Mathematical Programming manuscript No.
(will be inserted by the editor)

Michal Kočvara · **Michael Stingl**

On the solution of large-scale SDP problems by the modified barrier method using iterative solvers

Dedicated to the memory of Jos Sturm

Received: date / Revised version: date

Abstract The limiting factors of second-order methods for large-scale semidefinite optimization are the storage and factorization of the Newton matrix. For a particular algorithm based on the modified barrier method, we propose to use iterative solvers instead of the routinely used direct factorization techniques. The preconditioned conjugate gradient method proves to be a viable alternative for problems with a large number of variables and modest size of the constrained matrix. We further propose to avoid explicit calculation of the Newton matrix either by an implicit scheme in the matrix-vector product or using a finite-difference formula. This leads to huge savings in memory requirements and, for certain problems, to further speed-up of the algorithm.

1 Introduction

The currently most efficient and popular methods for solving general linear semidefinite programming (SDP) problems

$$\min_{x\in\mathbb{R}^n} f^T x \quad \text{s.t.} \quad \mathscr{A}(x) \preccurlyeq 0 \qquad (\mathscr{A}:\mathbb{R}^n \to \mathbb{S}^m)$$

are the dual-scaling [2] and primal-dual interior-point techniques [1,10,17,22]. These techniques are essentially second-order algorithms: one solves a sequence of unconstrained minimization problems by a Newton-type method. To compute

Michal Kočvara
Institute of Information Theory and Automation, Academy of Sciences of the Czech Republic, Pod vodárenskou věží 4, 18208 Praha 8, Czech Republic and Czech Technical University, Faculty of Electrical Engineering, Technická 2, 166 27 Prague, Czech Republic, E-mail: kocvara@utia.cas.cz

Michael Stingl
Institute of Applied Mathematics, University of Erlangen, Martensstr. 3, 91058 Erlangen, Germany, E-mail: stingl@am.uni-erlangen.de

the search direction, it is necessary to construct a sort of “Hessian” matrix and solve the Newton equation. Although there are several forms of this equation to choose from, the typical choice is the so-called Schur complement equation (SCE) with a symmetric and positive definite Schur complement matrix of order $n \times n$. In many practical SDP problems, this matrix is dense, even if the data matrices are sparse.

Recently, an alternative method for SDP problems has been proposed in [12]. It is based on a generalization of the augmented Lagrangian technique and termed modified barrier or penalty/barrier method. This again is a second-order method and one has to form and solve a sequence of Newton systems. Again, the Newton matrix is of order $n \times n$, symmetric, positive definite and often dense.

The Schur complement or the Newton equations are most frequently solved by routines based on the Cholesky factorization. As a result, the applicability of the SDP codes is restricted by the memory requirements (a need to store a full $n \times n$ matrix) and the complexity of the Cholesky algorithm, $n^3/3$. In order to solve large-scale problems (with n larger than a few thousands), the only option (for interior-point or modified barrier codes) is to replace the direct factorization method by an iterative solver, like a Krylov subspace method.

In the context of primal-dual interior point methods, this option has been proposed by several authors with ambiguous results [5,14,26]. The main difficulty is that, when approaching the optimal point of the SDP problem, the Newton (or SC) matrix becomes more and more ill-conditioned. In order to solve the system by an iterative method, it is necessary to use an efficient preconditioner. However, as we want to solve a general SDP problem, the preconditioner should also be general. And it is well-known that there is no general *and* efficient preconditioner. Consequently, the use of iterative methods in interior-point codes seems to be limited to solving problems with just (very) low accuracy.

A new light on this rather pessimistic conclusion has been shed in the recent papers by Toh and Kojima [24] and later by Toh in [23]. In [23], a symmetric quasi-minimal residual method is applied to an augmented system equivalent to SCE that is further transformed to a so-called reduced augmented system. It is shown that if the SDP problem is primal and dual nondegenerate and strict complementarity holds at the optimal point, the system matrix of the augmented reduced system has a bounded condition number, even close to the optimal point. The use of a diagonal preconditioner enables the authors to solve problems with more than 100 000 variables with a relatively high accuracy.

A promising approach has been also studied by Fukuda et al. [8]. The authors propose a Lagrangian dual predictor-corrector algorithm using the BFGS method in the corrector procedure and the conjugate gradient method for the solution of the linear system. The conjugate gradient method uses as preconditioner the BFGS matrix from the predictor procedure. Although this approach delivers promising results for medium size examples, it remains unclear whether it can be practically efficient for large scale problems.

In this paper, we investigate this approach in the context of the modified barrier method from [12]. Similar to the primal-dual interior-point method from [23], we face highly ill-conditioned dense matrices of the same size. Also similarly to [23], the use of iterative solvers (instead of direct ones) brings greatest advantage when solving problems with very large number of variables (up to one hundred thousand and possibly more) and medium size of the constrained matrix (up to one thousand). Despite of these similarities there are a few significant differences. First, the Schur complement equation has a lot of structure that is used for its further reformulation and for designing the preconditioners. Contrary to that, the Newton matrix in the modified barrier method is just the Hessian of a (generalized) augmented Lagrangian and as such has no intrinsic structure. Further, the condition number of the Hessian itself is bounded close to the optimal point, provided a standard constraint qualification, strict complementarity and a standard second order optimality sufficient condition hold for the SDP problem. We also propose two "Hessian-free" methods. In the first one, the Hessian-vector product (needed by the iterative solver) is calculated by an implicit formula without the need to explicitly evaluate (and store) the full Hessian matrix. In the second one, we use approximate Hessian calculation, based on the finite difference formula for a Hessian-vector product. Both approaches bring us large savings in memory requirements and further significant speed-up for certain classes of problems. Last but not least, our method can also be applied to nonconvex SDP problems [11,13].

We implemented the iterative solvers in the code PENNON [12] and compare the new versions of the code with the standard linear SDP version called PENSDP[1] working with Cholesky factorization. Because other parts of the codes are identical, including stopping criteria, the tests presented here give a clear picture of advantages and disadvantages of each version of the code.

The paper is organized as follows. In Section 2 we present the basic modified barrier algorithm and some details of its implementation. Section 3 offers motivation for the use of iterative solvers based on complexity estimates. In Section 4 we present examples of ill-conditioned matrices arising in the algorithm and introduce the preconditioners used in our testing. The results of extensive tests are presented in Section 5. We compare the new codes on four collections of SDP problems with different backgrounds. In Section 6 we demonstrate that the use of iterative solvers does not necessarily lead to reduced accuracy of the solution. We conclude our paper in Section 7.

We use standard notation: $\mathbb{S}^m$ is the space of real symmetric matrices of dimension $m \times m$. The inner product on $\mathbb{S}^m$ is defined by $\langle A, B\rangle := \operatorname{trace}(AB)$. Notation $A \preccurlyeq B$ for $A, B \in \mathbb{S}^m$ means that the matrix $B - A$ is positive semidefinite.

2 The algorithm

The basic algorithm used in this article is based on the nonlinear rescaling method of R. Polyak [20] and was described in detail in [12]. Here we briefly review it and emphasize points that will be needed in the rest of the paper.

Our goal is to solve optimization problems with a linear objective function subject to a linear matrix inequality as a constraint:

$$\begin{aligned} &\min_{x\in\mathbb{R}^n} f^T x \\ &\text{subject to} \\ &\qquad \mathscr{A}(x) \preccurlyeq 0\,; \end{aligned} \tag{1}$$

[1] See www.penopt.com.

here $f \in \mathbb{R}^n$ and $\mathscr{A} : \mathbb{R}^n \to \mathbb{S}^m$ is a linear matrix operator $\mathscr{A}(x) := A_0 + \sum_{i=1}^n x_i A_i$, $A_i \in \mathbb{S}^m$, $i = 0, 1, \ldots, n$.

The algorithm is based on a choice of a smooth penalty/barrier function $\Phi_p : \mathbb{S}^m \to \mathbb{S}^m$ that satisfies a number of assumptions (see [12]) guaranteeing, in particular, that

$$\mathscr{A}(x) \preccurlyeq 0 \Longleftrightarrow \Phi_p(\mathscr{A}(x)) \preccurlyeq 0\,.$$

Thus for any $p > 0$, problem (1) has the same solution as the following "augmented" problem

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} f^T x \\ &\text{subject to} \\ &\qquad \Phi_p(\mathscr{A}(x)) \preccurlyeq 0\,. \end{aligned} \tag{2}$$

The Lagrangian of (2) can be viewed as a (generalized) augmented Lagrangian of (1):

$$F(x,U,p) = f^T x + \langle U, \Phi_p(\mathscr{A}(x)) \rangle_{\mathbb{S}_m}\,; \tag{3}$$

here $U \in \mathbb{S}^m_+$ is a Lagrangian multiplier associated with the inequality constraint.

The algorithm below can be seen as a generalization of the Augmented Lagrangian method.

Algorithm 1 *Let x^1 and U^1 be given. Let $p^1 > 0$. For $k = 1, 2, \ldots$ repeat until a stopping criterion is reached:*

$$\begin{aligned} (i) \qquad & x^{k+1} = \arg\min_{x \in \mathbb{R}^n} F(x, U^k, p^k) \\ (ii) \qquad & U^{k+1} = D\Phi_p(\mathscr{A}(x^{k+1}))[U^k] \\ (iii) \qquad & p^{k+1} < p^k\,. \end{aligned}$$

Here $D\Phi(X)[Y]$ denotes the directional derivative of Φ with respect to X in direction Y.

By imposing standard assumptions on problem (1), it can be proved that any cluster point of the sequence $\{(x_k, U_k)\}_{k>0}$ generated by Algorithm 1 is an optimal solution of problem (1). The proof is based on extensions of results by Polyak [20]; for the full version we refer to [21]. Let us emphasize a property that is important for the purpose of this article. Assuming the standard constraint qualification, strict complementarity and a standard second order optimality sufficient condition hold at the optimal point, there exists $\overline{p}$ such that the minimum eigenvalue of the Hessian of the Lagrangian (3) is bounded away from zero for all $p \le \overline{p}$ and all (x, U) close enough to the solution (x^*, U^*); see [21]. An analogous result has been proved in [20] in the context of standard inequality constrained nonlinear programming problems.

Details of the algorithm were given in [12]. Hence, in the following we just recall facts needed in the rest of the paper and some new features of the algorithm. The most important fact is that the unconstrained minimization in Step (i) is performed by the Newton method with line-search. Therefore, the algorithm is essentially a *second-order method*: at each iteration we have to compute the Hessian of the Lagrangian (3) and solve a linear system with this Hessian.

2.1 Choice of Φ_p

The penalty function Φ_p of our choice is defined as follows:

$$\Phi_p(\mathscr{A}(x)) = -p^2(\mathscr{A}(x) - pI)^{-1} - pI\,. \tag{4}$$

The advantage of this choice is that it gives closed formulas for the first and second derivatives of Φ_p. Defining

$$\mathscr{Z}(x) = -(\mathscr{A}(x) - pI)^{-1} \tag{5}$$

we have (see [12]):

$$\frac{\partial}{\partial x_i}\Phi_p(\mathscr{A}(x)) = p^2\mathscr{Z}(x)\frac{\partial \mathscr{A}(x)}{\partial x_i}\mathscr{Z}(x) \tag{6}$$

$$\begin{aligned}\frac{\partial^2}{\partial x_i \partial x_j}\Phi_p(\mathscr{A}(x)) = p^2\mathscr{Z}(x)\Bigg(&\frac{\partial \mathscr{A}(x)}{\partial x_i}\mathscr{Z}(x)\frac{\partial \mathscr{A}(x)}{\partial x_j} + \frac{\partial^2 \mathscr{A}(x)}{\partial x_i \partial x_j}\\ &+ \frac{\partial \mathscr{A}(x)}{\partial x_j}\mathscr{Z}(x)\frac{\partial \mathscr{A}(x)}{\partial x_i}\Bigg)\mathscr{Z}(x)\,.\end{aligned} \tag{7}$$

2.2 Multiplier and penalty update, stopping criteria

For the penalty function Φ_p from (4), the formula for update of the matrix multiplier U in Step (ii) of Algorithm 1 reduces to

$$U^{k+1} = (p^k)^2\mathscr{Z}(x^{k+1})U^k\mathscr{Z}(x^{k+1}) \tag{8}$$

with $\mathscr{Z}$ defined as in (5). Note that when U^k is positive definite, so is U^{k+1}. We set U^1 equal to a positive multiple of the identity, thus all the approximations of the optimal Lagrangian multiplier U remain positive definite.

Numerical tests indicate that big changes in the multipliers should be avoided for the following reasons. Big change of U means big change of the augmented Lagrangian that may lead to a large number of Newton steps in the subsequent iteration. It may also happen that already after few initial steps, the multipliers become ill-conditioned and the algorithm suffers from numerical difficulties. To overcome these, we do the following:

1. Calculate U^{k+1} using the update formula in Algorithm 1.
2. Choose a positive $\mu_A \le 1$, typically 0.5.
3. Compute $\lambda_A = \min\left(\mu_A, \mu_A\frac{\|U^k\|_F}{\|U^{k+1}-U^k\|_F}\right)$.
4. Update the current multiplier by

$$U^{new} = U^k + \lambda_A(U^{k+1} - U^k).$$

Given an initial iterate x^1, the initial penalty parameter p^1 is chosen large enough to satisfy the inequality

$$p^1 I - \mathscr{A}(x^1) \succ 0.$$

Let $\lambda_{\max}(\mathscr{A}(x^{k+1})) \in \left(-\infty, p^k\right)$ denote the maximal eigenvalue of $\mathscr{A}(x^{k+1})$, $\pi < 1$ be a constant factor, depending on the initial penalty parameter p^1 (typically chosen between 0.3 and 0.6) and x_{feas} be a feasible point. Let l be set to 0 at the beginning of Algorithm 1. Using these quantities, our strategy for the penalty parameter update can be described as follows:

1. If $p^k < p_{eps}$, set $\gamma = 1$ and go to 6.
2. Calculate $\lambda_{\max}(\mathscr{A}(x^{k+1}))$.
3. If $\pi p^k > \lambda_{\max}(\mathscr{A}(x^{k+1}))$, set $\gamma = \pi$, $l = 0$ and go to 6.
4. If $l < 3$, set $\gamma = \left(\lambda_{\max}(\mathscr{A}(x^{k+1})) + p^k\right)/2p^k$, set $l := l+1$ and go to 6.
5. Let $\gamma = \pi$, find $\lambda \in (0,1)$ such, that
$$\lambda_{\max}\left(\mathscr{A}(\lambda x^{k+1} + (1-\lambda)x_{\text{feas}})\right) < \pi p^k,$$
set $x^{k+1} = \lambda x^{k+1} + (1-\lambda)x_{\text{feas}}$ and $l := 0$.
6. Update current penalty parameter by $p^{k+1} = \gamma p^k$.

The reasoning behind steps 3 to 5 is as follows: As long as the inequality

$$\lambda_{\max}(\mathscr{A}(x^{k+1})) < \pi p^k \tag{9}$$

holds, the values of the augmented Lagrangian in the next iteration remain finite and we can reduce the penalty parameter by the predefined factor π (compare step 3). However, as soon as inequality (9) is violated, an update using π would result in an infinite value of the augmented Lagrangian in the next iteration. Therefore the new penalty parameter should be chosen from the interval $(\lambda_{\max}(\mathscr{A}(x^{k+1})), p^k)$. Because a choice close to the left boundary of the interval leads to large values of the augmented Lagrangian, while a choice close to the right boundary slows down the algorithm, we choose γ such that

$$p^{k+1} = \frac{\lambda_{\max}(\mathscr{A}(x^{k+1})) + p^k}{2}$$

(compare step 4). In order to avoid stagnation of the penalty parameter update process due to repeated evaluations of step 4, we redefine x^{k+1} using the feasible point x_{feas} whenever step 4 is executed in three successive iterations (compare step 5); this is controlled by the parameter l. If no feasible point is yet available, Algorithm 1 is stopped and restarted from the scratch with a different choice of initial multipliers. The parameter p_{eps} is typically chosen as 10^{-6}. In case we detect problems with convergence of Algorithm 1, p_{eps} is decreased and the penalty parameter is updated again, until the new lower bound is reached.

The unconstrained minimization in Step (i) is not performed exactly but is stopped when

$$\left\| \frac{\partial}{\partial x} F(x,U,p) \right\| \le \alpha\,, \tag{10}$$

where $\alpha = 0.01$ is a good choice in most cases. Also here, α is decreased if we encounter problems with accuracy.

To stop the overall Algorithm 1, we have implemented two groups of criteria. Firstly, the algorithm is stopped if both of the following inequalities hold:

$$\frac{|f^T x^k - F(x^k, U^k, p)|}{1 + |f^T x^k|} < \varepsilon, \qquad \frac{|f^T x^k - f^T x^{k-1}|}{1 + |f^T x^k|} < \varepsilon. \tag{11}$$

Secondly, we have implemented the DIMACS criteria [16]. To define these criteria, we rewrite our SDP problem (1) as

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} f^T x \\ &\text{subject to} \\ &\qquad \mathscr{C}(x) \preccurlyeq C_0 \end{aligned} \tag{12}$$

where $\mathscr{C}(x) - C_0 = \mathscr{A}(x)$. Recall that U is the corresponding Lagrangian multiplier and let $\mathscr{C}^*(\cdot)$ denote the adjoint operator to $\mathscr{C}(\cdot)$. The DIMACS error measures are defined as

$$\begin{aligned} \text{err}_1 &= \frac{\|\mathscr{C}^*(U) - f\|}{1 + \|f\|} \\ \text{err}_2 &= \max\left\{0, \frac{-\lambda_{\min}(U)}{1 + \|f\|}\right\} \qquad \text{err}_4 = \max\left\{0, \frac{-\lambda_{\min}(\mathscr{C}(x) - C_0)}{1 + \|C_0\|}\right\} \\ \text{err}_5 &= \frac{\langle C_0, U\rangle - f^T x}{1 + |\langle C_0, U\rangle| + |f^T x|} \qquad \text{err}_6 = \frac{\langle \mathscr{C}(x) - C_0, U\rangle}{1 + |\langle C_0, U\rangle| + |f^T x|}. \end{aligned}$$

Here, err_1 represents the (scaled) norm of the gradient of the Lagrangian, err_2 and err_4 is the dual and primal infeasibility, respectively, and err_5 and err_6 measure the duality gap and the complementarity slackness. Note that, in our code, $\text{err}_2 = 0$ by definition; also err_3 that involves the slack variable (not used in our problem formulation) is automatically zero.

In the code we typically require that (11) is satisfied with $\varepsilon = 10^{-4}$ and, at the same time,

$$\text{err}_k \le \delta_{\text{DIMACS}}, \quad k \in \{1, 4, 5, 6\}. \tag{13}$$

with $\delta_{\text{DIMACS}} = 10^{-7}$.

2.3 Complexity

As mentioned in the Introduction, every second-order method for SDP problems has two bottlenecks: evaluation of the Hessian of the augmented Lagrangian (or a similar matrix of similar size) and the solution of a linear system with this matrix. What are the complexity estimates in our algorithm?

The complexity of Hessian assembling, when working with the function Φ_p from (4) is $O(m^3 n + m^2 n^2)$ for dense data matrices and $O(m^3 + K^2 n^2)$ for sparse data matrices, where K is the maximal number of nonzeros in A_i, $i = 1, \ldots, n$; here we used the sparse techniques described in [7].

In the standard implementation of the algorithm (code PENSDP), we use Cholesky decomposition for the solution of the Newton system (as do all other second-order SDP codes). The complexity of Cholesky algorithm is $O(n^3)$ for dense matrices and $O(n^\kappa)$, $1 \le \kappa \le 3$ for sparse matrices, where κ depends on the sparsity structure of the matrix, going from a diagonal to a full matrix.

As vast majority of linear SDP problems lead to dense Hessians (even if the data matrices A_i are sparse), in the rest of the paper we will concentrate on this situation.

3 Iterative solvers

In step (i) of Algorithm 1 we have to approximately solve an unconstrained minimization problem. As already mentioned before, we use the Newton method with line-search to this purpose. In each iteration step of the Newton method we solve a system of linear equations

$$Hd = -g \tag{14}$$

where H is the Hessian and g the gradient of the augmented Lagrangian (3). In the majority of SDP software (including PENSDP) this (or similar) system is solved by a version of the Cholesky method. In the following we will discuss an alternative approach of solving the linear system by an iterative algorithm.

3.1 Motivation for iterative solvers

Our motivation for the use of iterative solvers is two-fold. Firstly we intend to improve the complexity of the Cholesky algorithm, at least for certain kinds of problems. Secondly, we also hope to improve the complexity of Hessian assembling.

3.1.1 Complexity of Algorithm 1 summarized

The following table summarizes the complexity bottlenecks of Algorithm 1 for the case of linear SDP problems. Recall that K is the maximal number of nonzeros in $A_i,\ i = 1,\dots,n$. Note further that we assume $\mathscr{A}(x)$ to be dense.

Hessian computation	
dense data matrices	$O(m^3n + m^2n^2)$
sparse data matrices	$O(m^3 + K^2n^2)$
Cholesky method	
dense Hessian	$O(n^3)$
sparse Hessian	$O(n^\kappa)$

where $1 \le \kappa \le 3$ depends on the sparsity pattern. This shows that, for dense problems, Hessian computation is the critical issue when m (size of A_i) is large compared to n (number of variables). On the other hand, Cholesky algorithm takes the most time when n is (much) larger than m.

3.1.2 Complexity: Cholesky versus iterative algorithms

At this moment, we should be more specific in what we mean by an iterative solver. In the rest of the paper we will only consider Krylov type methods, in particular, the conjugate gradient (CG) method.

From the complexity viewpoint, the only demanding step in the CG method is a matrix-vector product with a matrix of dimension n (when applied to our system (14)). For a dense matrix and vector, it needs $O(n^2)$ operations. Theoretically, in exact arithmetics, the CG method needs n iterations to find an exact solution of (14), hence it is equally expensive as the Cholesky algorithm. There are, however, two points that may favor the CG method.

First, it is well known that the convergence behavior of the CG method depends solely on the spectrum of the matrix H and the right-hand side vector. In particular, it is given by the condition number and the possible clustering of the eigenvalues; for details, see, e.g., [19]. In practice it means that if the spectrum is "favorable", we may need much smaller number of steps than n, to obtain a reasonably exact solution. This fact leads to a very useful idea of preconditioning when, instead of (14), we solve a "preconditioned" system

$$M^{-1}Hd = -M^{-1}g$$

with a matrix M chosen in such a way that the new system matrix $M^{-1}H$ has a "good" spectrum. The choice of M will be the subject of the next section.

The second, and very important, point is that we actually do not need to have an exact solution of (14). On the contrary, a rough approximation of it will do (compare [9, Thm. 10.2]). Hence, in practice, we may need just a few CG iterations to reach the required accuracy. This is in contrast with the Cholesky method where we cannot control the accuracy of the solution and always have to compute the exact one (within the machine precision). Note that we always start the CG method with initial approximation $d_0 = 0$; thus, performing just one CG step, we would obtain the steepest descend method. Doing more steps, we improve the search direction toward the Newton direction; note the similarity to the Toint-Steihaug method [19].

To summarize these two points: when using the CG algorithm, we may expect to need just $O(n^2)$ operations, at least for well-conditioned (or well-preconditioned) systems.

Note that we are still talking about dense problems. The use of the CG method is a bit nonstandard in this context—usually it is preferable for large sparse problems. However, due to the fact that we just need a very rough approximation of the solution, we may favor it to the Cholesky method also for medium-sized dense problems.

3.1.3 Complexity: explicit versus implicit versus approximate Hessian

Our second goal is to improve the complexity of Hessian computation. When solving (14) by the CG method (and any other Krylov type method), the Hessian is only needed in a matrix-vector product of the type $Hv := \nabla^2 F(x^k)v$. Because we only need to compute the products, we have two alternatives to explicit Hessian calculation—an implicit, operator, formula and an approximate finite-difference formula.

Implicit Hessian formula Instead of computing the Hessian matrix explicitly and then multiplying it by a vector v, we can use the following formula for the Hessian-vector multiplication[2]

$$\nabla^2 F(x^k)v = \mathscr{A}^* \Big((p^k)^2 \mathscr{Z}(x^k) U^k \mathscr{Z}(x^k) A(v) \mathscr{Z}(x^k) \Big), \tag{15}$$

where $A(v) = \sum_{i=1}^n v_i A_i$. Hence, in each CG step, we only have to evaluate matrices $A(v)$ (which is simple), $\mathscr{Z}(x^k)$ and $\mathscr{Z}(x^k)U^k\mathscr{Z}(x^k)$ (which are needed in the gradient computation, anyway), and perform two additional matrix-matrix products. The resulting complexity formula for one Hessian-vector product is thus $O(m^3 + Kn)$.

Additional (but very important) advantage of this approach is the fact that we do not have to store the Hessian in the memory, thus the memory requirements (often the real bottleneck of SDP codes) are drastically reduced.

Approximate Hessian formula We may use a finite difference formula for the approximation of this product

$$\nabla^2 F(x^k)v \approx \frac{\nabla F(x^k + hv) - \nabla F(x^k)}{h} \tag{16}$$

with $h = (1 + \|x^k\|_2 \sqrt{\varepsilon_{\text{FD}}})$; see [19]. In general, ε_{FD} is chosen so that the formula is as accurate as possible and still not influenced by round-off errors. The "best" choice is obviously case dependent; in our implementation, we use $\varepsilon_{\text{FD}} = 10^{-6}$. Hence the complexity of the CG method amounts to the number of CG iterations times the complexity of gradient evaluation, which is of order $O(m^3 + Kn)$. This may be in sharp contrast with the Cholesky method approach by which we have to compute the full Hessian *and* factorize it. Again, we have the advantage that we do not have to store the Hessian in the memory.

Both approaches may have their dark side. With certain SDP problems it may happen that the Hessian computation is not much more expensive than the gradient evaluation. In this case the Hessian-free approaches may be rather time-consuming. Indeed, when the problem is ill-conditioned and we need many CG iterations, we have to evaluate the gradient many (thousand) times. On the other hand, when using Cholesky method, we compute the Hessian just once.

At a first glance, the implicit approach is clearly preferable to the approximate one, due to the fact that it is exact. Note, however, that our final goal is to solve nonlinear SDPs. In this case it may happen that the second order partial derivatives of $\mathscr{A}$ are expensive to compute or not available at all and thus the implicit approach is not applicable. For this reason we also performed testing with the finite difference formula, to see how much is the behavior of the overall algorithm influenced by the use of approximate Hessian calculation.

[2] We are grateful to the anonymous referee for suggesting this option.

3.2 Preconditioned conjugate gradients

We use the very standard preconditioned conjugate gradient method. The algorithm is given below. Because our stopping criterion is based on the residuum, one consider an alternative: the minimum residual method. Another alternative is the QMR algorithm that can be favorable even for symmetric positive definite systems thanks to its robustness.

We solve the system $Hd = -g$ with a symmetric positive definite and, possibly, ill-conditioned matrix H. To improve the conditioning, and thus the behavior of the iterative method, we will solve a transformed system $(C^{-1}HC^{-1})(Cd) = -C^{-1}g$ with C symmetric positive definite. We define the preconditioner M by $M = C^2$ and apply the standard conjugate gradient method to the transformed system. The resulting algorithm is given below.

Algorithm 2 (Preconditioned conjugate gradients) *Given M, set* $d_0 = 0$, $r_0 = g$, *solve* $Mz_0 = r_0$ *and set* $p_0 = -z_0$.

For $k = 0, 1, 2\ldots$ *repeat until convergence:*

$$\begin{array}{rl}
(i) & \alpha_k = \dfrac{r_k^T z_k}{p_k^T H p_k} \\
(ii) & d_{k+1} = d_k + \alpha_k p_k \\
(iii) & r_{k+1} = r_k + \alpha_k H p_k \\
(iv) & \textit{solve } M z_{k+1} = r_{k+1} \\
(v) & \beta_{k+1} = \dfrac{r_{k+1}^T z_{k+1}}{r_k^T z_k} \\
(vi) & p_{k+1} = -r_{k+1} + \beta_{k+1} p_k
\end{array}$$

From the complexity point of view, the only expensive parts of the algorithm are the Hessian-vector products in steps (i) and (iii) (note that only one product is needed) and, of course, the application of the preconditioner in step (iv).

The algorithm is stopped when the scaled residuum is small enough:

$$\|Hd_k + g\|/\|g\| \leq \varepsilon_{\mathrm{CG}}\,,$$

in practice, when

$$\|r_k\|/\|g\| \leq \varepsilon_{\mathrm{CG}}\,.$$

In our tests, the choice $\varepsilon_{\mathrm{CG}} = 5 \cdot 10^{-2}$ was sufficient.

4 Preconditioning

4.1 Conditioning of the Hessian

It is well known that in context of penalty or barrier optimization algorithms the biggest trouble with iterative methods is the increasing ill-conditioning of the Hessian when we approach the optimum of the original problem. Indeed, in certain methods the Hessian may even become singular. The situation is not much better in our case, i.e., when we use Algorithm 1 for SDP problems. Let us demonstrate it on examples.

Consider first problem `theta2` from the SDPLIB collection [3]. The dimension of this problem is $n = 498$. Figure 1 shows the spectrum of the Hessian at the initial and the optimal point of Algorithm 1 (note that we use logarithmic scaling in the vertical axes). The corresponding condition numbers are $\kappa_{\rm ini} = 394$ and $\kappa_{\rm opt} = 4.9 \cdot 10^7$, respectively. Hence we cannot expect the CG method to be

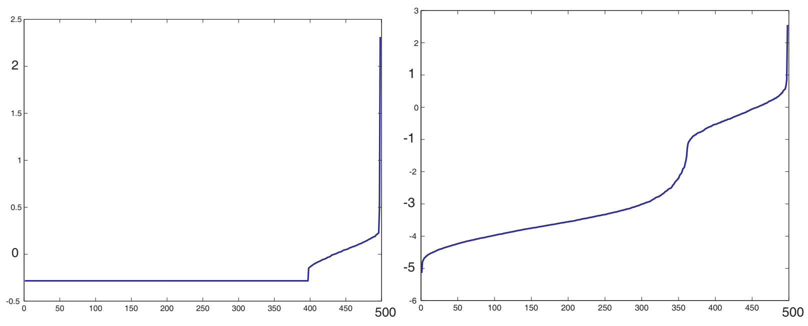

Fig. 1 Example `theta2`: spectrum of the Hessian at the initial (left) and the optimal (right) point.

very effective close to the optimum. Indeed, Figure 2 presents the behavior of the residuum $\|Hd+g\|/\|g\|$ as a function of the iteration count, again at the initial and the optimal point. While at the initial point the method converges in few iterations (due to low condition number and clustered eigenvalues), at the optimal point one observes extremely slow convergence, but still convergence. The zig-zagging na-

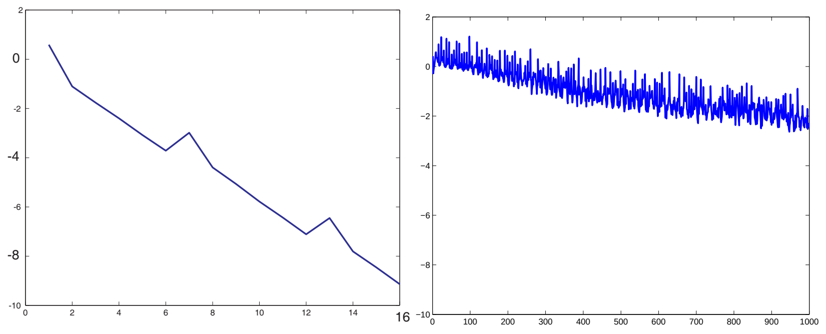

Fig. 2 Example `theta2`: CG behavior at the initial (left) and the optimal (right) point.

ture of the latter curve is due to the fact that CG method minimizes the norm of the error, while here we plot the norm of the residuum. The QMR method offers a much smoother curve, as shown in Figure 3 (left), but the speed of convergence remains almost the same, i.e., slow. The second picture in Figure 3 shows the behavior of the QMR method with diagonal preconditioning. We can see that the convergence speed improves about two-times, which is still not very promising. However, we should keep in mind that we want just an approximation of d and typically stop the iterative method when the residuum is smaller than 0.05; in this case it would be after about 180 iterations.

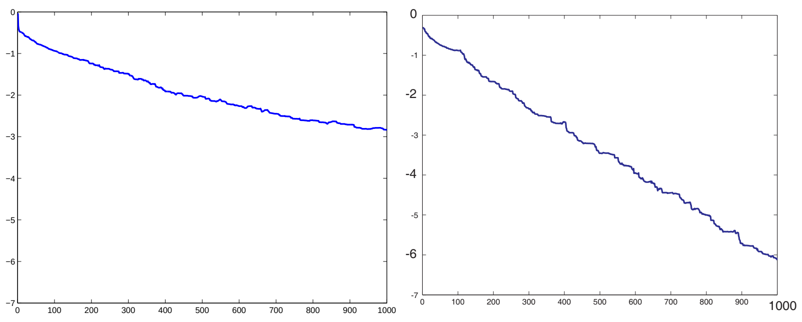

Fig. 3 Example `theta2`: QMR behavior at the optimal (right) point; without (left) and with (right) preconditioning.

The second example, problem `control3` from SDPLIB with $n = 136$, shows even a more dramatic picture. In Figure 4 we see the spectrum of the Hessian, again at the initial and the optimal point. The condition number of these two matrices is $\kappa_{\rm ini} = 3.1 \cdot 10^8$ and $\kappa_{\rm opt} = 7.3 \cdot 10^{12}$, respectively. Obviously, in the second case, the calculations in the CG method are on the edge of machine precision and we can hardly expect convergence of the method. And, indeed, Figure 5 shows that while at $x_{\rm ini}$ we still get convergence of the CG method, at $x_{\rm opt}$ the method does not converge anymore. So, in this case, an efficient preconditioner is a real necessity.

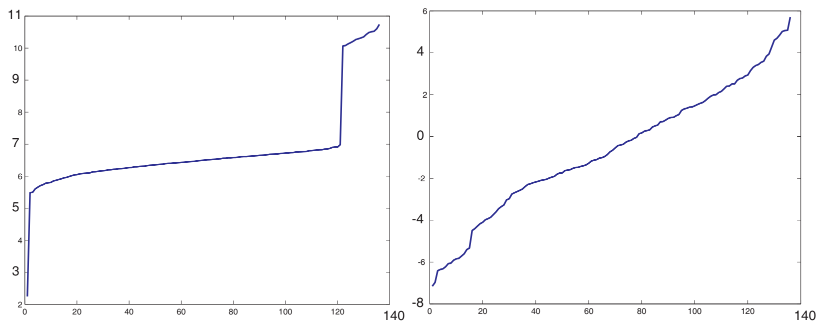

Fig. 4 Example `control3`: spectrum of the Hessian at the initial (left) and the optimal (right) point.

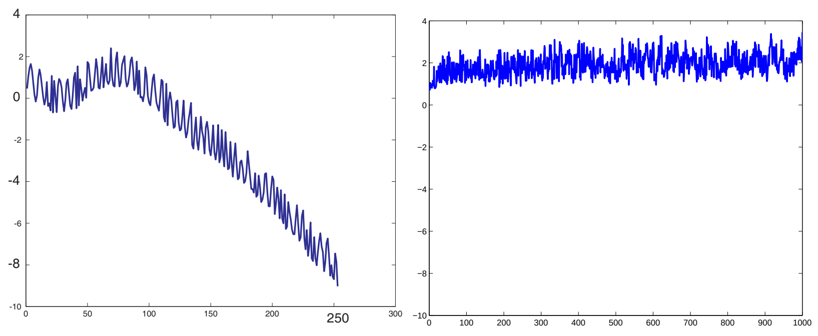

Fig. 5 Example `control3`: CG behavior at the initial (left) and the optimal (right) point.

4.2 Conditions on the preconditioner

Once again, we are looking for a preconditioner—a matrix $M \in \mathbb{S}^n_+$—such that the system $M^{-1}Hd = -M^{-1}g$ can be solved more efficiently than the original system $Hd = -g$. Hence

(i) the preconditioner should be efficient

in the sense that the spectrum of $M^{-1}H$ is "good" for the CG method. Further,

(ii) the preconditioner should be simple.

When applying the preconditioner, in every iteration of the CG algorithm we have to solve the system

$$Mz = p.$$

Clearly, the application of the "most efficient" preconditioner $M = H$ would return us to the complexity of the Cholesky method applied to the original system. Consequently M should be simple enough, so that $Mz = p$ can be solved efficiently.

The above two requirements are general conditions for any preconditioner used within the CG method. The next condition is typical for our application within optimization algorithms:

(iii) the preconditioner should only use Hessian-vector products.

This condition is necessary for the use of the Hessian-free version of the algorithm. We certainly do not want the preconditioner to destroy the Hessian-free nature of this version. When we use the CG method with exact (i.e. computed) Hessian, this condition is not needed.

Finally,

(iv) the preconditioner should be "general".

Recall that we intend to solve general SDP problems without any *a-priori* knowledge about their background. Hence we cannot rely on special purpose preconditioners, as known, for instance, from finite-element discretizations of PDEs.

4.3 Diagonal preconditioner

This is a simple and often-used preconditioner with

$$M = \operatorname{diag}(H).$$

It surely satisfies conditions (ii) and (iv). On the other hand, though simple and general, it is not considered to be very efficient. Furthermore, it does not really satisfy condition (iii), because we need to know the diagonal elements of the Hessian. It is certainly possible to compute these elements by Hessian-vector products. For that, however, we would need n gradient evaluations and the approach would become too costly.

4.4 Symmetric Gauss-Seidel preconditioner

Another classic preconditioner with

$$M = (D+L)^T D^{-1}(D+L) \quad \text{where} \quad H = D - L - L^T$$

with D and L being the diagonal and strictly lower triangular matrix, respectively. Considered more efficient than the diagonal preconditioner, it is also slightly more expensive. Obviously, matrix H must be computed and stored explicitly in order to calculate M. Therefore this preconditioner cannot be used in connection with formula (16) as it does not satisfy condition (iii).

4.5 L-BFGS preconditioner

Introduced by Morales-Nocedal [18], this preconditioner is intended for application within the Newton method. (In a slightly different context, the (L-)BFGS preconditioner was also proposed in [8].) The algorithm is based on limited-memory BFGS formula ([19]) applied to successive CG (instead of Newton) iterations.

Assume we have a finite sequence of vectors x^i and gradients $g(x^i)$, $i = 1,\dots,k$. We define the correction pairs (t^i, y^i) as

$$t^i = x^{i+1} - x^i, \qquad y^i = g(x^{i+1}) - g(x^i), \qquad i = 1,\dots,k-1\,.$$

Using a selection σ of μ pairs from this sequence, such that

$$1 \le \sigma_1 \le \sigma_2 \le \dots \le \sigma_\mu := k-1$$

and an initial approximation

$$W_0 = \frac{(t^{\sigma_\mu})^T y^{\sigma_\mu}}{(y^{\sigma_\mu})^T y^{\sigma_\mu}} I\,,$$

we define the L-BFGS approximation W of the inverse of H; see, e.g. [19]. To compute a product of W with a vector, we use the following algorithm of complexity $n\mu$.

Algorithm 3 (L-BFGS) *Given a set of pairs $\{t^{\sigma_i}, y^{\sigma_i}\}$, $i = 1,2,\ldots,\mu$, and a vector d, we calculate the product $r = Wd$ as*

```
(i)    $q = d$
(ii)   for $i = \mu, \mu-1, \ldots, 1$, put
           $\alpha_i = \frac{(t^{\sigma_i})^T q}{(y^{\sigma_i})^T t^{\sigma_i}}, \qquad q = q - \alpha_i y^{\sigma_i}$
(iii)  $r = W_0 q$
(iv)   for $i = 1, 2, \ldots, \mu$, put
           $\beta = \frac{(y^{\sigma_i})^T r}{(y^{\sigma_i})^T t^{\sigma_i}}, \qquad r = r + t^{\sigma_i}(\alpha_i - \beta)$.
```

The idea of the preconditioner is the following. Assume we want to solve the unconstrained minimization problem in Step (i) of Algorithm 1 by the Newton method. At each Newton iterate $x^{(i)}$, we solve the Newton system $H(x^{(i)})d^{(i)} = -g(x^{(i)})$. The first system at $x^{(0)}$ will be solved by the CG method without preconditioning. The CG iterations $x_\kappa^{(0)}$, $g(x_\kappa^{(0)})$, $\kappa = 1,\ldots,K_0$ will be used as correction pairs to build a preconditioner for the next Newton step. If the number of CG iterations K_0 is higher than the prescribed number of correction pairs μ, we just select some of them (see the next paragraph). In the next Newton step $x^{(1)}$, the correction pairs are used to build an approximation $W^{(1)}$ of the inverse of $H(x^{(1)})$ and this approximation is used as a preconditioner for the CG method. Note that this approximation is not formed explicitly, rather in the form of matrix-vector product $z = W^{(1)}p$ —just what is needed in the CG method. Now, the CG iterations in the current Newton step are used to form new correction pairs that will build the preconditioner for the next Newton step, and so on. The trick is in the assumption that the Hessian at the old Newton step is close enough to the one at the new Newton step, so that its approximation can serve as a preconditioner for the new system.

As recommended in the standard L-BFGS method, we used 16–32 correction pairs, when available. Often the CG method finished in less iterations and in that case we could only use the available iterations for the correction pairs. If the number of CG iterations is higher than the required number of correction pairs μ, the question is how to select these pairs. We have two options: Either we take the last μ pairs or an "equidistant" distribution over all CG iterations. The second option is slightly more complicated but may be expected to deliver better results. The following Algorithm 4 gives a guide to such an equidistant selection.

Algorithm 4 *Given an even number μ, set $\gamma = 1$ and $\mathscr{P} = \emptyset$. For $i = 1,2,\ldots$ do:*

```
Initialization
  If $i < \mu$
    – insert $\{t^i, y^i\}$ in $\mathscr{P}$
Insertion/subtraction
  If $i$ can be written as $i = (\frac{\mu}{2} + \ell - 1)2^\gamma$ for some $\ell \in \{1,2,\ldots,\frac{\mu}{2}\}$ then
    – set index of the subtraction pair as $k = (2\ell - 1)2^{\gamma-1}$
    – subtract $\{t^k, y^k\}$ from $\mathscr{P}$
    – insert $\{t^i, y^i\}$ in $\mathscr{P}$
```

– *if* $\ell = \frac{\mu}{2}$, *set* $\gamma = \gamma + 1$

The L-BFGS preconditioner has the big advantage that it only needs Hessian-vector products and can thus be used in the Hessian-free approaches. On the other hand, it is more complex than the above preconditioners; also our results are not conclusive concerning the efficiency of this approach. For many problems it worked satisfactorily, for some, on the other hand, it even lead to higher number of CG steps than without preconditioner.

5 Tests

For testing purposes we have used the code PENNON, in particular its version for linear SDP problems called PENSDP. The code implements Algorithm 1; for the solution of the Newton system we use either the LAPACK routine DPOTRF based on Cholesky decomposition (dense problems) or our implementation of sparse Cholesky solver (sparse problems). In the test version of the code we replaced the direct solver by conjugate gradient method with various preconditioners. The resulting codes are called

PEN-E-PCG(*prec*) with explicit Hessian calculation
PEN-I-PCG(*prec*) with implicit Hessian calculation (15)
PEN-A-PCG(*prec*) with approximate Hessian calculation (16)

where *prec* is the name of the particular preconditioner. In two latter cases, we only tested the BFGS preconditioner (and a version with no preconditioning). All the other preconditioners either need elements of the Hessian or are just too costly with this respect.

Few words about the accuracy. It has already been mentioned that the conditioning of the Hessian increases as the optimization algorithm gets closer to the optimal point. Consequently, a Krylov-type iterative method is expected to have more and more difficulties when trying to reach higher accuracy of the solution of the original optimization problem. This was indeed observed in practice [23,24]. This ill-conditioning may be so severe that it does not allow to solve the problem within reasonable accuracy at all. Fortunately, this was not observed in the presented approach. We explain this by the fact that, in Algorithm 1, the minimum eigenvalue of the Hessian of the Lagrangian is bounded away from zero, even if we are close to the solution. At the same time, the penalty parameter p (affecting the maximum eigenvalue of the Hessian) is not updated when it reaches certain lower bound p_{eps}; see Section 2.

For the tests reported in this section, we set the stopping criteria in (11) and (13) as $\varepsilon = 10^{-4}$ and $\delta_{\text{DIMACS}} = 10^{-3}$, respectively. At the end of this section we report what happens when we try to increase the accuracy.

The conjugate gradient algorithm was stopped when

$$\|Hd + g\| / \|g\| \leq \varepsilon_{\text{CG}}$$

where $\varepsilon_{\text{CG}} = 5 \cdot 10^{-2}$ was sufficient. This relatively very low accuracy does not significantly influence the behavior of Algorithm 1; see also [9, Thm. 10.2]. On the other hand, it has the effect that for most problems we need a very low number of CG iterations at each Newton step; typically 4–8. Hence, when solving problems with dense Hessians, the complexity of the Cholesky algorithm $O(n^3)$ is replaced by $O(\kappa n^2)$ with $\kappa < 10$. We may thus expect great savings for problems with larger n.

In the following paragraphs we report the results of our testing for four collections of linear SDP test problems: the SDPLIB collection of linear SDPs by Borchers [3]; the set of various large-scale problems collected by Hans Mittelmann and called here HM-problems [15]; the set of examples from structural optimization called TRUSS collection[3]; and a collection of very-large scale problems with relatively small-size matrices provided by Kim Toh and thus called TOH collection [23].

5.1 SDPLIB

Let us start with a comparison of preconditioners for this set of problems. Figure 6 presents a performance profile [6] on three preconditioners: diagonal, BFGS, and symmetric Gauss-Seidel. Compared are the CPU times needed to solve 23 selected problems of the SDPLIB collection. We used the PEN-E-PCG version of the code with explicit Hessian computation. The profile shows that the preconditioners deliver virtually identical results, with SGS having slight edge. Because the BFGS preconditioner is the most universal one (it can also be used with the "I" and "A" version of the code), it will be our choice for the rest of this paragraph.

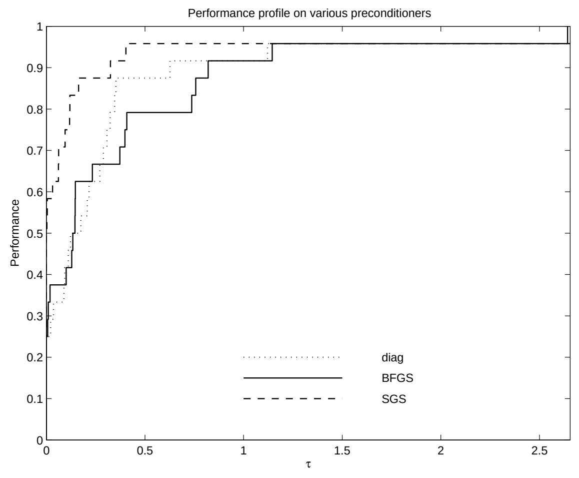

Fig. 6 Performance profile on preconditioners; SDPLIB problems

Table 2 gives a comparison of PENSDP (i.e., a code with Cholesky solver), PEN-E-PCG(BFGS) (with explicit Hessian computation), PEN-I-PCG(BFGS) (with implicit Hessian computation) and PEN-A-PCG(BFGS) (approximate Hessian computation). Not only the CPU times in seconds are given, but also times per

[3] Available at http://www2.am.uni-erlangen.de/∼kocvara/pennon/problems.html

Newton iteration and number of CG steps (when applicable) per Newton iteration. We have chosen the form of a table (contrary to a performance profile), because we think it is important to see the differences between the codes on particular examples. Indeed, while for most examples the PCG-based codes are about as fast as PENSDP, in a few cases they are significantly faster. These examples (`theta*`, `thetaG*`) are typical of a high ratio of n to m. In such situation, the complexity of the solution of the Newton system is much higher than the complexity of Hessian computation and PCG versions of the code are expected to be efficient (see Table 3 and the text below). In (almost) all other problems, most time is spent on Hessian computation and thus the solver of the Newton system does not effect the total CPU time. In a few problems (`control*`, `truss8`), PEN-*-PCG(BFGS) were significantly slower than PENSDP; these are the very ill-conditioned problems when the PCG method needs many iterations to reach even the low accuracy required.

Looking at PEN-I-PCG(BFGS) results, we see even stronger effect "the higher the ratio n to m, the more efficient code". In all examples, this code is faster than the other Hessian-free code PEN-A-PCG(BFGS); in some cases PEN-A-PCG(BFGS) even failed to reach the required accuracy (typically, it failed in the line-search procedure when the search direction delivered by inexact Hessian formula calculations was not a direction of descent).

Table 1 Dimensions of selected SDPLIB problems.

	dimensions	
problem	n	m
arch8	174	335
control7	666	105
equalG11	801	801
equalG51	1001	1001
gpp250-4	250	250
gpp500-4	501	500
maxG11	800	800
maxG32	2000	2000
maxG51	1000	1000
mcp250-1	250	250
mcp500-1	500	500
qap9	748	82
qap10	1021	101
qpG51	1000	2000
ss30	132	426
theta3	1106	150
theta4	1949	200
theta5	3028	250
theta6	4375	300
thetaG11	2401	801
truss8	496	628

The results of the table are also summarized in form of the performance profile in Figure 7. The profile confirms that while PENSDP is the fastest code in most cases, PEN-I-PCG(BFGS) is the best performer in average.

Table 3 compares the CPU time spent in different parts of the algorithm for different types of problems. We have chosen typical representatives of problems

Table 2 Results for selected SDPLIB problems. CPU times in seconds; CPU/it–time per a Newton iteration; CG/it–average number of CG steps per Newton iteration. 3.2Ghz Pentium 4 with 1GB DDR400 running Linux.

	PENSDP		PEN-E-PCG(BFGS)			PEN-I-PCG(BFGS)			PEN-A-PCG(BFGS)		
problem	CPU	CPU/it	CPU	CPU/it	CG/it	CPU	CPU/it	CG/it	CPU	CPU/it	CG/it
arch8	6	0.07	7	0.07	16	10	0.11	20	30	0.29	39
control7	72	0.56	191	1.39	357	157	0.75	238	191	2.45	526
equalG11	49	1.58	66	2.00	10	191	5.79	10	325	9.85	13
equalG51	141	2.76	174	3.55	8	451	9.20	8	700	14.29	10
gpp250-4	2	0.07	2	0.07	4	4	0.14	4	4	0.14	4
gpp500-4	15	0.42	18	0.51	5	32	0.91	5	45	1.29	6
hinf15	421	9.57	178	2.83	8	22	0.40	7	44	0.76	13
maxG11	13	0.36	12	0.33	11	46	1.24	12	66	1.83	12
maxG32	132	4.13	127	3.74	18	634	18.65	18		failed	
maxG51	91	1.82	98	1.78	8	286	5.20	8	527	9.58	9
mcp250-1	1	0.03	1	0.03	5	2	0.07	6	4	0.10	7
mcp500-1	5	0.14	5	0.13	7	10	0.26	7	14	0.39	7
qap9	3	0.09	29	0.22	60	26	0.08	55	48	0.57	273
qap10	9	0.20	34	0.67	67	18	0.24	98		failed	
qpG11	49	1.36	46	1.31	10	214	6.11	10	240	6.86	11
qpG51	181	4.31	191	4.34	3	323	7.34	3	493	11.20	4
ss30	15	0.26	15	0.28	5	10	0.19	5	11	0.20	6
theta3	11	0.22	8	0.14	8	3	0.05	8	5	0.08	10
theta4	40	0.95	30	0.50	12	8	0.14	9	11	0.20	11
theta5	153	3.26	74	1.23	8	14	0.22	7	23	0.40	11
theta6	420	9.55	178	2.83	8	21	0.38	7	44	0.76	13
thetaG11	218	2.63	139	1.70	11	153	1.72	13	359	3.86	26
truss8	9	0.12	23	0.27	115	36	0.40	130		failed	

with $n \approx m$ (`equalG11`) and $n/m \gg 1$ (`theta4`). For all four codes we show the total CPU time spent in the unconstrained minimization, and cumulative times of function and gradient evaluations, Hessian evaluation, and solution of the Newton system. We can clearly see that in the `theta4` example, solution of the Newton system is the decisive part, while in `equalG11` it is the function/gradient/Hessian computation.

Table 3 Cumulative CPU time spent in different parts of the codes: in the whole unconstrained minimization routine (CPU); in function and gradient evaluation (f+g); in Hessian evaluation (hess) ; and in the solution of the Newton system (chol or CG).

	PENSDP				PEN-E-PCG(BFGS)				PEN-I-PCG(BFGS)			PEN-A-PCG(BFGS)		
problem	CPU	f+g	hess	chol	CPU	f+g	hess	CG	CPU	f+g	CG	CPU	f+g	CG
theta4	40.1	0.8	10.5	28.7	30.0	1.6	14.5	13.7	7.4	1.5	5.7	10.8	10.1	0.5
equalG11	44.7	27.5	14.0	1.8	60.2	43.1	14.6	2.3	185.6	42.6	142.6	319.9	318.3	1.4

5.2 HM collection

Table 4 lists a selection of large-scale problems from the HM collection, together with their dimensions and number of nonzeros in the data matrices.

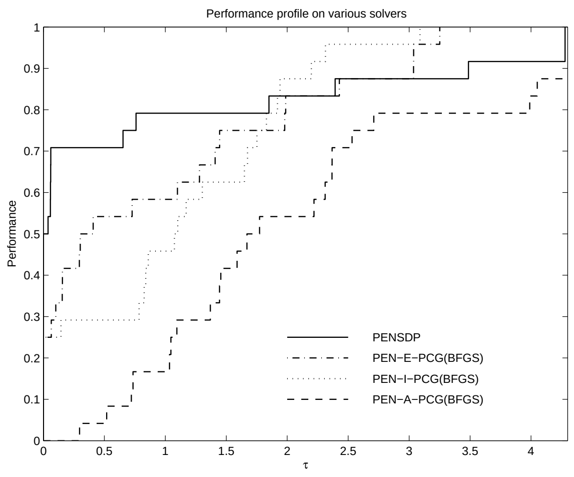

Fig. 7 Performance profile on solvers; SDPLIB problems

Table 4 Dimensions of selected HM-problems.

problem	m	n	nzs	blocks
cancer_100	570	10 470	10 569	2
checker_1.5	3 971	3 971	3,970	2
cnhil10	221	5 005	24 310	2
cnhil8	121	1 716	7 260	2
cphil10	221	5 005	24 310	2
cphil12	364	12 376	66 429	2
foot	2 209	2,209	2 440 944	2
G40_mb	2 001	2 000	2 003 000	2
G40mc	2 001	2 000	2 000	2
G48mc	3 001	3 000	3 000	2
G55mc	5 001	5 000	5 000	2
G59mc	5 001	5 000	5 000	2
hand	1 297	1 297	841 752	2
neosfbr20	363	7 201	309 624	2
neu1	255	3 003	31 880	2
neu1g	253	3 002	31 877	2
neu2c	1 256	3 002	158 098	15
neu2	255	3 003	31 880	2
neu2g	253	3 002	31 877	2
neu3	421	7 364	87 573	3
rabmo	6 827	5 004	60 287	2
rose13	106	2 379	5 564	2
taha1a	1 681	3 002	177 420	15
taha1b	1 610	8 007	107 373	25
yalsdp	301	5 051	1 005 250	4

The test results are collected in Table 5, comparing again PENSDP with PEN-E-PCG(BFGS), PEN-I-PCG(BFGS) and PEN-A-PCG(BFGS). Contrary to the SDPLIB collection, we see a large number of failures of the PCG based codes, due to exceeded time limit of 20000 seconds. This is the case even for problems with large n/m. These problems, all generated by SOSTOOLS or GLOP-

TIPOLY, are typified by high ill-conditioning of the Hessian close to the solution; while in the first few steps of Algorithm 1 we need just few iterations of the PCG method, in the later steps this number becomes very high and the PCG algorithm becomes effectively non-convergent. There are, however, still a few problems with large n/m for which PEN-I-PCG(BFGS) outperforms PEN-E-PCG(BFGS) and this, in turn, clearly outperforms PENSDP: `cancer_100, cphil*, neosfbr20, yalsdp`. These problems are "good" in the sense that the PCG algorithm needs, on average, a very low number of iterations per Newton step. In other problems with this property (like the `G*` problems), n is proportional to m and the algorithm complexity is dominated by the Hessian computation.

Table 5 Results for selected HM-problems. CPU times in seconds; CPU/it–time per a Newton iteration; CG/it–average number of CG steps per Newton iteration. 3.2Ghz Pentium 4 with 1GB DDR400 running Linux; time limit 20 000 sec.

	PENSDP		PEN-E-PCG(BFGS)			PEN-I-PCG(BFGS)			PEN-A-PCG(BFGS)		
problem	CPU	CPU/it	CPU	CPU/it	CG/it	CPU	CPU/it	CG/it	CPU	CPU/it	CG/it
cancer_100	4609	112.41	818	18.18	8	111	2.47	8	373	6.32	19
checker_1.5	1476	20.22	920	15.59	7	2246	39.40	6	3307	58.02	7
cnhil10	664	18.44	3623	77.08	583	703	13.78	477	1380	30.67	799
cnhil8	43	1.13	123	2.62	52	30	0.67	116	87	2.23	280
cphil10	516	18.43	266	9.50	16	15	0.56	17	22	0.79	19
cphil12	5703	219.35	1832	73.28	21	71	2.63	17	92	3.41	19
foot	1480	26.43	2046	33.54	4	3434	57.23	4	4402	73.37	4
G40_mb	987	21.00	1273	26.52	10	4027	83.90	10	5202	110.68	11
G40mc	669	13.94	663	13.26	8	1914	38.28	8	3370	67.40	8
G48mc	408	12.75	332	10.38	1	330	10.31	1	381	11.91	1
G55mc	6491	150.95	6565	139.68	9	18755	416.78	8		timed out	
G59mc	9094	185.59	8593	175.37	8		timed out			timed out	
hand	262	5.95	332	7.38	6	670	14.89	5	1062	23.60	7
neosfbr20	4154	67.00	4001	57.99	109	884	13.19	137	1057	16.02	131
neu1	958	11.01		timed out			timed out			timed out	
neu1g	582	10.78	5677	68.40	1378	1548	20.92	908		timed out	
neu2c	2471	27.46		timed out			timed out			timed out	
neu2	1032	10.98		timed out			timed out			timed out	
neu2g	1444	10.78		timed out			timed out			timed out	
neu3	14402	121.03		timed out			timed out			timed out	
rabmo	1754	18.66		timed out			timed out			timed out	
rose13	77	1.88	862	19.59	685	668	3.01	492	1134	8.86	1259
taha1a	1903	24.40	5976	74.70	1207	6329	72.75	1099	13153	137.01	1517
taha1b	7278	72.06		timed out		9249	79.73	835		timed out	
yalsdp	1817	38.62	2654	54.16	7	29	0.59	7	37	0.77	8

5.3 TRUSS collection

Unlike the previous two collections of problems with different background and of different type, the problems from the TRUSS collection are all of the same type and differ just by the dimension. Looking at the CPU-time performance profile on the preconditioners (Figure 8) we see a different picture than in the previous paragraphs: the SGS preconditioner is the winner, closely followed by the diagonal one; BFGS is the poorest one now. Thus in this collection we only present results of PEN-E-PCG with the SGS preconditioner.

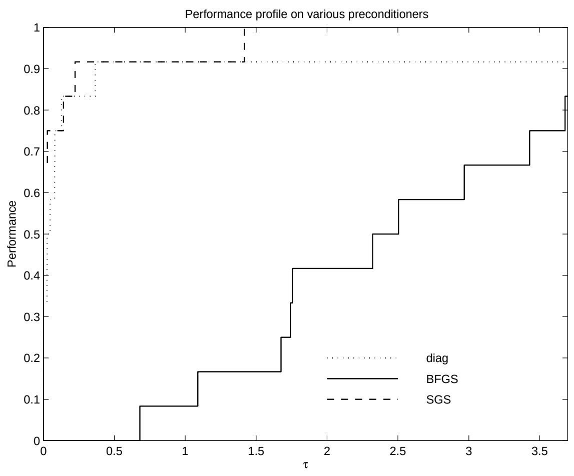

Fig. 8 Performance profile on preconditioners; TRUSS problems

The results of our testing (see Table 6) correspond to our expectations based on complexity estimates. Because the size of the constrained matrices *m* is larger than the number of variables *n*, we may expect most CPU time to be spent in Hessian evaluation. Indeed, for both PENSDP and PEN-E-PCG(SGS) the CPU time per Newton step is about the same in most examples. These problems have ill-conditioned Hessians close to the solution; as a result, with the exception of one example, PEN-A-PCG(BFGS) never converged to a solution and therefore it is not included in the table.

Table 6 Results for selected TRUSS problems. CPU times in seconds; CPU/it–time per a Newton iteration; CG/it–average number of CG steps per Newton iteration. 3.2Ghz Pentium 4 with 1GB DDR400 running Linux; time limit 20 000 sec.

			PENSDP		PEN-PCG(SGS)		
problem	n	m	CPU	CPU/it	CPU	CPU/it	CG/it
buck3	544	1 186	42	0.27	92	0.46	32
buck4	1 200	2 546	183	1.49	421	2.60	40
buck5	3 280	6 802	3215	15.46	10159	27.02	65
trto3	544	866	14	0.13	17	0.18	6
trto4	1 200	1 874	130	0.78	74	0.52	12
trto5	3 280	5 042	1959	8.86	1262	8.89	5
vibra3	544	1 186	39	0.27	132	0.35	10
vibra4	1 200	2 546	177	1.50	449	1.98	11
vibra5	3 280	6 802	2459	15.56		timed out	
shmup3	420	2 642	271	3.15	309	3.81	6
shmup4	800	4 962	1438	15.63	1824	20.49	10
shmup5	1 800	11 042	10317	83.20	16706	112.88	6

5.4 TOH collection

As predicted by complexity results (and as already seen in several examples in the previous paragraphs), PCG-based codes are expected to be most efficient for problems with large n and (relatively) small m. The complexity of the Cholesky algorithm $O(n^3)$ is replaced by $O(10n^2)$ and we may expect significant speed-up of the resulting algorithm. This is indeed the case of the examples from this last collection.

The examples arise from maximum clique problems on randomly generated graphs (`theta*`) and maximum clique problems from the Second DIMACS Implementation Challenge [25].

The dimensions of the problems are shown in Table 7; the largest example has almost 130 000 variables. Note that the Hessians of all the examples are *dense*, so to solve the problems by PENSDP (or by any other interior-point algorithm) would mean to store and factorize a full matrix of dimension 130 000 by 130 000. On the other hand, PEN-I-PCG(BFGS) and PEN-A-PCG(BFGS), being effectively first-order codes, have only modest memory requirements and allow us to solve these large problems within a very reasonable time.

Table 7 Dimensions of selected TOH problems.

problem	n	m
ham_7_5_6	1 793	128
ham_9_8	2 305	512
ham_8_3_4	16 129	256
ham_9_5_6	53 761	512
theta42	5 986	200
theta6	4 375	300
theta62	13 390	300
theta8	7 905	400
theta82	23 872	400
theta83	39 862	400
theta10	12 470	500
theta102	37 467	500
theta103	62 516	500
theta104	87 845	500
theta12	17 979	600
theta123	90 020	600
theta162	127 600	800
keller4	5 101	171
sanr200-0.7	6 033	200

We first show a CPU-time based performance profile on the codes PENSDP, PEN-E-PCG, PEN-I-PCG, and PEN-A-PCG; see Figure 9. All iterative codes used the BFGS preconditioner. We can see dominance of the Hessian-free codes with PEN-I-PCG as a clear winner. From the rest, PEN-E-PCG is clearly faster than PENSDP. Note that, due to memory limitations caused by explicit Hessian calculation, PENSDP and PEN-E-PCG were only able to solve about 60 per cent of the examples.

Table 8 collects the results. As expected, larger problems are not solvable by the second-order codes PENSDP and PEN-E-PCG(BFGS), due to memory limi-

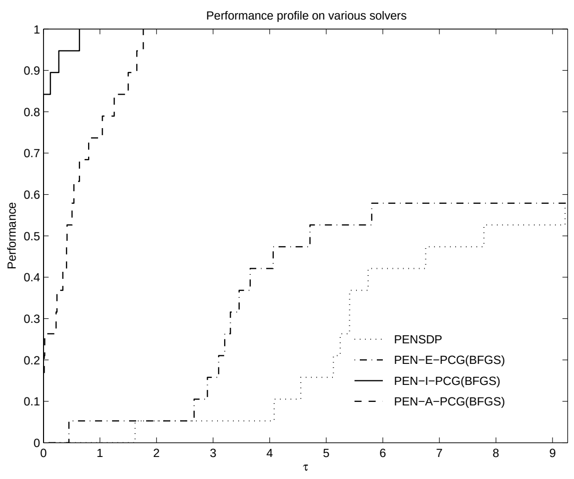

Fig. 9 Performance profile on codes; TOH problems

tations. They can be, on the other hand, easily solved by PEN-I-PCG(BFGS) and PEN-A-PCG(BFGS): to solve the largest problem from the collection, `theta162`, these codes needed just 614 MB of memory. But not only memory is the limitation of PENSDP. In all examples we can see significant speed-up in CPU time going from PENSDP to PEN-E-PCG(BFGS) and further to PEN-I-PCG(BFGS).

To our knowledge, aside from the code described in [23], the only available code capable of solving problems of this size is SDPLR by Burer and Monteiro ([4]). SDPLR formulates the SDP problem as a standard NLP and solves this by a first-order method (Augmented Lagrangian method with subproblems solved by limited memory BFGS). Table 9 thus also contains results obtained by SDPLR; the stopping criterion of SDPLR was set to get possibly the same accuracy as by the other codes. While the `hamming*` problems can be solved very efficiently, SDPLR needs considerably more time to solve the `theta` problems, with the exception of the largest ones. This is due to a very high number of L-BFGS iterations needed.

6 Accuracy

There are two issues of concern when speaking about possibly high accuracy of the solution:

– increasing ill-conditioning of the Hessian of the Lagrangian when approaching the solution and thus decreasing efficiency of the CG method;
– limited accuracy of the finite difference formula in the A-PCG algorithm (approximate Hessian-matrix product computation).

Because the A-PCG algorithm is outperformed by the I-PCG version, we neglect the second point. In the following we will thus study the effect of the DIMACS stopping criterion δ_{DIMACS} on the behavior of PEN-I-PCG.

Table 8 Results for selected TOH problems. CPU times in seconds; CPU/it–time per a Newton iteration; CG/it–average number of CG steps per Newton iteration. AMD Opteron 250/2.4GHz with 4GB RAM; time limit 100 000 sec.

	PENSDP		PEN-E-PCG(BFGS)			PEN-I-PCG(BFGS)			PEN-A-PCG(BFGS)		
problem	CPU	CPU/it	CPU	CPU/it	CG/it	CPU	CPU/it	CG/it	CPU	CPU/it	CG/it
ham_7_5_6	34	0.83	9	0.22	2	1	0.02	2	1	0.02	2
ham_9_8	100	2.13	44	1.02	2	33	0.77	2	38	0.88	2
ham_8_3_4	17701	431.73	1656	39.43	1	30	0.71	1	30	0.71	1
ham_9_5_6	memory			memory		330	7.17	1	333	7.24	1
theta42	1044	23.20	409	7.18	16	25	0.44	9	33	0.61	12
theta6	411	9.34	181	2.97	8	24	0.44	7	69	1.15	17
theta62	13714	253.96	1626	31.88	9	96	1.88	10	62	1.27	5
theta8	2195	54.88	593	9.88	8	93	1.55	10	124	2.07	12
theta82	memory			memory		457	7.62	14	664	12.53	23
theta83	memory			memory		1820	26.00	21	2584	43.07	35
theta10	12165	217.23	1947	29.95	13	227	3.07	10	265	4.27	12
theta102	memory			memory		1299	16.44	13	2675	41.80	35
theta103	memory			memory		2317	37.37	12	5522	72.66	24
theta104	memory			memory		11953	140.62	25	9893	164.88	30
theta12	27565	599.24	3209	58.35	7	254	4.62	8	801	10.01	17
theta123	memory			memory		10538	140.51	23	9670	163.90	27
theta162	memory			memory		13197	173.64	13	22995	365.00	30
keller4	783	14.50	202	3.42	12	19	0.32	9	62	0.72	23
sanr200-0.7	1146	23.88	298	5.32	12	30	0.55	12	47	0.87	18

Table 9 Results for selected TOH problems. Comparison of codes PEN-I-PCG(BFGS) and SDPLR. CPU times in seconds; CPU/it–time per a Newton iteration; CG/it–average number of CG steps per Newton iteration. AMD Opteron 250/2.4GHz with 4GB RAM; time limit 100 000 sec.

	PEN-I-PCG(BFGS)			SDPLR	
problem	CPU	CPU/it	CG/it	CPU	iter
ham_7_5_6	1	0.02	2	1	101
ham_9_8	33	0.77	2	13	181
ham_8_3_4	30	0.71	1	7	168
ham_9_5_6	330	7.17	1	30	90
theta42	25	0.44	9	92	6720
theta6	24	0.44	7	257	9781
theta62	96	1.88	10	344	6445
theta8	93	1.55	10	395	6946
theta82	457	7.62	14	695	6441
theta83	1820	26.00	21	853	6122
theta10	227	3.07	10	712	6465
theta102	1299	16.44	13	1231	5857
theta103	2317	37.37	12	1960	7168
theta104	11953	140.62	25	2105	6497
theta12	254	4.62	8	1436	7153
theta123	10538	140.51	23	2819	6518
theta162	13197	173.64	13	6004	1 6845
keller4	19	0.32	9	29	2922
sanr200-0.7	30	0.55	12	78	5547

We have solved selected examples with several values of $\delta_{\rm DIMACS}$, namely

$$\delta_{\rm DIMACS} = 10^{-1},\ 10^{-3},\ 10^{-5}\,.$$

We have tested two versions of the code a *monotone* and a *nonmonotone* one.

Nonmonotone strategy This is the strategy used in the standard version of the code PENSDP. We set α, the stopping criterion for the unconstrained minimization (10), to a modest value, say 10^{-2}. This value is then automatically recomputed (decreased) when the algorithm approaches the minimum. Hence, in the first iterations of the algorithm, the unconstrained minimization problem is solved very approximately; later, it is solved with increasing accuracy, in order to reach the required accuracy. The decrease of α is based on the required accuracy ε and $\delta_{\rm DIMACS}$ (see (11), (13)). To make this a bit more transparent, we set, for the purpose of testing,

$$\alpha = \min\{10^{-2}, \delta_{\rm DIMACS}\}\,.$$

Monotone strategy In the nonmonotone version of the code, already the first iterations of the algorithm obtained with $\delta_{\rm DIMACS} = 10^{-1}$ differ from those obtained with $\delta_{\rm DIMACS} = 10^{-2}$, due to the different value of α from the very beginning. Sometimes it is thus difficult to compare two runs with different accuracy: theoretically, the run with lower accuracy may need more time than the run with higher required accuracy. To eliminate this phenomenon, we performed the tests with the "monotone" strategy, where we always set

$$\alpha = 10^{-5}\,,$$

i.e., to the lowest tested value of $\delta_{\rm DIMACS}$. By this we guarantee that the first iterations of the runs with different required accuracy will always be the same. Note that this strategy is rather inefficient when low accuracy is required: the code spends too much time in the first iterations to solve the unconstrained minimization problem more exactly than it is actually needed. However, with this strategy we will better see the effect of decreasing $\delta_{\rm DIMACS}$ on the behavior of the (A-)PCG code.

Note that for $\delta_{\rm DIMACS} = 10^{-5}$ both, the monotone and the nonmonotone version coincide. Further, in the table below we only show the DIMACS error measures err_1 (optimality conditions) and err_4 (primal feasibility) that are critical in our code; all the other measures were always well below the required value.

In Table 10 we examine, for selected examples, the effect of increasing Hessian ill-conditioning (when decreasing $\delta_{\rm DIMACS}$) on the overall behavior of the code. We only have chosen examples for which the PCG version of the code is significantly more efficient than the Cholesky-based version, i.e., problems with large factor n/m. The table shows results for both, the monotone and nonmonotone strategy.

We draw two main conclusions from the table: the increased accuracy does not really cause problems; and the nonmonotone strategy is clearly advisable in practice. In the monotone strategy, to reach the accuracy of 10^{-5}, one needs at most 2–3 times more CG steps than for 10^{-1}. In the nonmonotone variant of the code, the CPU time increase is more significant; still it does not exceed the factor 5 which we consider reasonable.

Note also that the actual accuracy is often significantly better than the one required, particularly for $\delta_{\rm DIMACS} = 10^{-1}$. This is due to the fact that the primal stopping criterion (11) with $\varepsilon = 10^{-4}$ is still in effect.

Table 10 Convergence of PEN-I-PCG(BFGS) on selected problems using the monotone (mon=Y) and nonmonotone (mon=N) strategy. Shown are the cumulated CPU time in seconds, number of Newton steps and number of CG iterations and the DIMACS error measures. AMD Opteron 250/2.4GHz running Linux.

δ_{DIMACS}	mon	CPU	Nwt	CG	err_1	err_4	objective
				theta42			
1.0E-01	Y	37	64	856	4.0E-07	1.6E-03	23.931571
1.0E-03	Y	37	64	856	4.0E-07	1.6E-03	23.931571
1.0E-05	Y	77	71	1908	3.8E-07	3.3E-05	23.931708
1.0E-01	N	15	48	303	9.7E-04	3.7E-02	23.931777
1.0E-03	N	19	54	398	2.8E-04	1.8E-03	23.931564
1.0E-05	N	77	71	1908	3.8E-07	3.3E-05	23.931708
				theta6			
1.0E-01	Y	39	71	657	1.5E-07	4.1E-03	63.476509
1.0E-03	Y	39	71	657	1.5E-07	4.1E-03	63.476509
1.0E-05	Y	100	81	1901	2.0E-07	1.1E-05	63.477087
1.0E-01	N	18	52	243	3.4E-03	3.5E-02	63.476054
1.0E-03	N	26	60	413	9.6E-05	4.1E-03	63.476483
1.0E-05	N	100	81	1901	2.0E-07	1.1E-05	63.477087
				cancer-100			
1.0E-01	Y	280	62	850	4.4E-05	7.1E-03	27623.143
1.0E-03	Y	291	64	885	7.7E-05	1.2E-03	27623.292
1.0E-05	Y	666	74	2209	6.8E-06	1.1E-05	27623.302
1.0E-01	N	129	42	353	2.6E-02	0.0E+00	27624.890
1.0E-03	N	181	47	528	3.2E-04	0.0E+00	27623.341
1.0E-05	N	666	74	2209	6.8E-06	1.1E-05	27623.302
				keller4			
1.0E-01	Y	36	72	1131	3.8E-06	2.1E-04	14.012237
1.0E-03	Y	36	72	1131	3.8E-06	2.1E-04	14.012237
1.0E-05	Y	38	74	1257	5.9E-07	1.9E-05	14.012242
1.0E-01	N	12	58	346	1.8E-03	5.4E-04	14.012400
1.0E-03	N	17	60	500	1.9E-05	1.3E-04	14.012248
1.0E-05	N	38	74	1257	5.9E-07	1.9E-05	14.012242
				hamming-9-8			
1.0E-01	Y	38	52	79	4.2E-08	4.7E-05	223.99992
1.0E-03	Y	38	52	79	4.2E-08	4.7E-05	223.99992
1.0E-05	Y	38	52	79	4.2E-08	4.7E-05	223.99992
1.0E-01	N	32	43	66	6.1E-06	5.0E-04	224.00016
1.0E-03	N	37	50	75	1.4E-05	5.6E-04	224.00011
1.0E-05	N	38	52	79	4.2E-08	4.7E-05	223.99992
				neosbfr20			
1.0E-01	Y	3149	95	30678	3.0E-07	3.3E-06	238.56085
1.0E-03	Y	3149	95	30678	3.0E-07	3.3E-06	238.56085
1.0E-05	Y	3149	95	30678	3.0E-07	3.3E-06	238.56085
1.0E-01	N	758	67	7258	2.9E-03	4.2E-06	238.56109
1.0E-03	N	1056	75	10135	4.8E-04	9.3E-06	238.56094
1.0E-05	N	3149	95	30678	3.0E-07	3.3E-06	238.56085

7 Conclusion and outlook

In the framework of a modified barrier method for linear SDP problems, we propose to use iterative solvers for the computation of the search direction, instead of the routinely used factorization technique. The proposed algorithm proved to be more efficient than the standard code for certain groups of examples. The examples for which the new code is expected to be faster can be assigned a priori, based on the complexity estimates (namely on the ratio of the number of variables and the size of the constrained matrix). Furthermore, using an implicit formula for the Hessian-vector product or replacing it by a finite difference formula, we reach huge savings in the memory requirements and, often, further speed-up of the algorithm.

Inconclusive is the testing of various preconditioners. It appears that for different groups of problems different preconditioners are recommendable. While the diagonal preconditioner (considered poor man's choice in the computational linear algebra community) seems to be the most robust one, BFGS preconditioner is the best choice for many problems but, at the same time, clearly the worst one for the TRUSS collection.

Acknowledgements This research was supported by the Academy of Sciences of the Czech Republic through grant No. A1075402. The authors would like to thank Kim Toh for providing them with the collection of "large *n* small *m*" problems. They are also indebted to three anonymous referees for many valuable comments and suggestions.

References

1. Alizadeh, F., Haeberly, J.P.A., Overton, M.L.: Primal–dual interior–point methods for semidefinite programming : Convergence rates, stability and numerical results. SIAM J. Optimization **8**, 746–768 (1998)
2. Benson, S.J., Ye, Y., Zhang, X.: Solving large-scale sparse semidefinite programs for combinatorial optimization. SIAM J. Optimization **10**, 443–462 (2000)
3. Borchers, B.: SDPLIB 1.2, a library of semidefinite programming test problems. Optimization Methods and Software **11 & 12**, 683–690 (1999). Available at http://www.nmt.edu/~borchers/
4. Burer, S., Monteiro, R.: A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization. Math. Prog. (series B), **95(2)**, 329–357 (2003)
5. Choi, C., Ye, Y.: Solving sparse semidefinite programs using the dual scaling algorithm with an iterative solver. Working paper, Computational Optimization Laboratory, University of Iowa, Iowa City, IA (2000)
6. Dolan, E.D., Moré, J.: Benchmarking optimization software with performance profiles. Math. Prog. **91**, 201–213 (2002)
7. Fujisawa, K., Kojima, M., Nakata, K.: Exploiting sparsity in primal-dual interior-point method for semidefinite programming. Math. Prog. **79**, 235–253 (1997)
8. Fukuda, M., Kojima, M., Shida, M.: Lagrangian dual interior-point methods for semidefinite programs. SIAM J. Optimization **12**, 1007–1031 (2002)
9. Geiger, C., Kanzow, C.: Numerische Verfahren zur Lösung unrestringierter Optimierungsaufgaben. Springer-Verlag (1999). In German.
10. Helmberg, C., Rendl, F., Vanderbei, R.J., Wolkowicz, H.: An interior–point method for semidefinite programming. SIAM J. Optimization **6**, 342–361 (1996)
11. Kočvara, M., Leibfritz, F., Stingl, M., Henrion, D.: A nonlinear SDP algorithm for static output feedback problems in COMPlib. LAAS-CNRS Research Report No. 04508, LAAS, Toulouse (2004)
12. Kočvara, M., Stingl, M.: PENNON—a code for convex nonlinear and semidefinite programming. Optimization Methods and Software **18**, 317–333 (2003)
13. Kočvara, M., Stingl, M.: Solving nonconvex SDP problems of structural optimization with stability control. Optimization Methods and Software **19**, 595–609 (2004)
14. Lin, C.J., Saigal, R.: An incomplete cholesky factorization for dense matrices. BIT **40**, 536–558 (2000)
15. Mittelmann, H.: Benchmarks for optimization software; as of November 26, 2005. Available at http://plato.la.asu.edu/bench.html

16. Mittelmann, H.D.: An independent benchmarking of SDP and SOCP solvers. Math. Prog. **95**, 407–430 (2003)
17. Monteiro, R.D.C.: Primal–dual path-following algorithms for semidefinite programming. SIAM J. Optimization **7**, 663–678 (1997)
18. Morales, J.L., Nocedal, J.: Automatic preconditioning by limited memory quasi-Newton updating. SIAM J. Optimization **10**, 1079–1096 (2000)
19. Nocedal, J., Wright, S.: Numerical Optimization. Springer Series in Operations Research. Springer, New York (1999)
20. Polyak, R.: Modified barrier functions: Theory and methods. Math. Prog. **54**, 177–222 (1992)
21. Stingl, M.: On the solution of nonlinear semidefinite programs by augmented Lagrangian methods. Ph.D. thesis, Institute of Applied Mathematics II, Friedrich-Alexander University of Erlangen-Nuremberg (2005). Submitted.
22. Sturm, J.: Primal-dual interior point approach to semidefinite programming. Ph.D. thesis, Tinbergen Institute Research Series vol. 156, Thesis Publishers, Amsterdam, The Netherlands (1997). Available at http://members.tripodnet.nl/SeDuMi/sturm/papers/thesisSTURM.ps.gz
23. Toh, K.C.: Solving large scale semidefinite programs via an iterative solver on the augmented systems. SIAM J. Optimization **14**, 670–698 (2003)
24. Toh, K.C., Kojima, M.: Solving some large scale semidefinite programs via the conjugate residual method. SIAM J. Optimization **12**, 669–691 (2002)
25. Trick, M., Chvátal, V., Cook, W., Johnson, D., McGeoch, C., Trajan, R.: The second DIMACS implementation challenge: NP hard problems: Maximum clique, graph coloring, and satisfiability. Tech. rep., Rutgers University. Available at http://dimacs.rutgers.edu/Challenges/
26. Zhang, S.L., Nakata, K., Kojima, M.: Incomplete orthogonalization preconditioners for solving large and dense linear systems which arise from semidefinite programming. Applied Numerical Mathematics **41**, 235–245 (2002)