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. When solving large-scale semidefinite programming problems by second-order methods, the storage and factorization of the Newton matrix are the limiting factors. 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 large number of variables and modest size of the constrained matrix. We further propose to approximate the Newton matrix in the matrix-vector product by 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

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

are the dual-scaling [2] and primal-dual interior-point techniques [1,8,15,21]. These techniques are essentially second-order algorithms: one solves a sequence of unconstrained minimization problems by the Newton method. To compute the search direction, one has to construct a sort of "Hessian" matrix and solve the Newton equation. Although one can choose from several forms of this equation, 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 [10]. 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 often 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

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

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 [6, 12, 25]. 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 [23] and later by Toh in [22]. In [22], 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.

In this paper, we investigate this approach in the context of the modified barrier method from [10]. Similar to the primal-dual interior-point method from [22], we face highly ill-conditioned dense matrices of the same size. Also similarly to [22], the use of iterative solvers (instead of direct ones) brings biggest 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 few significant differences. First, the SCE 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 the SDP problem is primal and dual nondegenerate and strict complementarity holds at the optimum. We also propose approximate Hessian calculation, based on the finite difference formula for a Hessian-vector product. This brings us huge 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 [9, 11].

We implemented the iterative solvers in the code PENNON [10] 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/disadvantages of each version of the code.

[1] See www.penopt.com.

The paper is organized as follows. In Section 2 we present the basic modified barrier algorithm and some details of its implementation. Section 3 gives motivation for the use of iterative solvers based on complexity estimates. In Section 4 we present examples if 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 background. In Section 6 we demonstrate that the use of iterative solvers does not necessary 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 $\langle A, B\rangle_{\mathbb{S}_m} := \mathrm{trace}(AB)$. 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 was described in detail in [10]. Here we briefly recall it and stress 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:

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

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

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 [10]) guaranteeing, in particular, that

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

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

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

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

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

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:*

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

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 [19]; for the full version we refer to [20]. Let us emphasize a property that is important for the purpose of this article. Assuming the SDP problem is primal and dual nondegenerate and strict complementarity holds at the optimal point, there exist $\overline{p}$ such that the minimum eigenvalue of the Hessian of the Lagrangian (3) is bounded away from zero for all $p \leq \overline{p}$ and all (x, U) close enough to the solution (x^*, U^*).

Details of the algorithm were given in [10]. 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 is 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:

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

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

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

we have (see [10]):

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

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

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

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

with $\mathcal{Z}$ defined as in (5).

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 $\mu_A \leq 1$, 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

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

Let $\lambda_{\max}(\mathcal{A}(x^k)) \in \left(0, p^k\right)$ denote the maximal eigenvalue of $\mathcal{A}(x^k)$, $\pi < 1$ 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. Using this notation, our strategy for the penalty parameter update can be described as follows:

1. If $p < p_{eps}$, set $\gamma = 1$ and go to 6.
2. Calculate $\lambda_{\max}(\mathcal{A}(x^k))$.
3. If $\pi p^k > \lambda_{\max}(\mathcal{A}(x^k))$, set $\gamma = \pi$ and go to 6.
4. If $l < 3$, set $\gamma = \left(\lambda_{\max}(\mathcal{A}(x^k)) + p_k\right)/2$, set $l := l + 1$ and go to 6.
5. Let $\gamma = \pi$, find $\lambda \in (0, 1)$ such, that

$$\lambda_{\max}\left(\mathcal{A}(\lambda x^{k+1} + (1-\lambda) x_{\text{feas}})\right) < \pi p_k$$

   and set $x^{k+1} = \lambda x^{k+1} + (1-\lambda) x_{\text{feas}}$.
6. Update current penalty parameter by $p^{k+1} = \gamma p^k$.

The redefinition of x^{k+1} guarantees that the values of the augmented Lagrangian in the next iteration remain finite. The parameter p_{eps} is typically chosen as 10^{-6}. In case we detect problems with convergence of the overall algorithm, 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

$$\|\frac{\partial}{\partial x} F(x, U, p)\| \leq \alpha\,, \tag{9}$$

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

Algorithm 1 is stopped if both of the following inequalities hold:

$$\frac{|f(x^k) - F(x^k, U^k, p)|}{1 + |f(x^k)|} < \epsilon\,, \qquad \frac{|f(x^k) - f(x^{k-1})|}{1 + |f(x^k)|} < \epsilon\,, \tag{10}$$

where ϵ is typically 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 $O(m^3n + m^2n^2)$ for dense data matrices and $O(m^2n + K^2n^2)$ for sparse data matrices, where K is the maximal number of nonzeros in $A_i,\ i = 1, \ldots, n$.

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. That means, in each iteration step of the Newton method we solve a system of linear equations

$$Hd = -g \tag{11}$$

where H is the Hessian and g the gradient of the augmented Lagrangian 3. In the vast 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. First we intend to improve the complexity of the Cholesky algorithm, at least for certain kind of problems. Second, 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 $A_i,\ i = 1, \ldots, n$.

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

where $1 \leq \kappa \leq 3$ 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 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 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 (11)). 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 (11), 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 in finite arithmetics the actual number of CG iterations, needed to reach a given precision, depends solely on the spectrum of the matrix H, in particular, on the condition number and the possible grouping of the eigenvalues; for details, see, e.g., [18]. 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 the very useful idea of preconditioning when, instead of (11), 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 (11). On the contrary, a rough approximation of it will do (we will return to this in the next section). 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).

Summarizing 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: exact versus approximate Hessian Our second goal is to improve the complexity of Hessian computation. This can be done indirectly. When solving (11) by the CG method (and any other Krylov type method), the Hessian is only needed in a matrix-vector product of the type $Hv := \nabla^2 F(x_k)v$. We may use finite difference formula for the approximation of this product

$$\nabla^2 F(x_k)v \approx \frac{\nabla F(x_k + hv) - \nabla F(x_k)}{h} \tag{12}$$

with $h = (1 + \|x_k\|_2\sqrt{\varepsilon})$. In our implementation, we use $\varepsilon = 10^{-6}$. Hence the complexity of the CG method amounts the number of CG iterations times the complexity of gradient evaluation. This may be in sharp contrast with the Cholesky method approach when we have to compute the full Hessian *and* solve the system by Cholesky method. Additional (perhaps the main) advantage of this Hessian-free 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.

Note that this approach may have its 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 approach 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.

3.2. Preconditioned conjugate gradients

We use the very standard preconditioned conjugate gradient method. The algorithm is recalled below. Because our stopping criterium is based on the residuum, one may think, as an alternative, of the minimum residual method. Another alternative is the QMR algorithm that can be favorable even for symmetric positive definite systems due 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 to the transformed system the standard conjugate gradient method. The resulting algorithm is given below.

Algorithm 2 (Preconditioned conjugate gradients) *Given d_0 and M. Set $r_0 = Hd_0 + g$, $p_0 = g_0$. Solve $Mz_0 = r_0$ and set $p_0 = -z_0$.*

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

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

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:

$$\|Hd_k + g\|/\|g\| \leq \epsilon\,,$$

in practice, when

$$\|r_k\|/\|g\| \leq \epsilon\,.$$

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

4. Preconditioning

4.1. Conditioning of the Hessian

It is well known that the biggest trouble with iterative methods in context of penalty or barrier optimization algorithms 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 in few examples.

Consider first problem `theta2` from the SDPLIB collection [4]. 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 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, though still convergence. The zig-zagging nature of the latter curve is due to the fact that CG method minimizes the norm of the error, while we plot here 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 about the same, i.e., slow. The second picture in Figure 3

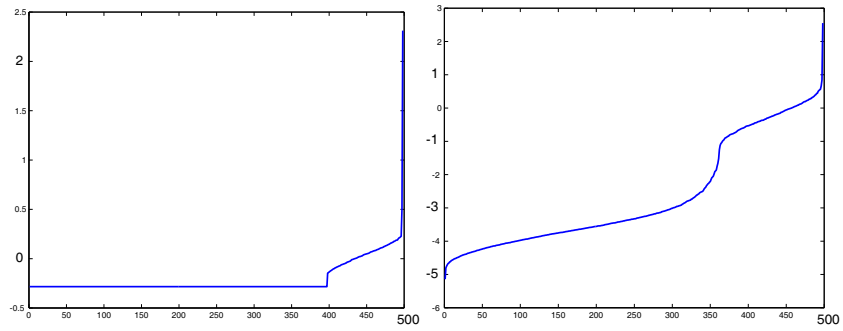

Fig. 1. Example `theta2`: spectrum of the Hessian at the initial (left) and the optimal (right) point.

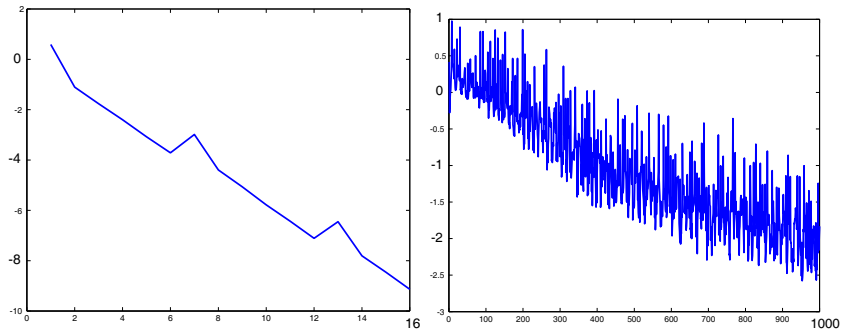

Fig. 2. Example `theta2`: CG behavior at the initial (left) and the optimal (right) point.

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.

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, we are close to machine precision and can hardly expect convergence. 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.

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

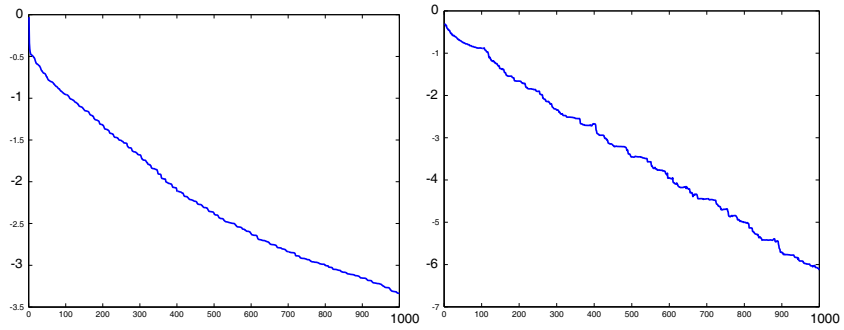

Fig. 3. Example `theta2`: QMR behavior at the optimal (right) point; without (left) and with (right) preconditioning.

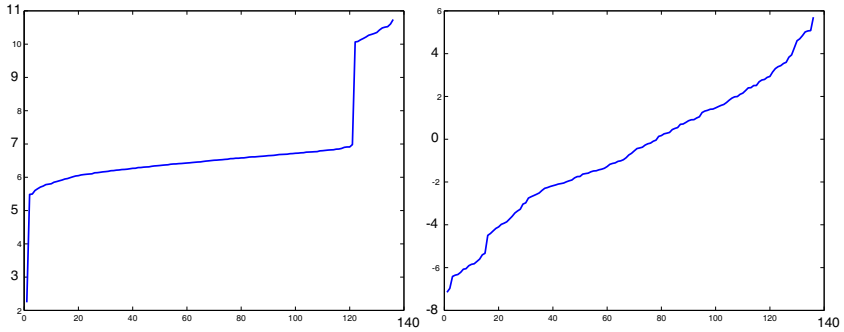

Fig. 4. Example `control3`: spectrum of the Hessian at the initial (left) and the optimal (right) point.

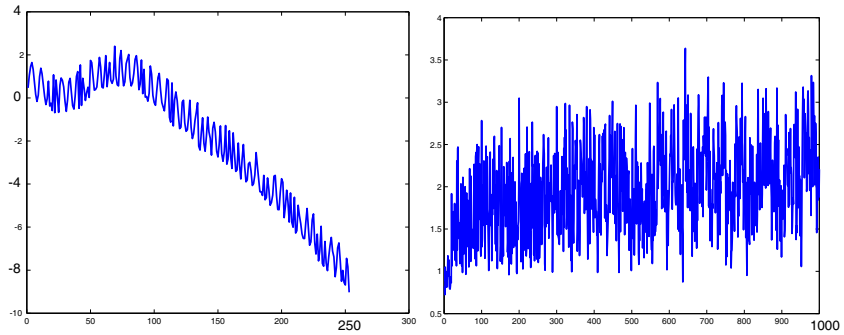

Fig. 5. Example `control3`: CG behavior at the initial (left) and the optimal (right) point.

$Hd = -g$. 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 is for the case when we want to use 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, and this is perhaps the most critical point,

(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

$$M = \mathrm{diag}\,(H).$$

It surely satisfies conditions (ii) and (iv). On the other hand, being 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 approximations of these elements using formula (12). 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. It cannot be used in connection with formula (12) as it does not satisfy condition (iii).

4.5. L-BFGS preconditioner

Introduced by Morales-Nocedal [17], this preconditioner is intended for application within the Newton method. The algorithm is based on limited-memory BFGS formula ([18]) applied to successive CG (instead of Newton) iterations.

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

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

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

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

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. [18]. 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) $\quad q = d$

(ii) $\quad$ *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) $\quad r = W_0 q$

(iv) $\quad$ *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, if they were 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, we may ask 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 we may expect it 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 $\mathcal{P} = \emptyset$. For $i = 1, 2, \ldots$ do:*

Initialization
If $i < \mu$
- *insert $\{t^i, y^i\}$ in $\mathcal{P}$*

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 $\mathcal{P}$*
- *insert $\{t^i, y^i\}$ in $\mathcal{P}$*
- *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 approach. 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 some problems it worked satisfactorily, in others it even lead to higher number of CG steps than without preconditioner.

4.6. AINV preconditioner

Preconditioners based on incomplete matrix factorization are known to be very efficient. Most of them are based on incomplete Cholesky factorization. They are, however, rather complex, need substantial amount of memory and, in particular, need all elements of the system matrix (Hessian). Contrary to this, the AINV preconditioner is based on the H-orthogonalization. It always exists, does not need to know the sparsity pattern of the matrix and only uses matrix-vector products.

Using AINV, we will obtain an approximate factorization of the inverse of H:

$$M = ZD^{-1}Z^T \approx H^{-1}$$

with diagonal D and Z an upper triangular matrix with ones on the diagonal. Z is actually a sparse approximation of L^{-T}, where L is a lower triangular matrix and $H = LDL^T$. Matrix Z is obtained by H-orthogonalization of the unit matrix by the following algorithm ([3]):

Algorithm 5 (AINV) *Initialize $z_i = e_i$ (the unit vector) for $i = 1, \ldots, n$.*
For $i = 1, 2, \ldots, n$ do:

$$
\begin{array}{ll}
(i) & v = Hz_i \\
(ii) & p_i = v^T z_i \\
(iii) & \textit{for } j = i+1, \ldots, n \textit{ do} \\
 & \quad (a) \qquad z_j = z_j - (v^T z_j / p_i) z_i \\
 & \quad (b) \qquad \textit{if } (v^T z_j / p_i) \leq \varepsilon_{\mathrm{AINV}} \textit{ put } z_j = 0
\end{array}
$$

Put $Z = [z_1, z_2, \ldots, z_n]$ and $D = diag(p_1, p_2, \ldots, p_n)$.

Clearly, the critical point of this algorithm is choice of $\varepsilon_{\mathrm{AINV}}$ in step (iii)(b). Unfortunately, in our applications we almost face an "either-or" situation. Either Z is full or Z is empty and we end up with a diagonal preconditioner. Only a narrow range of $\varepsilon_{\mathrm{AINV}}$ leads to sparse Z and even then its quality is often not much better than of the simple diagonal preconditioner (note that, typically, H is dense in our applications). Hence, after many experiments, we set $\varepsilon_{\mathrm{AINV}} = 0.95$ which typically leads to a diagonal preconditioner (however, more expensive than the standard one). Just rarely we met the situation when Z included few nonzero elements.

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-PCG(*prec*), where *prec* is the name of the particular preconditioner. We have further implemented the optional approximate computation of the Hessian based on formula (12). This version of the code is called PEN-A-PCG(*prec*). In this case, we only tested the BFGS preconditioner (and a version with no preconditioning). All other preconditioners either need elements of the Hessian or are just too costly in this context.

Few words about the accuracy. It was already 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, 22]. This ill-conditioning may be so severe that it does not allow one to solve the problem within reasonable accuracy at all. Fortunately, this was not observed in the presented approach. However, to prevent difficulties, we decreased the default PENSDP stopping criterium in (10) from 10^{-7} to 10^{-4}. This still gives a reasonable accuracy (for most applications) of 4–5 digits in the objective value. The reduced stopping criterium was actually important mainly in the A-PCG version of the code, due to the approximate Hessian calculation. At the end of this section we report on what happens when we try to increase the accuracy.

The conjugate gradient algorithm was stopped when

$$\|Hd + g\|/\|g\| \leq \epsilon$$

where $\epsilon = 5 \cdot 10^{-2}$ was sufficient. This relatively very low accuracy does not significantly influence the behavior of Algorithm 1. 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 $\kappa < 10$. For problems with larger n we may thus expect great savings.

In the following paragraphs we report on results of our testing for four collections of linear SDP test problems: the SDPLIB collection of linear SDPs by Borchers [4]; the set of various large-scale problems collected by Hans Mittelmann and called here HM-problems [13]; the set of examples from structural optimization called TRUSS collection[2]; and a collection of very-large scale problems with relatively small-size matrices provided by Kim Toh and thus called TOH collection [22].

5.1. SDPLIB

Let us start with a comparison of preconditioners for this set of problems. Figure 6 presents a performance profile ([7]) on all four preconditioners: diagonal, BFGS, approximate inverse and symmetric Gauss-Seidel. Compared are the CPU times needed to solve 23 selected problems of the SDPLIB collection. We used the version of the code with exact Hessian computation. The profile shows that the BFGS preconditioner has slight edge; in average, it is faster and more robust that the other ones, so it will be our choice for the rest of this paragraph.

Table 5.1 gives a comparison of PENSDP (i.e., code with Cholesky solver), PEN-PCG(BFGS) (with exact Hessian computation) and PEN-A-PCG(BFGS) (approximate Hessian computation). Given are not only the CPU times in seconds, but also times per one Newton iteration and number of CG steps (when applicable) per one Newton iteration. We have chosen the form of a table (as opposed 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 is the PEN-PCG(BFGS) about as fast as PENSDP, in few cases it is significantly faster. These examples (`theta*` and `qap*`) are typical by a high ratio of n to m. In such situation, the complexity of the solution of the Newton system dominates the complexity of Hessian computation and PCG version of the code is expected to be efficient (see Table 5.1 and the text below). In (almost) all other problems, most time is spent in Hessian computation and thus the solver of the Newton system does not effect the total CPU time. In few problems (`control*`, `truss8`), PEN-PCG(BFGS) was 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-A-PCG(BFGS) results, we see even stronger effect "the higher the ratio n to m, the more efficient code"; in all other examples the code is slower than the other two.

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

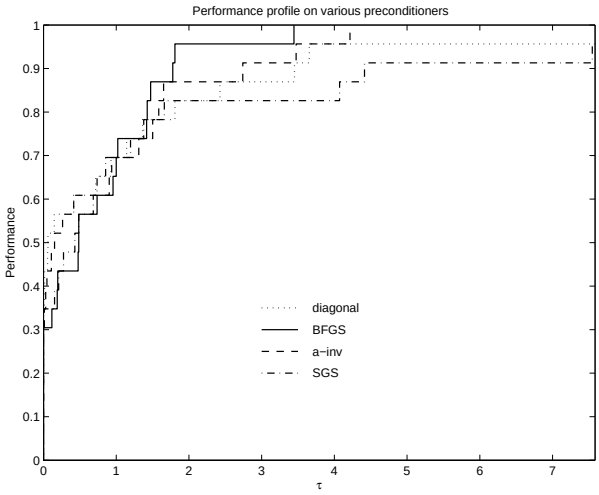

Fig. 6. Performance profile on preconditioners; SDPLIB problems

Table 5.1 compares the CPU time spent in different parts of the algorithm for different types of problems. We have chosen typical representatives of problems with $n \approx m$ (`equalG11`) and $n/m \gg 1$ (`theta4`). For the three codes PENSDP, PEN-PCG(diag) and PEN-A-PCG(BFGS) we show the total CPU time spent in the unconstrained minimization, 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.

5.2. HM collection

Table 5.2 lists a selection of large-scale problems from the HM collection, together with their dimensions and number of nonzeros in the data matrices.

Again we start the testing with a CPU-time performance profile on preconditioners (Figure 7) and again we see the dominance of the BFGS preconditioner.

The test results are collected in Table 5.2, comparing again PENSDP with PEN-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 GLOPTIPOLY, are typical by high ill-conditioning of the Hessian; 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 few problems with large n/m for which PEN-A-PCG(BFGS) outperforms PEN-PCG(BFGS) and this, in turn, clearly outperforms PENSDP: `cancer_100, cphil*, yalsdp`. These problems are ‘good” in the sense that the PCG algorithm needs, in average, a very low number

Table 1. Results for selected SDPLIB problems. PENSDP–standard code with Cholesky algorithm; PEN-PCG(BFGS)–code with CG algorithm and BFGS preconditioner; PEN-A-PCG(BFGS)–code with CG algorithm, approximate Hessian computations and BFGS preconditioner. CPU times in seconds; CPU/it–time per a Newton iteration; CG/it–average number of CG steps per one Newton iteration. 3.2Ghz Pentium 4 with 1GB DDR400 running Linux.

	dimensions		PENSDP		PEN-PCG(BFGS)			PEN-A-PCG(BFGS)		
problem	n	m	CPU	CPU/it	CPU	CPU/it	CG/it	CPU	CPU/it	CG/it
arch8	174	335	6	0.06	6	0.08	13		failed	
control7	666	105	63	0.79	120	1.71	162	164	2.31	387
control10	1326	150	765	3.38	1807	8.21	433	1961	10.89	723
control11	1596	165	937	5.42	1421	12.69	502	1870	19.68	973
equalG11	801	801	65	3.10	110	1.22	2	223	7.43	7
equalG51	1001	1001	296	3.65	295	6.56	3	438	9.73	3
gpp250-4	250	250	4	0.14	5	0.22	5	8	0.36	6
gpp500-4	501	500	26	0.67	32	1.10	5	51	1.89	5
maxG11	800	800	10	0.43	17	0.54	6	40	1.14	6
maxG32	2000	2000	108	4.70	185	5.97	8	476	14.00	6
maxG51	1000	1000	115	2.45	204	3.82	4	343	6.86	4
mcp250-1	250	250	1	0.03	2	0.06	4	2	0.06	4
mcp500-1	500	500	5	0.16	7	0.18	5	11	0.30	4
qap9	748	82	3	0.09	4	0.07	7	4	0.06	30
qap10	1021	101	8	0.22	11	0.16	7	8	0.10	20
qpG51	1000	2000	172	5.93	241	7.53	3	400	11.76	3
ss30	132	426	10	0.28	13	0.32	4	7	0.18	4
theta3	1106	150	9	0.26	7	0.23	4	4	0.08	4
theta4	1949	200	43	1.02	24	0.85	6	11	0.22	7
theta5	3028	250	93	3.32	43	2.29	5	16	0.39	6
theta6	4375	300	366	9.15	139	4.80	5	41	0.67	8
thetaG11	2401	801	134	2.68	312	2.35	6	939	7.83	51
truss8	496	628	6	0.13	55	0.70	157	11	0.22	7

Table 2. 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-PCG(BFGS)				PEN-A-PCG(BFGS)		
problem	CPU	f+g	hess	chol	CPU	f+g	hess	CG	CPU	f+g	CG
theta4	44.2	1.1	11.1	31.8	20.7	2.8	12.7	5.0	10.4	10.0	0.2
equalG11	43.7	28.5	12.5	1.7	98.1	77.2	18.5	2.4	184.3	182.4	0.9

of iterations per one 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.

5.3. TRUSS collection

Unlike the previous two sets of problems collecting examples 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 diagonal preconditioner is the winner, closely followed by SGS; BFGS is the poorest one now.

Table 3. 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
neu3g	463	8 007	106 952	2
rabmo	6 827	5 004	60 287	2
rose13	106	2 379	5 564	2
rose15	138	3 860	9 182	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 results of our testing (see Table 5.3) 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 spent in Hessian evaluation. Indeed, for both PENSDP and PEN-PCG(diag) the CPU time per one Newton step is about the same in all examples. These problems are ill-conditioned; 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.

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 $O(10n^2)$, we may expect significant speed-up of the resulting algorithm. This is indeed the case for 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 [24].

The dimensions of the problems are shown in Table 5.4; 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

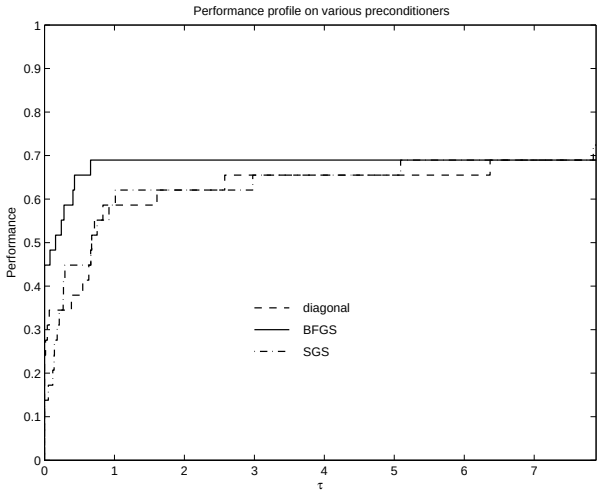

Fig. 7. Performance profile on preconditioners; HM problems

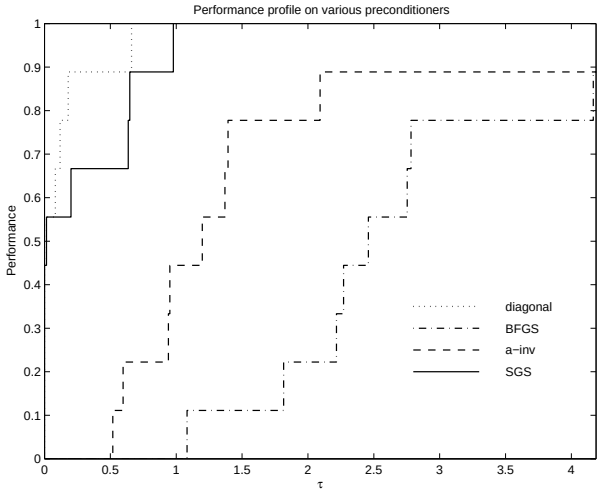

Fig. 8. Performance profile on preconditioners; TRUSS problems

to store and factorize a full matrix of dimension 130 000 by 130 000. On the other hand, PEN-A-PCG(BFGS), being effectively a first-order code, has just modest memory requirements and allows us to solve these large problems within a very reasonable time.

As always, we first show a CPU-time based performance profile on the preconditioners; see Figure 9. This time, diagonal preconditioner is a clear winner.

We further present a CPU-based performance profile on the different codes, PENSDP, PEN-PCG(diag), PEN-A-PCG(none) and PEN-A-PCG(BFGS); see Figure 9. We can

Table 4. Results for selected HM-problems. PENSDP–standard code with Cholesky algorithm; PEN-PCG(BFGS)–code with CG algorithm and BFGS preconditioner; PEN-A-PCG(BFGS)–code with CG algorithm, approximate Hessian computations and BFGS preconditioner. CPU times in seconds; CPU/it–time per a Newton iteration; CG/it–average number of CG steps per one Newton iteration. 3.2Ghz Pentium 4 with 1GB DDR400 running Linux; time limit 20 000 sec.

	PENSDP		PEN-PCG(BFGS)			PEN-A-PCG(BFGS)		
problem	CPU	CPU/it	CPU	CPU/it	CG/it	CPU	CPU/it	CG/it
cancer_100	5863	108.57	481	18.50	5	91	3.37	5
checker_1.5	1424	20.34	807	23.74	5	2001	57.17	5
cnhil10	failed		1775	13.05	44		timed out	
cnhil8	88	1.13	162	1.11	33	272	0.98	113
cphil10	334	18.56	205	9.32	14	15	0.75	16
cphil12	memory		1474	70.19	16	106	2.26	10
foot	1803	38.36	2737	57.02	3	5429	106.45	5
G40_mb	1118	34.94	1787	48.30	5	3296	89.08	5
G40mc	964	26.05	1349	26.98	4	2511	50.22	4
G55mc	8490	217.69	11731	325.86	4		timed out	
G59mc	14925	414.58	18008	473.89	5		timed out	
hand	355	8.66	501	12.85	5		timed out	
neosfbr20	3638	63.82	3552	55.50	91	1270	18.96	135
neu1	1252	10.89		timed out			timed out	
neu1g	560	10.77	1158	25.73	349	1015	24.17	546
neu2c	2144	27.49		timed out			timed out	
neu2	1239	10.87		timed out			timed out	
neu2g	2130	10.76		timed out			timed out	
neu3	15182	108.44		timed out			timed out	
neu3g	33814	146.38		timed out			timed out	
rabmo	1436	18.18		timed out			timed out	
rose13	140	1.89	2269	5.64	170	360	2.06	209
rose15	13358	7.17		timed out			timed out	
taha1a	2578	24.79	6343	43.74	434	8948	62.57	499
taha1b	12192	69.67		timed out			timed out	
yalsdp	1421	38.41	1182	30.31	9	39	1.08	10

see dominance of the codes based on the approximate Hessian calculation. From the rest, PEN-A-PCG(none) is clearly faster than PENSDP.

Table 5.4 collects the results. As expected, larger problems are not solvable by the second-order codes PENSDP and PEN-PCG(SGS), due to memory limitations. They can be, on the other hand, easily solved by PEN-A-PCG(BFGS). Note that the largest problem from the collection, `theta162`, needed just 614 MB of memory. But not only memory is the limitation of PENSDP. We can see huge speed-up in CPU time going from PENSDP to PEN-PCG(SGS) and further to PEN-A-PCG(BFGS), in all examples.

As to our knowledge, aside from the code described in [22], the only available code capable of solving problems of this size is SDPLR by Burer and Monteiro ([5]). 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 5.4 thus also contains results obtained by SDPLR; the stopping criterium 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. This is due to a very high number of L-BFGS iterations needed.

Table 5. Results for selected TRUSS problems. PENSDP–standard code with Cholesky algorithm; PEN-PCG(diag)–code with CG algorithm and diagonal preconditioner. CPU times in seconds; CPU/it–time per a Newton iteration; CG/it–average number of CG steps per one Newton iteration. 3.2Ghz Pentium 4 with 1GB DDR400 running Linux; time limit 20 000 sec.

			PENSDP		PEN-PCG(diag)		
problem	n	m	CPU	CPU/it	CPU	CPU/it	CG/it
buck3	544	1 186	52	0.29	80	0.37	19
buck4	1 200	2 546	219	1.56	477	2.62	66
buck5	3 280	6 802	3445	15.66	6485	20.14	34
trto3	544	866	11	0.16	18	0.19	16
trto4	1 200	1 874	74	0.80	117	0.88	22
trto5	3 280	5 042	1348	8.93	1249	8.92	9
vibra3	544	1 186	34	0.29	67	0.34	11
vibra4	1 200	2 546	169	1.54	289	1.88	10
vibra5	3 280	6 802	2156	15.85	4786	18.77	20
shmup3	420	2 642	236	3.23	317	3.82	4
shmup4	800	4 962	1184	16.22	1910	20.32	5
shmup5	1 800	11 042	9494	85.53		timed out	

Table 6. 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

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).

For the purpose of this testing, we have adopted additional stopping rules for our algorithm. So far (in all tests in the previous sections), Algorithm 1 was terminated when both inequalities in (10) were satisfied. While $\varepsilon = 10^{-7}$ in the standard code

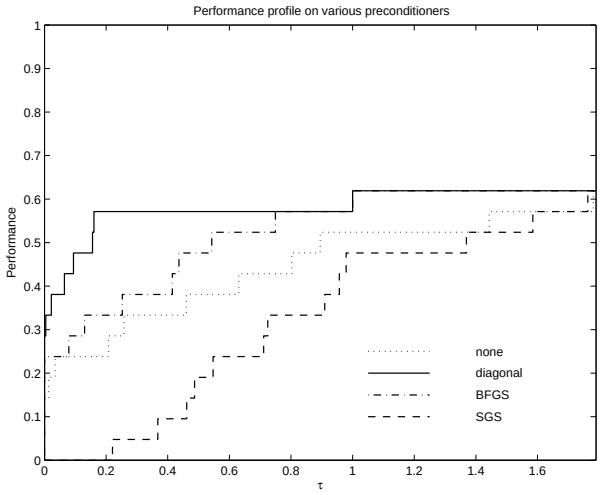

Fig. 9. Performance profile on preconditioners; TOH problems

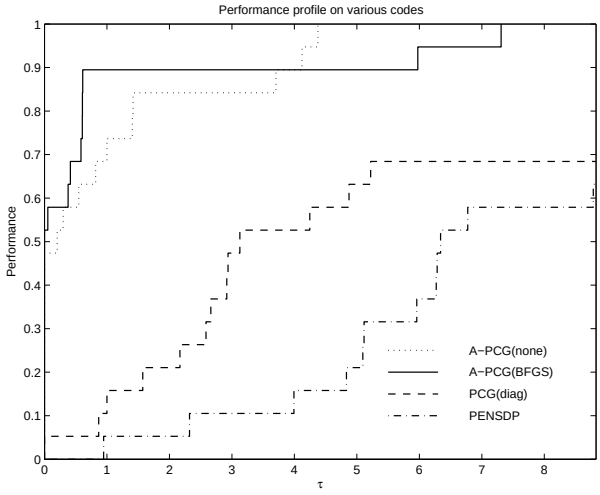

Fig. 10. Performance profile on codes; TOH problems

PENSDP, for the tests of the (A-)PCG version of the code, we set $\varepsilon = 10^{-4}$. In order to be able to exactly examine the effect of required accuracy on the algorithm behavior, we have additionally adopted the DIMACS criteria [14]. To define these criteria, we

Table 7. Results for selected TOH problems. PENSDP–standard code with Cholesky algorithm; PEN-PCG(diag)–code with CG algorithm and diagonal preconditioner; PEN-A-PCG(BFGS)–code with CG algorithm, approximate Hessian computations and BFGS preconditioner. CPU times in seconds; CPU/it–time per a Newton iteration; CG/it–average number of CG steps per one Newton iteration. Sun UltraSparc IIICu 1200MHz with 4GB RAM; time limit 100 000 sec.

	PENSDP		PEN-PCG(diag)			PEN-A-PCG(BFGS)			SDPLR	
problem	CPU	CPU/it	CPU	CPU/it	CG/it	CPU	CPU/it	CG/it	CPU	iter
ham_7_5_6	104	3.2	19	0.7	2	4	0.1	2	1	113
ham_9_8	266	9.8	138	5.3	3	210	4.7	2	46	222
ham_8_3_4	71264	2036.1	2983	80.1	2	104	2.7	1	21	195
ham_9_5_6	memory			memory		1984	37.4	1	71	102
theta42	3978	104.6	391	9.3	6	51	1.2	7	393	11548
theta6	1719	42.9	197	5.3	5	108	2.0	6	1221	20781
theta62	51359	1222.8	3779	77.1	6	196	4.3	5	1749	16784
theta8	8994	243.0	783	19.1	6	263	5.3	6	1854	15257
theta82	memory			memory		650	14.4	6	4650	20653
theta10	30610	956.5	6571	126.4	6	492	10.7	6	4636	18814
theta102	memory			memory		1948	47.5	8	12275	29083
theta103	memory			memory		6149	149.9	10	17687	29483
theta104	memory			memory		8400	215.3	7	timed out	
theta12	timed out		14098	223.7	10	843	16.2	5	8081	21338
theta123	memory			memory		11733	266.66	8	timed out	
theta162	memory			memory		50098	927.74	16	timed out	
keller4	3724	66.5	297	6.5	6	52	1.1	9	244	8586
sanr200-0.7	4210	107.9	393	9.1	6	52	1.2	7	405	12139

rewrite our SDP problem (1) as

$$\begin{aligned} &\min_{x\in\mathbb{R}^n} f^T x \\ &\text{subject to} \\ &\qquad \mathcal{C}(x) \preccurlyeq C_0 \end{aligned} \tag{13}$$

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

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

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 "DIMACS version" of the code we will require that (10) is satisfied with $\varepsilon = 10^{-4}$ and, at the same time,

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

In the following we will study the effect of δ_{DIMACS} on the behavior of the algorithm.

We have solved selected examples using the codes PEN-PCG(BFGS) and PEN-A-PCG(BFGS) with several values of δ_{DIMACS}, namely

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

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 criterium for the unconstrained minimization (9), 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 more and more exactly, in order to reach the required accuracy. The decrease of α is based on the required accuracy ε and δ_{DIMACS}. To make this a bit more transparent, we set, for the purpose of testing,

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

Monotone strategy In the nonmonotone version of the code, already the first iterations of the algorithm ran with $\delta_{\mathrm{DIMACS}} = 10^{-1}$ differ from the run with $\delta_{\mathrm{DIMACS}} = 10^{-2}$, 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 δ_{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 δ_{DIMACS} on the behavior of the (A-)PCG code.

Note that for $\delta_{\mathrm{DIMACS}} = 10^{-5}$ both, the monotone and the nonmonotone version coincide.

6.1. Testing PEN-PCG(·)

Here we examine the effect of increasing Hessian ill-conditioning (when decreasing δ_{DIMACS}) on the overall behavior of the code. Table 6.1 presents the results for selected examples. 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.

Table 8. Convergence of PEN-PCG(BFGS) on selected problems using the monotone (mon=Y) and non-monotone (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. 3.2Ghz Pentium 4 with 1GB DDR400 running Linux; time limit 20 000 sec.

δ_{DIMACS}	mon	CPU	Nwt	CG	err_1	err_4	err_5	err_6	objective
theta4									
1.0E-01	Y	33	68	382	2.4E-06	3.3E-03	4.3E-07	6.5E-07	50.32110492
1.0E-03	Y	39	73	573	3.2E-06	4.3E-05	2.9E-06	4.7E-09	50.32122133
1.0E-05	Y	40	74	580	1.7E-06	3.4E-06	3.1E-06	3.1E-11	50.32122195
1.0E-01	N	18	44	113	2.4E-03	5.0E-03	1.6E-03	3.5E-07	50.32130438
1.0E-03	N	24	55	237	2.0E-05	9.4E-05	4.5E-05	1.2E-08	50.32122357
1.0E-05	N	40	74	580	1.7E-06	3.4E-06	3.1E-06	3.1E-11	50.32122195
theta42									
1.0e-01	Y	355	65	333	3.2e-06	1.7e-03	2.8e-07	6.8e-07	23.93165748
1.0e-03	Y	450	71	618	1.8e-06	3.4e-05	5.4e-07	3.2e-09	23.93170806
1.0e-05	Y	521	74	864	1.2e-06	4.6e-06	1.0e-08	9.7e-11	23.93170827
1.0e-01	N	196	40	117	3.2e-03	6.8e-03	9.2e-05	5.4e-06	23.93165748
1.0e-03	N	385	66	459	4.7e-04	2.5e-06	3.6e-06	3.8e-09	23.93170839
1.0e-05	N	521	74	864	1.2e-06	4.6e-06	1.0e-08	9.7e-11	23.93170827
theta6									
1.0e-01	Y	212	74	374	2.0e-06	2.9e-03	3.1e-06	7.1e-07	63.47694156
1.0e-03	Y	265	81	657	1.7e-06	5.4e-05	2.6e-06	4.7e-09	63.47708631
1.0e-05	Y	280	83	738	8.1e-06	1.0e-05	5.9e-07	6.9e-10	63.47708706
1.0e-01	N	141	53	173	3.1e-03	2.6e-03	7.2e-04	5.4e-07	63.47717987
1.0e-03	N	240	81	459	8.9e-04	6.9e-05	4.2e-05	3.0e-10	63.47708743
1.0e-05	N	280	83	738	8.1e-06	1.0e-05	5.9e-07	6.9e-10	63.47708706
cancer-100									
1.0e-01	Y	1165	64	489	1.4e-04	8.5e-03	6.6e-06	7.7e-07	27623.55121
1.0e-03	Y	1269	69	548	6.3e-04	1.1e-05	4.6e-06	4.9e-11	27623.61311
1.0e-05	Y	2354	84	2260	1.9e-04	1.6e-07	5.5e-07	5.7e-11	27623.32523
1.0e-01	N	533	32	136	5.5e-01	4.2e-03	8.2e-04	1.1e-06	27625.63374
1.0e-03	N	903	49	389	2.5e-03	2.3e-05	5.5e-06	4.9e-09	27623.61346
1.0e-05	N	2354	84	2260	1.9e-04	1.6e-07	5.5e-07	5.7e-11	27623.32523
keller4									
1.0e-01	Y	283	67	482	9.4e-06	4.2e-04	5.2e-06	1.3e-07	14.01223772
1.0e-03	Y	302	69	546	4.0e-06	3.5e-05	3.3e-06	4.1e-10	14.01224167
1.0e-05	Y	307	70	560	4.2e-06	3.1e-06	2.5e-06	5.6e-10	14.01224167
1.0e-01	N	152	42	151	2.8e-03	6.2e-03	6.0e-05	9.6e-07	14.01229001
1.0e-03	N	302	69	546	4.0e-06	3.5e-05	3.3e-06	4.1e-10	14.01224167
1.0e-05	N	307	70	560	4.2e-06	3.1e-06	2.5e-06	5.6e-10	14.01224167
hamming-9-8									
1.0e-01	Y	92	65	329	2.8e-06	2.1e-04	6.6e-06	2.5e-07	223.9998864
1.0e-03	Y	96	67	359	5.3e-07	3.8e-05	1.1e-06	1.0e-08	223.9999954
1.0e-05	Y	99	68	362	8.3e-07	5.9e-06	1.5e-06	5.7e-10	223.9999997
1.0e-01	N	68	47	195	1.5e-03	3.2e-03	2.1e-03	7.4e-06	224.0039475
1.0e-03	N	86	55	410	8.1e-06	2.4e-05	8.1e-06	7.0e-09	224.0000032
1.0e-05	N	99	68	362	8.3e-07	5.9e-06	1.5e-06	5.7e-10	223.9999997

From the table we can conclude two main things: the increased accuracy does not really cause problems (up to exceptions—the `cancer-100` problem); and the non-monotone strategy is clearly advisable in practice. To reach the accuracy of 10^{-5}, one needs about 2–4 times more CG steps than for 10^{-1}. Only for the `cancer-100` problem, the highest accuracy causes problems but, still, the code can reach it.

Note also that the actual accuracy is often 1-2 digits better than the one required, particularly for $\delta_{\mathrm{DIMACS}} = 10^{-1}$. This is also due to the fact that the primal stopping criterium (10) with $\varepsilon = 10^{-4}$ is still in power.

6.2. Testing PEN-A-PCG(·)

Here not only increasing Hessian ill-conditioning but also limited accuracy of the finite difference formula effect the codes behavior when increasing the accuracy. Table 6.2 summarizes the results for the same examples as in the previous section, this time with the code PEN-A-PCG(BFGS), i.e., with the approximate Hessian calculation.

Compared to the version with exact Hessian (Table 6.1) we see a much larger number of CG iterations in all examples but `hamming-9-8`. This goes on account of the reduced accuracy of the approximate formula. Still in all cases the code is able to find the solution within the required accuracy. Furthermore, the speed-up caused by the use of approximate Hessian is such that, for given accuracy, the PEN-A-PCG code is always absolutely faster than PEN-PCG, notwithstanding the higher number of PCG iterations.

7. Conclusion and outlook

In the framework of a modified barrier method for linear SDP problems, we propose to use iterative solvers form 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, replacing the exact Hessian-vector product by a finite difference formula using just the gradient, 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. A new promising direction was shown in a recent article by Monteiro, O'Neal and Nemirovski [16] who propose an adaptive preconditioner particularly suitable for extremely ill-conditioned problems. The detailed testing of this preconditioner within PENNON will be performed in a future report. However, preliminary tests were quite promising and so we conclude our article with their results.

The MON (from Monteiro-O'Neal-Nemirovski) preconditioner is constructed adaptively during the CG process. The preconditioning matrix (initialized as identity) is updated by rank-one updates. The number of these updates typically does not exceed the dimension of the matrix (and can be indirectly controlled by a parameter). The rank-one updates are computed from Hessian-vector products, so this method fits perfectly into our framework. To test our first Matlab implementation of this method, we saved the

Table 9. Same as Table 6.1 but for PEN-A-PCG(BFGS)

δ_{DIMACS}	mon	CPU	Nwt	CG	err_1	err_4	err_5	err_6	objective
theta4									
1.0e-01	Y	32	80	1410	3.8e-06	3.3e-03	2.6e-08	6.4e-07	50.32110500
1.0e-03	Y	47	85	2141	3.3e-06	4.1e-05	1.1e-06	5.0e-09	50.32122130
1.0e-05	Y	48	86	2190	4.2e-06	5.3e-06	2.8e-06	2.5e-10	50.32122193
1.0e-01	N	11	52	380	5.0e-03	9.8e-03	7.3e-04	1.5e-06	50.32152530
1.0e-03	N	18	66	712	6.5e-05	7.1e-05	5.2e-05	4.0e-05	50.32122697
1.0e-05	N	48	86	2190	4.2e-06	5.3e-06	2.8e-06	2.5e-10	50.32122193
theta42									
1.0e-01	Y	70	74	1598	1.8e-06	1.7e-03	4.1e-07	6.7e-07	23.93165793
1.0e-03	Y	129	80	3118	1.2e-06	2.2e-05	1.4e-06	3.6e-09	23.93170804
1.0e-05	Y	163	82	3985	6.0e-06	4.5e-06	6.4e-06	3.1e-10	23.93170826
1.0e-01	N	16	41	302	3.6e-03	8.5e-03	6.0e-04	2.7e-06	23.93190516
1.0e-03	N	98	59	2358	3.2e-04	1.6e-06	3.1e-04	2.4e-09	23.93170841
1.0e-05	N	163	82	3985	6.0e-06	4.5e-06	6.4e-06	3.1e-10	23.93170826
theta6									
1.0e-01	Y	96	82	1493	5.1e-06	2.9e-03	1.7e-06	7.2e-07	63.47694000
1.0e-03	Y	134	87	2142	9.7e-06	6.0e-05	3.0e-07	5.4e-09	63.47708625
1.0e-05	Y	194	90	3205	7.7e-06	2.3e-06	3.5e-06	2.8e-11	63.47708718
1.0e-01	N	45	80	552	7.8e-03	2.2e-03	3.3e-04	1.3e-07	63.47723926
1.0e-03	N	70	69	1056	3.2e-05	6.6e-05	3.4e-05	2.0e-08	63.47709031
1.0e-05	N	194	90	3205	7.7e-06	2.3e-06	3.5e-06	2.8e-11	63.47708718
cancer-100									
1.0e-01	Y	322	66	764	9.3e-05	8.5e-03	5.1e-06	7.6e-07	27623.46835
1.0e-03	Y	350	71	827	9.6e-04	1.2e-05	5.8e-06	1.7e-09	27623.52935
1.0e-05	Y	2123	105	6022	1.7e-04	5.5e-10	5.8e-07	4.5e-11	27623.34520
1.0e-01	N	88	32	149	3.6e-01	1.6e-03	4.3e-04	1.7e-07	27625.66861
1.0e-03	N	204	49	451	5.4e-03	6.1e-07	1.5e-05	9.9e-09	27623.36620
1.0e-05	N	2123	105	6022	1.7e-04	5.5e-10	5.8e-07	4.5e-11	27623.34520
keller4									
1.0e-01	Y	70	78	2416	6.2e-06	4.3e-04	1.4e-06	1.3e-07	14.01223768
1.0e-03	Y	80	80	2751	4.1e-06	3.7e-05	2.7e-06	2.3e-09	14.01224161
1.0e-05	Y	84	81	2903	3.4e-06	3.3e-06	6.6e-07	3.1e-10	14.01224167
1.0e-01	N	13	48	366	6.4e-03	6.5e-03	1.3e-04	1.7e-06	14.01239670
1.0e-03	N	39	61	1266	1.5e-05	6.8e-06	1.5e-05	3.1e-08	14.01224257
1.0e-05	N	84	81	2903	3.4e-06	3.3e-06	6.6e-07	3.1e-10	14.01224167
hamming-9-8									
1.0e-01	Y	62	55	101	3.2e-06	1.8e-04	2.2e-06	2.5e-07	223.9998839
1.0e-03	Y	63	56	103	1.4e-06	2.5e-05	4.2e-07	7.3e-09	223.9999967
1.0e-05	Y	65	57	109	5.3e-07	8.0e-06	2.0e-07	1.8e-10	223.9999999
1.0e-01	N	47	44	65	1.0e-05	1.3e-03	3.7e-06	6.9e-07	224.0003052
1.0e-03	N	49	45	68	6.2e-06	6.3e-06	1.5e-06	6.6e-08	224.0000295
1.0e-05	N	65	57	109	5.3e-07	8.0e-06	2.0e-07	1.8e-10	223.9999999

Hessians of several problems at the initial point of Algorithm 1 and at the optimal point (within our optimality criteria). We present comparison of the MON preconditioner with the two general preconditioners, diagonal and SGS and with the plain CG method. We have chosen the same examples as in Section 4 where we show the spectrum of the matrices: `control3` and `theta2`. The results are presented in Figure 11; we plot the logarithm of the energetic norm of the error, scaled by the norm of the exact solution. Looking at the `control3` problem at the initial point, we see that the diagonal and SGS preconditioners are very efficient. The MON method has a “slow start”; these are the iterations where the preconditioner is being constructed. Once it is ready, the conver-

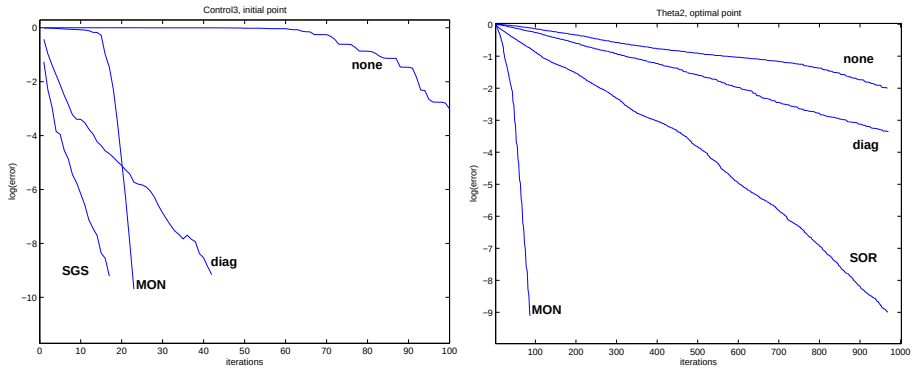

Fig. 11. Comparison of MON with standard preconditioners for problems `control3` at the initial point (left) and `theta2` at the optimal point (right)

gence rate becomes very high. However, for the purpose of low-precision computation, diagonal and SGS are preferable. A completely different picture is seen at the `theta2` problem at the optimal point. The standard preconditioners, although better than the plain CG method, are not particularly efficient. On the other hand, the MON preconditioner accelerates the CG method significantly. The (preliminary) conclusion is: if the standard preconditioners are efficient, one should prefer them to MON—the preconditioning matrix is known a priori and thus the effect is seen from the first iterations. On the other hand, when the standard methods become inefficient, one should switch to the MON preconditioner. Within an optimization algorithm (like Algorithm 1), one can think of a hybrid strategy, starting with SGS and switching to MON when the condition number of the Hessian becomes too high. Naturally, it remains a question of the true efficiency of the MON preconditioner, taking into account its actual complexity within an optimization code (the theoretical complexity may, as usual, be too pessimistic). This question is to be answered in a future article.

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.

References

1. F. Alizadeh, J.-P. A. Haeberly, and M. L. Overton. Primal–dual interior–point methods for semidefinite programming : Convergence rates, stability and numerical results. *SIAM Journal on Optimization*, 8:746–768, 1998.
2. S. J. Benson, Y. Ye, and X. Zhang. Solving large-scale sparse semidefinite programs for combinatorial optimization. *SIAM Journal on Optimization*, 10:443–462, 2000.
3. M. Benzi, J. K. Cullum, and M. Tůma. Robust approximate inverse preconditioning for the conjugate gradient method. *SIAM J. Sci. Comput.*, 22:1318–1332, 2000.
4. B. Borchers. 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/`.
5. S. Burer and R.D.C. Monteiro. A nonlinear programming algorithm for solving semidefinite programs via low-rank factorization. *Mathematical Programming (series B)*,, 95(2):329–357, 2003.

6. C. Choi and Y. Ye. 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.
7. E. D. Dolan and J.J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.
8. C. Helmberg, F. Rendl, R. J. Vanderbei, and H. Wolkowicz. An interior–point method for semidefinite programming. *SIAM Journal on Optimization*, 6:342–361, 1996.
9. M. Kočvara, F. Leibfritz, M. Stingl, and D. Henrion. A nonlinear SDP algorithm for static output feedback problems in COMPlib. LAAS-CNRS research report no. 04508, LAAS, Toulouse, 2004.
10. M. Kočvara and M. Stingl. PENNON—a code for convex nonlinear and semidefinite programming. *Optimization Methods and Software*, 18:317–333, 2003.
11. M. Kočvara and M. Stingl. Solving nonconvex SDP problems of structural optimization with stability control. *Optimization Methods and Software*, 19:595–609, 2004.
12. C.-J. Lin and R. Saigal. An incomplete cholesky factorization for dense matrices. *Applied Numerical Mathematics*, BIT:536–558.
13. H. Mittelmann. Benchmarks for optimization software; as of January 5, 2005. Available at `http://plato.la.asu.edu/bench.html`.
14. H. D. Mittelmann. An independent benchmarking of SDP and SOCP solvers. *Math. Prog.*, 95:407–430, 2003.
15. R. D. C. Monteiro. Primal–dual path-following algorithms for semidefinite programming. *SIAM Journal on Optimization*, 7:663–678, 1997.
16. R.D.C. Monteiro, J.W. O'Neal, and A. Nemirovski. A new conjugate gradient algorithm incorporating adaptive ellipsoid preconditioning. Report, School of ISyE, Georgia Tech, USA, October 2004.
17. J. L. Morales and J. Nocedal. Automatic preconditioning by limited memory quasi-Newton updating. *SIAM Journal on Optimization*, 10:1079–1096, 2000.
18. Jorge Nocedal and Stephen Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, New York, 1999.
19. R. Polyak. Modified barrier functions: Theory and methods. *Mathematical Programming*, 54:177–222, 1992.
20. M. Stingl. *On the Solution of Nonlinear Semidefinite Programs by Augmented Lagrangian Methods*. PhD thesis, Institute of Applied Mathematics II, Friedrich-Alexander University of Erlangen-Nuremberg, in preparation.
21. J. Sturm. *Primal-Dual Interior Point Approach to Semidefinite Programming*. PhD thesis, Tinbergen Institute Research Series vol. 156, Thesis Publishers, Amsterdam, The Netherlands. Available at `http://members.tripodnet.nl/SeDuMi/sturm/papers/thesisSTURM.ps.gz`.
22. K. C. Toh. Solving large scale semidefinite programs via an iterative solver on the augmented systems. *SIAM J. Optim.*, 14:670–698.
23. K. C. Toh and M. Kojima. Solving some large scale semidefinite programs via the conjugate residual method. *SIAM J. Optim.*, 12:669–691.
24. M. Trick, V. Chvátal, W. Cook, D. Johnson, C. McGeoch, and R. Trajan. The second DIMACS implementation challenge: NP hard problems: Maximum clique, graph coloring, and satisfiability. Technical report, Rutgers University. Available at `http://dimacs.rutgers.edu/Challenges/`.
25. S.-L. Zhang, K. Nakata, and M. Kojima. Incomplete orthogonalization preconditioners for solving large and dense linear systems which arise from semidefinite programming. *Applied Numerical Mathematics*, 41:235–245.