1181.000†
thetaG11	2401	801	–	400.00
thetaG11n	1600	800	–	400.00
thetaG51	6910	1001	–	349.00
thetaG51n	5910	1000	–	349.00

† For some problems, we obtained the following alternative objective values that we believe to be more accurate: `qap9`: -1409.8, `qap10`: -1092.4, `equalG32`: 1567.627, `maxG51`: 4006.256, `qpG51`: 1181.800

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. In contrast 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 number of 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 up to two orders of magnitude. SDPT3-3.0 automati-

cally makes a choice between MATLAB's built-in `chol` routine and the sparse Cholesky solver based on the density of the Schur complement matrix.

5.2. *Vectorized matrices vs. sparse matrices*

The current release, SDPT3-3.0, of the code stores the constraint matrix in "vectorized" form as described in Sections 2 and 4. In the previous versions, and in SDPT3-2.3, `A` is a doubly subscripted cell array of symmetric matrices for the semidefinite blocks, as we outlined at the end of the previous section. 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 (11) by avoiding loops over the index k. 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 25% for `qpG11` up to 99% for the larger `theta` problems, the `control` problems, `copo14`, `hamming-7-5-6`, and the `nb` problems. Other significant parts are computing the corrector search direction (up to 75%) and computing step lengths (up to 60%).

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.

Results are given in Tables 3 through 6: Tables 3 and 4 give results on the DIMACS problems for both SDPT3-3.0 and SDPT3-2.3, while Tables 5 and 6 give the comparable results for the SDPLIB problems. In all of these, the format is the same. We give the number of iterations required, the time in seconds, and four measures of the precision of the computed answer. These accuracy measures are computed as follows:

$$\begin{aligned}
\text{err}_1 &= \frac{\|Ax - b\|}{1 + \max |b_k|}, \\
\text{err}_3 &= \frac{\|A^T y + z - c\|}{1 + \max |c|}, \\
\text{err}_5 &= \frac{\langle c,\, x\rangle - b^T y}{1 + |\langle c,\, x\rangle| + |b^T y|}, \\
\text{err}_6 &= \frac{\langle x,\, z\rangle}{1 + |\langle c,\, x\rangle| + |b^T y|}.
\end{aligned}$$

In err_3 the norm is subordinate to the inner product and the maximum is taken over all components of c. These measures are almost identical to the measures reported by Mittelmann in [15], except that he uses $\|b\|_1$ instead of $\max |b_k|$ in err_1, and similarly $\|c\|_1$ instead of $\max |c|$ in err_3. Mittelmann also reports cone violation measures err_2 and err_4 which are always zero for our iterates.

In general, both versions of our codes solved all problems without second-order cone constraints to reasonable accuracy (in terms of all measures) using either of the search directions. SDPT3-3.0 had difficulty obtaining high accuracy solutions to several DIMACS problems involving second-order cone constraints. We comment on some of these problems in detail in Section 5.6. We note that on two problems, our codes

Table 3. Computational results on SDP problems in the DIMACS Challenge problems using SDPT3-2.3. These were performed on a Pentium III PC (800MHz) with 1G of memory

	HKM						NT					
Problem	Itn	err_1	err_3	err_5	err_6	time	Itn	err_1	err_3	err_5	err_6	time
bm1	17	1-06	3-13	8-08	1-07	834	14	9-07	5-13	3-02	3-02	2891
copo14	12	2-11	1-14	8-10	1-09	112	12	2-11	9-15	7-10	1-09	111
copo23	16	3-12	6-14	2-10	4-10	4375	16	2-12	6-14	2-10	3-10	4343
hamming-7-5-6	10	2-15	0	2-10	2-10	83	10	3-15	0	2-10	2-10	83
hamming-9-8	12	8-15	0	2-10	2-10	341	12	7-15	0	2-10	2-10	646
hinf12	42	2-08	5-10	−2-01	1-08	6	39	3-08	3-10	−2-01	5-08	7
hinf13	24	8-05	1-12	−2-02	1-04	4	23	8-05	7-13	−2-02	9-05	5
minphase	32	1-08	0	−2-04	3-08	6	37	2-08	0	−4-04	2-06	9
torusg3-8	15	2-11	8-16	1-09	1-09	112	14	2-10	7-16	2-09	2-09	484
toruspm3-8-50	13	2-11	6-16	3-09	3-09	93	14	5-11	6-16	2-10	2-10	470
truss5	19	5-07	6-15	3-08	2-07	34	19	5-07	1-14	−9-08	3-08	37
truss8	22	3-06	8-15	−2-06	2-07	306	21	2-06	1-14	−8-09	2-06	299

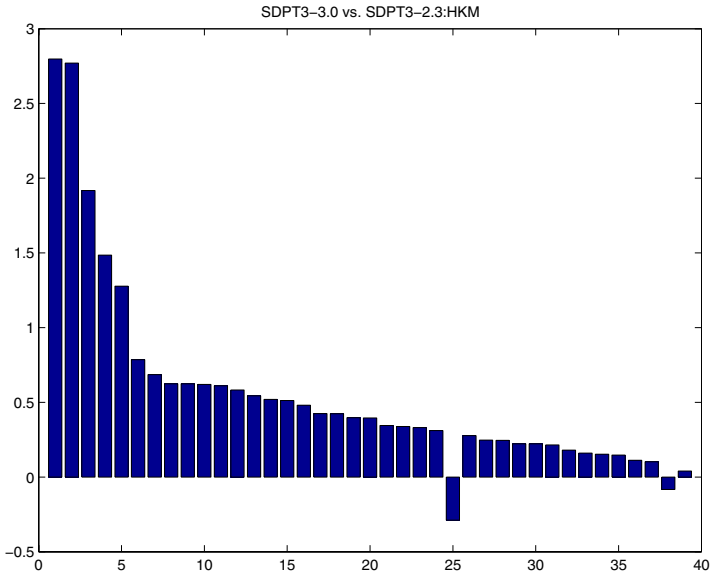

Fig. 3. Comparing SDPT3-3.0 and SDPT3-2.3 using the HKM search direction. Bars above the axis demonstrate a win for 3.0.

terminated with an indication that `X` and `Z` were not both positive definite: `qpG11` (version 2.3, NT only) and `sched-100-100-orig` (version 3.0, NT only). However, this is a conservative test designed to stop if numerical difficulties are imminent. Using SeDuMi's `eigK.m` routine to check the iterates, it was found that in both cases both variables were feasible in the conic constraints.

The objective values of the optimal solutions generated by our codes match the optimal objective values listed in Tables 1 and 2 in most cases. Exceptions, however, are not limited to the problems where our codes had accuracy problems. In particular, for problems `bm1, torusg3-8, qap9, qap10, maxG51`, and `qpG51`, SDPT3

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

	HKM						NT					
Problem	Itn	err_1	err_3	err_5	err_6	time	Itn	err_1	err_3	err_5	err_6	time
bm1	18	4-07	8-12	1-07	1-07	811	16	1-08	5-13	1-05	1-05	2758
copo14	15	1-10	6-15	−1-09	8-10	40	13	6-11	6-15	9-09	7-09	36
copo23	17	2-09	1-14	8-08	5-10	1805	16	8-10	1-14	2-08	5-09	1695
hamming-7-5-6	10	2-15	9-15	9-11	9-11	66	10	2-15	9-15	9-11	9-11	68
hamming-9-8	11	5-15	9-14	4-09	4-09	212	11	5-15	8-14	4-09	4-09	418
hinf12	42	2-08	4-10	−2-01	1-08	5	39	2-08	2-10	−2-01	6-09	5
hinf13	23	9-05	8-13	−2-02	3-05	4	22	1-04	9-13	−2-02	6-06	4
minphase	32	8-09	3-12	−2-04	1-08	5	37	2-08	7-13	−5-04	3-06	7
torusg3-8	15	2-11	7-16	3-10	3-10	89	14	2-10	7-16	3-09	3-09	407
toruspm3-8-50	14	2-11	6-16	2-09	2-09	84	15	4-11	6-16	7-10	7-10	432
truss5	16	4-07	8-15	−3-10	1-07	9	16	4-07	8-15	−1-07	3-08	10
truss8	15	3-06	8-15	−3-06	1-07	44	14	2-06	1-14	5-07	2-06	47
filter48-socp	38	1-06	9-14	1-05	1-06	51	45	1-06	8-14	1-05	1-06	60
filtinf1	27	3-05	4-12	2-01	4-01	38	27	3-05	2-11	4-01	7-01	39
nb	15	1-05	2-09	2-04	2-04	42	14	1-05	1-08	2-04	2-04	31
nb-L1	16	7-05	4-09	2-05	1-05	73	16	2-04	9-11	1-06	9-07	59
nb-L2	12	2-09	1-11	6-09	6-09	57	11	4-09	1-08	5-07	5-07	45
nb-L2-bessel	13	8-06	4-12	9-07	5-08	39	11	3-07	2-09	6-07	7-07	26
nql30	13	6-08	5-09	2-05	4-05	11	16	2-06	3-11	−5-06	1-06	12
nql60	13	4-07	1-08	4-05	1-04	63	15	3-06	2-10	−2-05	9-06	57
nql180	15	1-05	3-08	7-05	1-03	5622	16	7-05	4-10	−-3-03	6-05	3235
nql30old	12	5-05	2-08	−7-05	1-04	12	12	5-05	2-08	−8-05	1-04	12
nql60old	13	1-04	7-09	−8-04	1-04	87	13	9-05	5-08	−4-04	5-04	75
nql180old	9	9-04	3-05	−5-02	2-01	4015	10	1-03	8-06	−3-01	8-02	2742
qssp30	21	7-08	1-09	7-07	8-07	24	18	3-07	2-11	−1-07	3-08	17
qssp60	21	5-05	2-09	6-05	2-05	154	20	3-06	1-11	3-06	1-07	108
qssp180	24	3-04	1-08	8-04	2-04	17714	25	3-05	4-12	7-05	4-08	9790
qssp30old	11	3-04	4-05	5-02	6-02	58	12	5-04	6-05	3-02	4-02	60
qssp60old	11	4-04	2-04	1-01	2-01	397	12	4-04	4-04	1-01	2-01	382
sched-50-50-orig	28	7-04	3-09	−9-05	6-06	21	29	2-04	3-07	−1-05	7-06	20
sched-50-50-scaled	23	1-04	4-15	2-05	3-05	18	22	6-05	4-15	1-06	7-06	16
sched-100-50-orig	39	6-03	3-11	−8-04	2-06	63	33	6-03	2-11	8-04	5-07	50
sched-100-50-scaled	26	8-04	9-13	1-04	1-04	44	22	7-04	1-09	3-04	3-04	35
sched-100-100-orig	33	5-02	1-10	−2-02	4-07	102	50	5-01	3-10	1-00	1-07	136
sched-100-100-scaled	19	4-02	1-14	−4-03	3-06	65	17	3-02	1-14	−2-03	1-02	55
sched-200-100-orig	41	6-03	3-09	−4-03	3-06	348	39	6-03	1-08	−5-03	4-06	309
sched-200-100-scaled	27	3-03	6-09	−8-04	7-04	247	25	3-03	7-10	−1-03	3-04	216

generates accurate solutions whose optimal objective values differ from previously published optimal objective values which we believe to be incorrect. We also provide an accurate optimal objective value for `equalG32` which was previously unavailable. Finally, we note that some of the listed optimal objective values in Table 1 contain sign errors, see the caption for Table 1 for details.

To compare the two codes in terms of time, we consider only the problems that both codes could solve, and omit the simplest problems with times under 20 seconds (the `hinf` problems, `minphase`, and `truss5` and `truss7`). For the remaining problems, we compute the ratio of the times taken by the two codes, take its logarithm to base 2, and then plot the results in decreasing order of absolute values. The results are shown in Figures 3 and 4 for the HKM and NT search directions. A bar of height 1 indicates that SDPT3-3.0 was 2 times faster than SDPT3-2.3, of −1 the reverse, and of 3 that

Table 5. Computational results on SDPLIB problems using SDPT3-2.3. These were performed on a Pentium III PC (800MHz) with 1G of memory

	HKM						NT					
Problem	Itn	err_1	err_3	err_5	err_6	time	Itn	err_1	err_3	err_5	err_6	time
arch8	19	9-10	5-13	2-09	2-09	55	23	1-08	5-13	4-08	4-08	74
control7	23	7-07	2-09	1-07	8-07	149	22	4-07	2-09	1-06	2-06	156
control10	24	6-07	5-09	1-06	3-06	709	24	1-06	6-09	−1-06	8-07	802
control11	24	1-06	6-09	−3-06	4-07	1108	24	1-06	6-09	−1-06	1-06	1245
gpp250-4	15	3-08	5-14	−9-09	4-09	28	13	7-06	2-13	1-05	2-05	62
gpp500-4	15	3-08	2-14	4-09	5-09	169	13	7-08	3-14	2-05	2-05	501
hinf15	24	9-05	2-12	−4-02	2-05	7	23	1-04	1-12	−4-02	2-04	8
mcp250-1	13	2-11	5-16	4-09	4-09	14	15	7-12	4-16	9-10	9-10	55
mcp500-1	14	1-11	5-16	8-10	8-10	79	15	2-11	5-16	4-09	4-09	416
qap9	15	4-08	3-13	−2-05	1-08	19	15	5-08	3-13	−3-05	1-08	20
qap10	14	4-08	3-13	−6-05	1-08	34	14	4-08	4-13	−6-05	5-09	36
ss30	19	2-09	2-13	5-09	5-09	113	24	1-08	2-13	6-08	6-08	242
theta3	14	3-11	6-15	1-09	1-09	40	14	2-10	6-15	4-10	2-10	45
theta4	15	2-10	8-15	2-09	2-09	160	15	4-10	8-15	4-10	2-10	175
theta5	15	2-10	1-14	2-09	2-09	475	14	4-10	1-14	5-09	5-09	483
theta6	15	5-10	1-14	−2-10	4-10	1224	15	5-10	1-14	1-10	7-10	1302
truss7	23	3-06	2-13	−5-06	4-07	6	22	4-06	2-13	−1-05	2-08	8
truss8	22	3-06	8-15	−2-06	2-07	306	21	2-06	8-15	−8-09	2-06	299
equalG11	18	4-11	3-16	4-10	4-10	776	16	1-08	3-16	2-06	2-06	2451
equalG51	20	8-09	4-16	−8-11	2-10	1586	20	5-08	5-16	2-08	2-08	5648
equalG32	19	1-10	2-16	2-09	2-09	10170	15	4-07	2-16	9-05	9-05	33618
maxG11	14	2-11	8-16	2-09	2-09	292	14	4-11	7-16	1-09	1-09	1540
maxG51	16	1-11	4-16	2-09	2-09	951	16	9-11	3-16	3-10	3-10	4171
maxG32	15	1-10	1-15	2-09	2-09	3726	15	2-10	1-15	6-10	6-10	24957
qpG11	14	1-11	0	4-09	4-09	1429	15	2-11	0	4-09	4-09	6789
qpG112	15	2-11	0	4-10	4-10	337	15	6-11	0	4-10	4-10	1693
qpG51	21	6-11	0	4-09	4-09	4518	24	1-09	0	4-09	2-08	19817
qpG512	17	2-10	0	4-09	4-09	965	25	9-11	0	2-09	2-09	6677
thetaG11	20	3-09	8-14	−5-10	2-10	1196	17	8-09	5-15	1-09	3-09	2699
thetaG11n	15	4-12	0	2-09	2-09	786	15	4-12	0	1-09	1-09	2240
thetaG51	33	4-08	7-13	4-09	4-09	18992	30	7-09	8-14	2-08	2-08	23851
thetaG51n	19	2-09	2-14	−1-09	1-08	5159	22	5-09	3-14	−2-09	7-10	10415

SDPT3-3.0 was 8 times faster. Note that the new version using vectorized matrices is almost uniformly faster and often at least 50% faster using either direction.

5.3. HKM vs. NT

The new version of the code allows only two search directions, HKM and NT. Version 2.3 also allows the AHO direction of Alizadeh, Haeberly, and Overton [2] and the GT (Gu-Toh) direction — see [23], but these are uncompetitive 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.

To compare the two remaining search directions, we again use bar charts to show their relative performance as in Figures 3 and 4. The behavior is significantly different depending on whether the problem has semidefinite blocks or not, so we report our results on two charts after categorizing the problems in this way. From Figure 5, it is clear that the HKM direction is almost universally faster than the NT direction for problems with semidefinite blocks. NT offers a slight advantage on only two problems,

Table 6. Computational results on SDPLIB problems using SDPT3-3.0. These were performed on a Pentium III PC (800MHz) with 1G of memory

	HKM						NT					
Problem	Itn	err_1	err_3	err_5	err_6	time	Itn	err_1	err_3	err_5	err_6	time
arch8	21	1-09	5-13	8-10	8-10	41	24	2-08	4-13	1-07	1-07	55
control7	22	5-07	2-09	8-07	1-06	111	22	7-07	2-09	−1-07	6-07	129
control10	24	1-06	6-09	−1-06	3-07	508	24	1-06	6-09	−1-06	7-07	610
control11	24	2-06	6-09	−3-06	3-07	760	23	9-07	6-09	−4-07	2-06	891
gpp250-4	15	8-08	2-12	−1-08	4-09	24	13	7-08	6-14	3-06	3-06	55
gpp500-4	15	5-08	1-12	1-09	3-09	152	18	3-08	1-12	5-09	7-09	601
hinf15	23	9-05	2-12	−4-02	3-06	6	22	1-04	2-12	−5-02	2-05	7
mcp250-1	14	3-12	4-16	1-09	1-09	12	15	1-11	4-16	2-10	2-10	40
mcp500-1	15	1-11	5-16	6-10	6-10	60	16	3-11	5-16	2-10	2-10	327
qap9	15	4-08	3-13	−2-05	1-08	17	15	5-08	3-13	−3-05	1-08	18
qap10	14	4-08	3-13	−6-05	9-09	30	13	4-08	4-13	−6-05	6-08	29
ss30	21	6-09	3-13	1-08	1-08	138	26	3-09	3-13	1-07	1-07	264
theta3	15	2-10	2-14	1-09	1-09	37	14	2-10	2-14	4-10	1-10	39
theta4	15	2-10	3-14	1-09	1-09	129	14	3-10	3-14	6-10	4-10	132
theta5	15	3-10	4-14	2-09	2-09	392	14	4-10	4-14	3-10	3-10	398
theta6	14	2-10	5-14	3-09	3-09	968	14	6-10	5-14	8-10	1-09	1028
truss7	23	3-06	2-13	−4-06	3-07	4	21	2-06	1-13	−2-06	7-08	5
truss8	15	3-06	8-15	−3-06	1-07	44	14	2-06	1-14	5-07	2-06	47
equalG11	17	2-10	4-16	1-09	1-09	611	17	2-10	7-15	1-09	1-09	2210
equalG51	20	2-08	3-14	6-10	5-10	1338	20	9-09	5-16	1-07	1-07	5050
equalG32	19	3-10	2-16	2-09	2-09	8760	19	2-09	4-15	1-09	1-09	36683
maxG11	15	9-12	7-16	5-09	5-09	190	15	4-11	7-16	8-10	8-10	1357
maxG51	17	3-12	2-16	4-10	4-10	617	16	2-10	3-16	2-09	2-09	3077
maxG32	16	1-10	1-15	3-09	3-09	2417	16	2-10	1-15	6-10	6-10	23294
qpG11	16	2-11	0	2-10	2-10	1514	15	1-10	0	3-09	3-09	4528
qpG112	18	2-11	0	2-10	2-10	225	17	5-11	0	4-09	4-09	1532
qpG51	17	7-10	0	3-09	3-09	3168	25	8-10	0	4-09	4-09	15422
qpG512	19	4-10	0	5-10	5-10	632	29	6-10	0	4-09	4-09	5664
thetaG11	19	4-09	1-13	3-09	2-09	834	20	2-09	2-14	1-10	5-10	2334
thetaG11n	15	1-12	2-13	4-10	4-10	456	15	1-12	2-13	4-10	4-10	1587
thetaG51	38	1-08	3-13	9-10	1-09	17692	30	2-08	1-12	3-08	3-08	18572
thetaG51n	19	2-09	5-13	−2-09	9-09	3921	23	3-09	5-13	−1-08	7-10	8457

`copo14` and `copo23`, both with many small blocks. The HKM direction is *much* faster on `maxG32` and considerably faster on the other `maxG` and the `qpG` problems.

Figure 6 shows a reversed pattern for problems with quadratic blocks and no semidefinite blocks. The NT direction was faster on all but three of such problems. The reason for this behavior is not hard to understand. By comparing the formula in (23) for the HKM direction with (29) 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, and these computations can dominate the total work in each iteration. Because there is a class of problems on which the NT direction is faster, we feel it is worthwhile to keep both options.

5.4. Homogeneous vs infeasible interior-point methods

Version 2.3 also allows 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

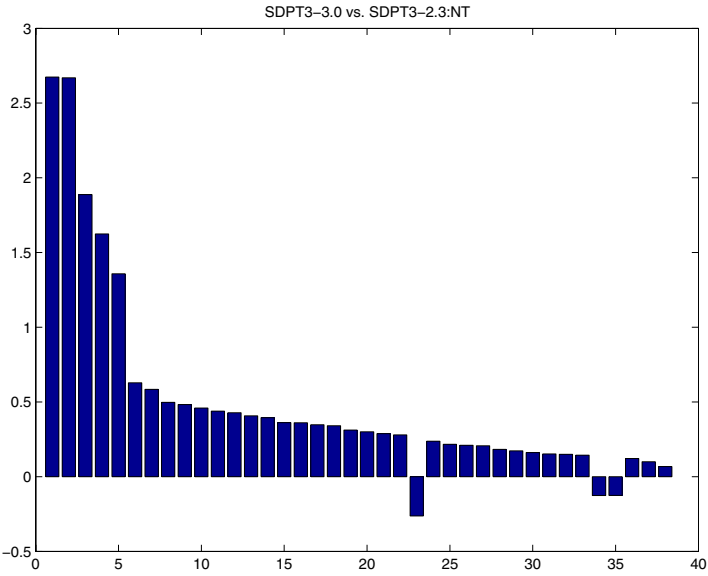

Fig. 4. Comparing SDPT3-3.0 and SDPT3-2.3 using the NT search direction. Bars above the axis demonstrate a win for 3.0

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, in our limited testing on randomly generated infeasible problems, we have observed no advantage to the homogeneous methods in detecting infeasibity. We should mention, however, that SDPT3-3.0 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.

5.5. Presentation of problems

We note that `qpG11` and `qpG112`, and similarly `qpG51` and `qpG512`, are basically the same problem, but in the second version the linear variables are explicitly identified, rather than being part of a large sparse semidefinite block. The improvement in running time is dramatic: a factor of three to five. It is thus crucial to present problems to the algorithms correctly. Unfortunately the versions of `qpG11` and `qpG51` in SDPLIB do not show this structure explicitly. SDPT3-3.0 provides a preprocessor routine called `detect_diag.m` that can be executed to detect such structures.

We also remark that the computation of the Lovasz theta function for a graph can be expressed as a semidefinite programming problem in two ways, and one of these is much more compact than the other, requiring a linear constraint only for each edge

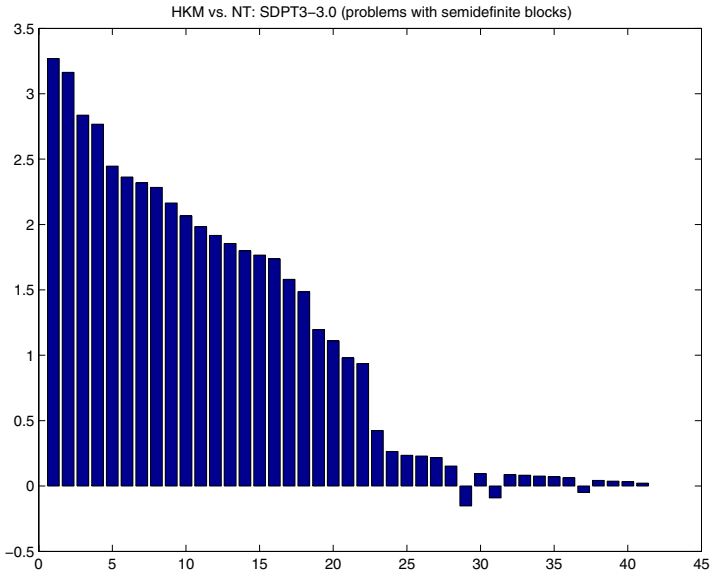

Fig. 5. Comparing the HKM and NT search directions in SDPT3-3.0 for problems that have semidefinite blocks. Bars above the axis demonstrate a win for the HKM direction

of the graph rather than also for each node, and so the problems `thetaG11n` and `thetaG51n` are much easier to solve than `thetaG11` and `thetaG51`, here by a factor up to three, although they are alternative semidefinite formulations of the same graph-theoretic problem for the same graphs.

Finally, SDPT3-2.3 includes 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 SDPT3-2.3, these gains are marginal, since our general sparse matrix routines provide almost as much speedup. We have therefore dropped these routines in SDPT3-3.0.

5.6. Accuracy problems

The accuracy of the solutions obtained by SDPT3-3.0 on several of the problems with second-order cone constraints is less than ideal. While our current implementation may have inherent limitations to solve such problems with high accuracy, the structure of some of these problems contribute to the accuracy problems SDPT3-3.0 faced on them. Many of these problems, including the problems from the **antenna** set (`nb`, `nb-L1`, `nb-L2`, and `nb-L2-bessel`), from the **nql** set (`nql30`, `nql30old`, etc.) and the **qssp** set (`qssp30`, `qssp30old`, etc.), are reformulations of problems with variables that are unrestricted in sign. In these problems, free variables are represented as the difference of two nonnegative variables to fit the problem into the standard form conic optimization framework. This transformation has some well-known unpleasant consequences for interior-point algorithms. The primal optimal face is unbounded (when

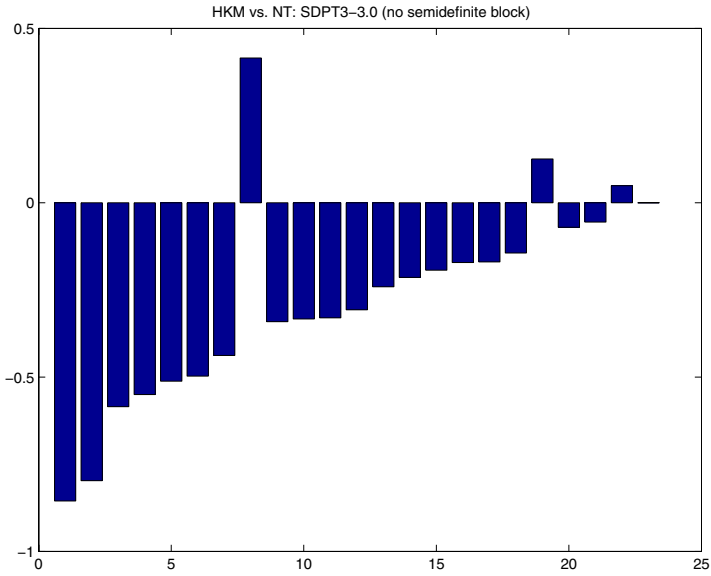

Fig. 6. Comparing the HKM and NT search directions in SDPT3-3.0 for problems with quadratic blocks and no semidefinite blocks. Bars below the axis demonstrate a win for the NT direction

nonempty), the dual feasible set does not have an interior, and there is no central path. For most of these problems, SDPT3-3.0 generates diverging iterates and this creates numerical difficulties as the algorithm progresses. Preliminary experiments showed that free-variable handling techniques that are commonly used in LP solvers can alleviate many of the accuracy problems we observed and such techniques will be included in future releases of our code.

6. Conclusion

Linear optimization problems involving cone constraints (e.g., positive semidefiniteness of variable matrices) continue to arise in a wide array of applications ranging from engineering and control to combinatorial optimization and scheduling. This paper describes computational experiments with a new version of SDPT3, a MATLAB implementation of infeasible-start primal-dual path following algorithms for the solution of such problems. Our experiments indicate that SDPT3 is a very robust code for solving medium-sized semidefinite optimization problems accurately and efficiently. The new version of SDPT3 is significantly faster than the previous versions 2.2 and 2.3. We also observe that, when using the HKM direction one can often obtain similar quality solutions for SDPs two-to-three times faster than when one uses the NT direction.

For problems with second-order (quadratic) cone constraints, experiments indicate that there is room for improvement in SDPT3 — especially with regards to the accuracy of the generated solutions. The efficiency of the two types of search directions we use

is reversed for SOCPs: Similar quality solutions can often be obtained faster using the NT direction rather than the HKM direction.

Overall, SDPT3 finds its niche as an efficient high-quality solver for medium sized semidefinite optimization problems.

References

1. F. Alizadeh, *Interior point methods in semidefinite programming with applications to combinatorial optimization*, SIAM J. Optim., **5** (1995), pp. 13–51.
2. 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. Optim., **8** (1998), pp. 746–768.
3. 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.
4. B. Borchers, *SDPLIB 1.2, a library of semidefinite programming test problems*, Optim. Methods Softw., **11 & 12** (1999), pp. 683–690. Available at `http://www.nmt.edu/~borchers/sdplib.html`.
5. B. Borchers, *CSDP, a C library for semidefinite programming*, Optim. Methods Softw., **11 & 12** (1999), pp. 613–623. Available at `http://www.nmt.edu/~borchers/csdp.html`.
6. K. Fujisawa, M. Kojima, and K. Nakata, *Exploiting sparsity in primal-dual interior-point method for semidefinite programming*, Math. Program., **79** (1997), pp. 235–253.
7. K. Fujisawa, M. Kojima, and K. Nakata, *SDPA (semidefinite programming algorithm) — user's manual*, Research Report, Department of Mathematical and Computing Science, Tokyo Institute of Technology, Tokyo. Available via anonymous ftp at `ftp.is.titech.ac.jp` in `pub/OpRes/software/SDPA`.
8. D. Goldfarb, and K. Scheinberg, *A product-form Cholesky factorization implementation of an interior-point method for second order cone programming*, preprint.
9. G.H. Golub, and C.F. Van Loan, *Matrix Computations*, 2nd ed., (Johns Hopkins University Press, Baltimore, MD, 1989)
10. C. Helmberg, F. Rendl, R. Vanderbei and H. Wolkowicz, *An interior-point method for semidefinite programming*, SIAM J. Optim., **6** (1996), pp. 342–361.
11. N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, (SIAM, Philadelphia, 1996)
12. M. Kojima, S. Shindoh, and S. Hara, *Interior-point methods for the monotone linear complementarity problem in symmetric matrices*, SIAM J. Optim., **7** (1997), pp. 86–125.
13. The MathWorks, Inc., *Using MATLAB*, (The MathWorks, Inc., Natick, MA, 1997)
14. S. Mehrotra, *On the implementation of a primal-dual interior point method*, SIAM J. Optim., **2** (1992), pp. 575–601.
15. H.D. Mittelmann, *An independent benchmarking of SDP and SOCP solvers*, Math. Program., this issue.
16. R.D.C. Monteiro, *Primal-dual path-following algorithms for semidefinite programming*, SIAM J. Optim., **7** (1997), pp. 663–678.
17. Yu. Nesterov and M. J. Todd, *Self-scaled barriers and interior-point methods in convex programming*, Math. Oper. Res., **22** (1997), pp. 1–42.
18. E.G. Ng and B.W. Peyton, *Block sparse Cholesky algorithms on advanced uniprocessor computers*, SIAM J. Sci. Comput., **14** (1993), pp. 1034–1056.
19. G. Pataki and S. Schmieta, *The DIMACS library of mixed semidefinite-quadratic-linear programs*. Available at `http://dimacs.rutgers.edu/Challenges/Seventh/Instances`.
20. F.A. Potra and R. Sheng, *On homogeneous interior-point algorithms for semidefinite programming*, Optim. Methods Softw. **9** (1998), pp. 161–184.
21. J.F. Sturm, *Using SeDuMi 1.02, a Matlab toolbox for optimization over symmetric cones*, Optim. Methods Softw., **11 & 12** (1999), pp. 625–653.
22. M.J. Todd, K.C. Toh, R.H. Tütüncü, *On the Nesterov-Todd direction in semidefinite programming*, SIAM J. Optim., **8** (1998), pp. 769–796.
23. K. C. Toh, *Some new search directions for primal-dual interior point methods in semidefinite programming*, SIAM J. Optim., **11** (2000), pp. 223–242.
24. K.C. Toh, *A note on the calculation of step-lengths in interior-point methods for semidefinite programming*, Comput. Optim. Appl., **21** (2002), pp. 301–310.
25. K.C. Toh, M.J. Todd, R.H. Tütüncü, *SDPT3 — a Matlab software package for semidefinite programming*, Optim. Methods Softw., **11 & 12** (1999), pp. 545-581.
26. T. Tsuchiya, *A polynomial primal-dual path-following algorithm for second-order cone programming*, Technical Report, The Institute of Statistical Mathematics, Minato-Ku, Tokyo, October 1997.

27. T. Tsuchiya, *A convergence analysis of the scaling-invariant primal-dual path-following algorithms for second-order cone programming*, Optim. Methods Softw., **11 & 12** (1999), pp. 141–182.
28. R.H. Tütüncü, K.C. Toh, M.J. Todd, *SDPT3 — a Matlab software package for semidefinite-quadratic-linear programming, version 3.0*. Available at `http://www.math.nus.edu.sg/~mattohkc/`.
29. L. Vandenberghe and S. Boyd, *Semidefinite programming*, SIAM Rev., **38** (1996), pp. 49–95.
30. L. Vandenberghe and S. Boyd, *User's guide to SP: software for semidefinite programming*, Information Systems Laboratory, Stanford University, November 1994. Available via anonymous ftp at `isl.stanford.edu` in `pub/boyd/semidef_prog`. Beta version.
31. Y. Zhang, *Solving large-scale linear programs by interior-point methods under the* MATLAB *environment*, Optim. Methods Softw., **10** (1998), pp. 1–31.