Optimization Methods and Software 18(3):317-333, 2003

PENNON
A Code for Convex Nonlinear and Semidefinite Programming

Michal Kočvara* Michael Stingl**

Abstract

We introduce a computer program PENNON for the solution of problems of convex Nonlinear and Semidefinite Programming (NLP-SDP). The algorithm used in PENNON is a generalized version of the Augmented Lagrangian method, originally introduced by Ben-Tal and Zibulevsky for convex NLP problems. We present generalization of this algorithm to convex NLP-SDP problems, as implemented in PENNON and details of its implementation. The code can also solve second-order conic programming (SOCP) problems, as well as problems with a mixture of SDP, SOCP and NLP constraints. Results of extensive numerical tests and comparison with other optimization codes are presented. The test examples show that PENNON is particularly suitable for large sparse problems.

1 Introduction

A class of iterative methods for convex nonlinear programming problems, introduced by Ben-Tal and Zibulevsky [3] and named PBM, proved to be very efficient for solving large-scale nonlinear programming (NLP) problems, in particular those arising from optimization of mechanical structures. The framework of the algorithm is given by the augmented Lagrangian method; the difference to the classic algorithm is in the definition of the augmented Lagrangian function. This is defined using a special penalty/barrier function satisfying certain properties; this definition guarantees good behavior of the Newton method when minimizing the augmented Lagrangian function.

The PBM algorithm has been recently generalized to convex semidefinite programming problems [10]. The idea is to use the PBM penalty function to construct another function that penalizes matrix inequality constraints. The efficiency of this approach is based on a special choice of the penalty function for matrix inequalities. This special choice affects the complexity of the algorithm, in particular the complexity of Hessian assembling, which is the bottleneck of all SDP codes working with second-order information. A slightly different approach

*Institute of Applied Mathematics, University of Erlangen, Martensstr. 3, 91058 Erlangen, Germany (kocvara@am.uni-erlangen.de). On leave from the Czech Academy of Sciences.

**Institute of Applied Mathematics, University of Erlangen, Martensstr. 3, 91058 Erlangen, Germany (stingl@am.uni-erlangen.de).

to the generalization of the PBM algorithm to SDP problems can be found in [14, 19, 20].

The aim of this paper is to introduce a code PENNON (PENalty method for NONlinear optimization) as a general tool for solving convex nonlinear optimization problems with NLP and SDP constraints and their combination. While in the NLP branch of the code we basically adapt the PBM method by Ben-Tal and Zibulevsky, the SDP part is based on ideas mentioned above and presented in [10]. The code is also capable of solving Second Order Conic Problems (SOCP), by solving a sequence of NLP approximations.

In Section 2 we define a general NLP-SDP problem and introduce the algorithm for its solution. In the next section, we give details of the algorithm, as implemented in PENNON. In the following section we present an NLP approximation of the SOCP problems, suitable for the algorithm. The last section contains results of extensive numerical tests.

We use the following notation: $\mathbb{S}^m$ is a space of all real symmetric matrices of order m, $A \succcurlyeq 0$ ($A \preccurlyeq 0$) means that $A \in \mathbb{S}^m$ is positive (negative) semidefinite, $A \circ B$ denotes the Hadamard (component-wise) product of matrices $A, B \in \mathbb{R}^{n\times m}$. The space $\mathbb{S}^m$ is equipped with the inner product $\langle A, B\rangle_{\mathbb{S}^m} = tr(AB)$. Let $\mathcal{A} : \mathbb{R}^n \to \mathbb{S}^m$ and $\Phi : \mathbb{S}^m \to \mathbb{S}^m$ be two matrix operators; for $B \in \mathbb{S}^m$ we denote by $D_{\mathcal{A}}\Phi(\mathcal{A}(x); B)$ the directional derivative of Φ at $\mathcal{A}(x)$ (for a fixed x) in the direction B.

2 The NLP-SDP problem and the algorithm

Our goal is to solve optimization problems with nonlinear objective subject to (linear and nonlinear) vector and matrix inequalities as constraints:

$$\begin{aligned} &\min_{x\in\mathbb{R}^n} f(x) \\ &\text{s.t.} \qquad g_i(x) \le 0, \qquad i = 1, \dots, m_g \\ &\qquad\quad \mathcal{A}(x) \preccurlyeq 0\,. \end{aligned} \tag{NLP-SDP}$$

Here f and g_i are convex C^2 functions from $\mathbb{R}^n$ to $\mathbb{R}$ and $\mathcal{A} : \mathbb{R}^n \to \mathbb{S}^{m_A}$ is a convex matrix operator. To simplify the presentation, we only consider inequality constraints. The equality constraints are, in the currents version of the code, reformulated through two inequality constraints.

The method is based on a choice of penalty/barrier functions $\varphi_g : \mathbb{R} \to \mathbb{R}$ and $\Phi_P : \mathbb{S}^{m_A} \to \mathbb{S}^{m_A}$ that penalize the inequality constraints. The matrix penalty function Φ_P is defined by means of another one-dimensional function $\varphi_A : \mathbb{R} \to \mathbb{R}$ as follows. Let $A = S^T \Lambda S$, where $\Lambda = \operatorname{diag}\left(\lambda_1, \lambda_2, \dots, \lambda_{m_A}\right)^T$, be an eigenvalue decomposition of a matrix A. Using φ_A, we define $\Phi_P : \mathbb{S}^m \to \mathbb{S}^m$ as

$$\Phi_P : A \longmapsto S^T \begin{pmatrix} P\varphi_A\left(\frac{\lambda_1}{P}\right) & 0 & \dots & 0 \\ 0 & P\varphi_A\left(\frac{\lambda_2}{P}\right) & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & P\varphi_A\left(\frac{\lambda_{m_A}}{P}\right) \end{pmatrix} S\,, \tag{1}$$

where $P > 0$ is a given number.

Let $\varphi_g : \mathbb{R} \to \mathbb{R}$ and $\varphi_A : \mathbb{R} \to \mathbb{R}$ have the following properties:

(φ_0) φ strictly convex, strictly monotone increasing and C^2,

(φ_1) $\mathrm{dom}\varphi = (-\infty, b)$ with $0 < b \leq \infty$,

(φ_2) $\varphi(0) = 0$,

(φ_3) $\varphi'(0) = 1$,

(φ_4) $\lim_{t \to b} \varphi'(t) = \infty$,

(φ_5) $\lim_{t \to -\infty} \varphi'(t) = 0$.

From these properties it follows that for any $p_i > 0,\ i = 1, \ldots, m_g$, and $P > 0$ we have

$$g_i(x) \leq 0 \iff p_i \varphi_g(g_i(x)/p_i) \leq 0, \quad i = 1, \ldots, m\ ,$$

and

$$\mathcal{A}(x) \preccurlyeq 0 \Longleftrightarrow \Phi_P(\mathcal{A}(x)) \preccurlyeq 0$$

which means that, for any $p_i > 0$ and $P > 0$, problem (NLP-SDP) has the same solution as the following "augmented" problem

$$\begin{aligned} &\min_{x \in \mathbb{R}^n} f(x) \\ &\text{s.t.} \qquad p_i \varphi_g(g_i(x)/p_i) \leq 0, \qquad i = 1, \ldots, m_g \\ &\qquad\qquad \Phi_P(\mathcal{A}(x)) \preccurlyeq 0\,. \end{aligned} \tag{NLP-SDP$_\phi$}$$

The Lagrangian of (NLP-SDP$_\phi$) can be viewed as a (generalized) augmented Lagrangian of (NLP-SDP):

$$F(x, u, U, p, P) = f(x) + \sum_{i=1}^{m_g} u_i p_i \varphi_g(g_i(x)/p_i) + \langle U, \Phi_P\left(\mathcal{A}(x)\right)\rangle_{\mathbb{S}_{m_A}}\ ; \tag{2}$$

here $u \in \mathbb{R}^{m_g}$ and $U \in \mathbb{S}^{m_A}$ are Lagrangian multipliers associated with the inequality constraints.

The basic algorithm combines ideas of the (exterior) penalty and (interior) barrier methods with the Augmented Lagrangian method.

Algorithm 2.1. *Let x^1 and U^1 be given. Let $p_i^1 > 0,\ i = 1, \ldots, m_g$, and $P^1 > 0$. For $k = 1, 2, \ldots$ repeat till a stopping criterium is reached:*

(i) $\quad x^{k+1} = \arg\min_{x \in \mathbb{R}^n} F(x, u^k, U^k, p^k, P^k)$

(ii) $\quad u_i^{k+1} = u_i^k \varphi_g'(g_i(x^{k+1})/p_i^k), \quad i = 1, \ldots, m_g$

$\qquad U^{k+1} = D_{\mathcal{A}} \Phi_p(\mathcal{A}(x); U^k)$

(iii) $\quad p_i^{k+1} < p_i^k,\ i = 1, \ldots, m_g$

$\qquad P^{k+1} < P^k$.

Algorithm 2.1 was implemented (mainly) in the C programming language and this implementation gave rise to a computer program called PENNON. In the next sections we will discuss details of the algorithm as implemented in PENNON: the choice of the penalty functions, the choice of initial values of x, u, U, p and P, the approximate minimization in step (i) and the update formulas.

3 The choice of penalty functions φ_g and Φ_P

As mentioned in the Introduction, Algorithm 2.1 is a generalization of the PBM method by Ben-Tal and Zibulevsky [3] introduced for convex NLPs. In [3], several choices of function φ satisfying (φ_1)–(φ_5) are presented. The most efficient one (for convex NLP) is the quadratic-logarithmic function defined as

$$\varphi^{\mathrm{ql}}(t) = \begin{cases} c_1 \frac{1}{2} t^2 + c_2 t + c_3 & t \geq r \\ c_4 \log(t - c_5) + c_6 & t < r \end{cases} \tag{3}$$

where $r \in (-1, 1)$ and c_i, $i = 1, \ldots, 6$, is chosen so that (φ_1)–(φ_5) hold. This is the function we use as φ_g in the NLP branch of our code.

The choice of function φ_A (and thus Φ_P) is discussed in detail in [10]. We show that φ^{ql} is not a good choice for the definition of Φ_P from two reasons. First, even if the function φ^{ql} and the operator $\mathcal{A}$ are convex, the penalty function Φ_p defined through the right branch of φ^{ql} is nonmonotone and its composition with a convex nonlinear operator $\mathcal{A}$ may result in a nonconvex function $\Phi_P(\mathcal{A}(x))$. This nonconvexity may obviously bring difficulties to Algorithm 2.1 and requires special treatment.

Second, the general definition (1) of the penalty function Φ_P may lead to a very inefficient algorithm. The (approximate) minimization in step (i) of Algorithm 2.1 is performed by the Newton method. Hence we need to compute the gradient and Hessian of the augmented Lagrangian (2) at each step of the Newton method. This computation may be extremely time consuming. Moreover, even if the data of the problem and the Hessian of the (original) Lagrangian are sparse matrices, the computation of the Hessian to the augmented Lagrangian involves many operations with full matrices, when using the general formula (1). In [10] we have shown that the complexity of Hessian assembling is $O(m_A^4 + m_A^3 n + m_A^2 n^2)$. Unfortunately, even if the constraint matrices $\frac{\partial \mathcal{A}(x)}{\partial x_i}$ are sparse, the complexity formula remains the same.

We avoid the above mentioned drawbacks by a choice of the function φ_A. In particular, we choose a function that allows for a "direct" computation of Φ_P and its first and second derivatives. The function used in our code is the reciprocal barrier function

$$\varphi^{\mathrm{rec}}(t) = \frac{1}{t-1} - 1\,. \tag{4}$$

Theorem 3.1. *Let $\mathcal{A} : \mathbb{R}^n \to \mathbb{S}^m$ be a convex operator. Let further Φ_P^{rec} be a function defined by (1) using φ^{rec}. Then for any $x \in \mathbb{R}^n$ there exists $P > 0$ such that*

$$\Phi_P^{\mathrm{rec}}(\mathcal{A}(x)) = P^2 \mathcal{Z}(x) - PI \tag{5}$$

$$\frac{\partial}{\partial x_i} \Phi_P^{\mathrm{rec}}(\mathcal{A}(x)) = P^2 \mathcal{Z}(x) \frac{\partial \mathcal{A}(x)}{\partial x_i} \mathcal{Z}(x) \tag{6}$$

$$\begin{aligned} \frac{\partial^2}{\partial x_i \partial x_j} \Phi_P^{\mathrm{rec}}(\mathcal{A}(x)) = P^2 \mathcal{Z}(x) \Bigg(& \frac{\partial \mathcal{A}(x)}{\partial x_i} \mathcal{Z}(x) \frac{\partial \mathcal{A}(x)}{\partial x_j} - \frac{\partial^2 \mathcal{A}(x)}{\partial x_i \partial x_j} \\ & + \frac{\partial \mathcal{A}(x)}{\partial x_j} \mathcal{Z}(x) \frac{\partial \mathcal{A}(x)}{\partial x_i} \Bigg) \mathcal{Z}(x) \end{aligned} \tag{7}$$

where

$$\mathcal{Z}(x) = (\mathcal{A}(x) - PI)^{-1}\,.$$

Furthermore, $\Phi_P^{\rm rec}(\mathcal{A}(x))$ *is monotone and convex in* x*.*

Using Theorem 3.1 we can compute the value of $\Phi_P^{\rm rec}$ and its derivatives directly, without the need of eigenvalue decomposition of $\mathcal{A}(x)$. The "direct" formulas (6)–(7) are particularly simple for affine operator

$$\mathcal{A}(x) = A_0 + \sum_{i=1}^{n} x_i A_i \quad \text{with } A_i \in \mathbb{S}^m,\ i = 0, 1, \ldots, n\,,$$

when $\dfrac{\partial \mathcal{A}(x)}{\partial x_i} = A_i$ and $\dfrac{\partial^2 \mathcal{A}(x)}{\partial x_i \partial x_j} = 0$.

The complexity of Hessian assembling, when working with function $\varphi^{\rm rec}$ is $O(m_A^3 n + m_A^2 n^2)$. In contrast to the general approach, for sparse constraint matrices with $O(1)$ entries, the complexity formula reduces to $O(m_A^2 n + n^2)$.

4 Implementation details

4.1 Initialization

As we have seen in Theorem 3.1, our algorithm can start with an arbitrary primal variable $x \in \mathbb{R}^n$. Therefore we simply choose $x^0 = 0$. In many cases the matrix constraint is block structured and we denote the number of blocks by M. Using this the initial values of the multipliers are set to

$$\begin{aligned} U_j^0 &= \tau \mu_j^s I_{m_j}, \qquad j = 1, \ldots, M, \\ u_i^0 &= \tau \mu_i^l, \qquad\quad i = 1, \ldots, m_g, \end{aligned}$$

where I_{m_j} are identity matrices of order m_j,

$$\mu_j^s = m_j^\gamma \max_{1 \le \ell \le n} \frac{1 + \left| \frac{\partial f(x)}{\partial x_j} \right|}{1 + \left\| \frac{\partial \mathcal{A}(x)}{\partial x_\ell} \right\|}\,, \tag{8}$$

$$\mu_i^l = \max_{1 \le \ell \le n} \frac{1 + \left| \frac{\partial f(x)}{\partial x_i} \right|}{1 + \left\| \frac{\partial g(x)}{\partial x_\ell} \right\|}\,. \tag{9}$$

In case we are solving pure SDP problems, we choose

$$\begin{aligned} \tau &= \min\left(1, \frac{1000}{\|\mathrm{grad} F(x^0, u^0, U^0, p^0, P^0)\|}\right), \\ \gamma &= 1. \end{aligned}$$

Otherwise, if we combine NLP and SDP constraints we set $\gamma = 2$ and $\tau = 1$. Furthermore, we calculate $\pi > 0$ so that

$$\lambda_{max}(\mathcal{A}_j(x)) < \pi, \quad j = 1, \ldots, k$$

and set $P^0 = \pi$ and $p^0 = \pi e$ where $e \in \mathbb{R}^{m_g}$ is the vector with ones in all components.

4.2 Unconstrained minimization

The tool used in step (i) of Algorithm 2.1 (approximate unconstrained minimization) is the modified Newton method. In each step we calculate the search direction d by solving the Newton equation and find α_{max} so that the conditions

$$\lambda_{max}(\mathcal{A}_j(x^k + \alpha d)) < p_j^k, \quad j = 1, \ldots, k$$

hold for all $0 < \alpha < \alpha_{max}$.

Optionally we combine the Newton method with a cubic linesearch with safeguard criterion.

4.3 Update of multipliers

First we would like to motivate the multiplier update formula in Algorithm 2.1.

Proposition 4.1. *Let x^{k+1} be the minimizer of the augmented Lagrangian F with respect to x in the k-th iteration. If we choose U^{k+1} and $u_i^{k+1}, i = 1, \ldots, m_g$, as in Algorithm 2.1 we have*

$$L(x^{k+1}, u^{k+1}, U^{k+1}, p^k, P^k) = 0,$$

where L denotes the standard Lagrangian of our initial problem (NLP-SDP).

An outline of the proof is given next. The gradient of F with respect to x reads as

$$\nabla_x F(x,u,U,p,P) = \nabla_x f(x) + \sum_{i=1}^{m_g} u_i \varphi'(g_i(x)/p_i) \nabla_x g_i(x) + \begin{pmatrix} \left\langle U, D_{\mathcal{A}} \Phi_P \left(\mathcal{A}(x); \frac{\partial \mathcal{A}(x)}{\partial x_1} \right) \right\rangle \\ \vdots \\ \left\langle U, D_{\mathcal{A}} \Phi_P \left(\mathcal{A}(x); \frac{\partial \mathcal{A}(x)}{\partial x_n} \right) \right\rangle \end{pmatrix}. \tag{10}$$

It can be shown that (10) can be written as

$$\nabla_x f(x) + \sum_{i=1}^{m_g} u_i \varphi'(g_i(x)/p_i) \nabla_x g_i(x) + \mathcal{A}^* D_{\mathcal{A}} \Phi_P \left(\mathcal{A}(x); U \right),$$

where $\mathcal{A}^*$ denotes the conjugate operator to $\mathcal{A}$. Now, if we define $U^{k+1} := D_{\mathcal{A}} \Phi_P \left(\mathcal{A}(x^k); U^k \right)$ and $u_i^{k+1} := u_i^k \varphi'(g_i(x^k)/p_i^k)$, we immediately see that

$$\nabla_x F(x^{k+1}, u^k, U^k, p^k, P^k) = \nabla_x L(x^{k+1}, u^{k+1}, U^{k+1}, p^k, P^k)$$

and so we get $L(x^{k+1}, u^{k+1}, U^{k+1}, p^k, P^k) = 0$.

For our special choice of the penalty function Φ_P^{rec}, the update of the matrix multiplier can be written as

$$U^{k+1} = (P^k)^2 \mathcal{Z}(x) U^k \mathcal{Z}(x), \tag{11}$$

where $\mathcal{Z}$ was defined in Theorem 3.1.

Numerical tests indicated that big changes in the multipliers should be avoided for two reasons. First, they may lead to a large number of Newton steps in the subsequent iteration. Second, it may happen that already after a few steps, the multipliers become ill-conditioned and the algorithm suffers from numerical troubles. To overcome these difficulties, we do the following:

- SDP multipliers:

  1. Calculate U^{k+1} using the update formula in Algorithm 2.1.
  2. Choose some 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

$$U^{new} = U^k + \lambda_A(U^{k+1} - U^k). \tag{12}$$

- NLP multipliers: For each $i = 1, \ldots, m_g$

1. Calculate u_i^{k+1} using the update formula in Algorithm 2.1.
2. Choose some positive $\mu_g \leq 1$, typically 0.5.
3. Check the inequalities
$$\mu_g < \frac{u_i^{k+1}}{u_i^k} < \frac{1}{\mu_g}.$$
4. If one of the inequalities is violated choose $u_i^{new} = \mu_g$ resp. $u_i^{new} = \frac{1}{\mu_g}$. Otherwise accept u_i^{k+1}.

4.4 Stopping criteria and penalty update

When testing our algorithm on pure SDP and pure NLP problems, we observed that the Newton method needs many steps during the first global iterations. To improve this, we adopted the following strategy ([3]): During the first three iterations we do not update the penalty vector p at all. Furthermore, we stop the unconstrained minimization if $\|\nabla_x F(x, u, U, p, P)\|$ is smaller than some $\alpha_0 > 0$, which is not too small, typically 1.0.

After this kind of "warm start", the penalty vector is updated by some constant factor dependent on the initial penalty parameter π. The penalty update is stopped, if some p_{eps} (typically 10^{-6}) is reached. The stopping criterion for the unconstrained minimization changes to $\|\nabla_x F(x, u, U, p, P)\| \leq \alpha$, where in most cases $\alpha = 0.01$ is a good choice.

If we combine SDP with NLP constraints, the testing indicates that it is much better to update the penalty constraints from the very beginning. Therefore the "warm start" is not performed in this case.

Algorithm 2.1 is stopped if one of the inequalities holds:

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

where ϵ is typically 10^{-7}.

4.5 Sparse linear algebra

Many optimization problems have very sparse data structure and therefore have to be treated by sparse linear algebra routines. Since the sparsity issue is more important in semidefinite programming, we will concentrate on this case. For linear SDP with constraint $\mathcal{A}(x) = \sum x_i A_i \succcurlyeq 0$, we can distinguish three basic types of sparsity:

- $\mathcal{A}(x)$ is a block diagonal matrix with many (small) blocks. This leads to a sparse Hessian of the augmented Lagrangian; see the `mater` examples.
- $\mathcal{A}(x)$ has few (large) blocks and
  - $\mathcal{A}(x)$ is dense but A_i are sparse; this is the case of most SDPLIB examples.
  - $\mathcal{A}(x)$ is sparse; see the `truss` examples.

In our implementation, we use sparse linear algebra routines to perform the following three tasks corresponding to the above three cases:

Cholesky factorization The first task is the factorization of the Hessian. In the initial iteration, we check the sparsity structure of the Hessian and do the following:

- If the fill-in of the Hessian is below 20% , we make use of the fact that the sparsity structure will be the same in each Newton step in all iterations. Therefore we create a symbolic pattern of the Hessian and store it. Then we factorize the Hessian by the sparse Cholesky solver of Ng and Peyton [15], which is very efficient for sparse problems with constant sparsity structure.
- Otherwise, if the Hessian is dense, we use the Cholesky solver from LAPACK which, in its newest version, is very robust even for small pivots.

Construction of the Hessian In each Newton step, the Hessian of the augmented Lagrangian has to be calculated. The complexity of this task can be drastically reduced if we make use of sparse structures of the constraint matrices $\mathcal{A}_j(x)$ and the corresponding partial derivatives $\frac{\partial \mathcal{A}_j(x)}{\partial x_i}$. Since there is a great variety of different sparsity types, we refer to the paper by Fujisawa, Kojima and Nakata on exploiting sparsity in semidefinite programming [8], where one can find the ideas we follow in our implementation.

Fast inverse computation of sparse matrices The third situation concerns the case when $A := \mathcal{A}(x)$ is a sparse matrix. When using the reciprocal penalty function φ^{rec}, we have to compute expressions of type

$$(A - I)^{-1} A_i (A - I)^{-1} A_j (A - I)^{-1}.$$

Note that we never need the inverse $(A-I)^{-1}$ alone, but always its multiplication with a sparse matrix, say M. Assume that not only A but also its Cholesky factor is sparse. Then, obviously, the Cholesky factor of $(A - I)$ will also be

sparse. Denote this factor by L. The i-th column of $Z := (A-I)^{-1}M$ can then be computed as

$$Z^i = (L^{-1})^T L^{-1} M^i,\ i = 1, \ldots, n.$$

If κ is the number of nonzeros in L, then the complexity of computing Z through Cholesky factorization is $O(n^2\kappa)$, compared to $O(n^3)$ when working with the explicite inverse of $(A-I)$ (a full matrix) and its multiplication by A_k.

5 Extensions

5.1 SOCP problems

Let us recall that Algorithm 2.1 is defined for general problems with combination of NLP and SDP constraints. It can be thus used, for instance, for solution of Second Order Conic Programming (SOCP) problems combined with SDP constraints, i.e., problems of the type

$$\begin{aligned} \min_{x\in\mathbb{R}^n}\ & b^T x \\ \text{s.t.}\quad & \mathcal{A}(x) \preccurlyeq 0 \\ & A^q x - c^q \leq_q 0 \\ & A^l x - c^l \leq 0 \end{aligned}$$

where $b \in \mathbb{R}^n$, $\mathcal{A} : \mathbb{R}^n \to \mathbb{S}^m$ is, as before, a convex operator, A^q are $k_q \times n$ matrices and A^l is an $k_l \times n$ matrix. The inequality symbol "$\leq_q$" means that the corresponding vector should be in the second-order cone defined by $K_q = \{z \in \mathbb{R}^q \mid z_1 \geq \|z_{2:q}\|\}$. The SOCP constraints cannot be handled directly by PENNON; written as NLP constraints, they are nondifferentiable at the origin. We can, however, perturb them by a small parameter $\varepsilon > 0$ to avoid the nondifferentiability. So, for instance, instead of constraint

$$a_1 x_1 \leq \sqrt{a_2 x_2^2 + \ldots + a_m x_m^2},$$

we work with a (smooth and convex) constraint

$$a_1 x_1 \leq \sqrt{a_2 x_2^2 + \ldots + a_m x_m^2 + \varepsilon}.$$

The value of ε can be decreased during the iterations of Algorithm 2.1. In PENNON we set $\varepsilon = p \cdot 10^{-6}$, where p is the penalty parameter in Algorithm 2.1. In this way, we obtain solutions of SOCP problems of high accuracy. This is demonstrated in Section 6.

5.2 Nonconvex problems

Algorithm 2.1 is proposed for general convex problems. A natural question arises, whether it can be generalized for finding local minima or stationary points of nonconvex problems. To this purpose we have implemented a modification of Algorithm 2.1 described below. However, when we tested the modified algorithm on various nonconvex problems, we realized that the idea is not yet ready and needs more tuning. It works well on many problems, but it also has severe

difficulties with many other problems. The algorithm is also rather sensitive on various parameters. From these reasons we do not give here any results for nonconvex problems.

Assume that functions f and g_i from (NLP-SDP), or even the operator $\mathcal{A}$, are generally nonconvex. In this case we use a quite simple idea: we apply Algorithm 2.1 and whenever we hit a nonconvex point in Step (i), we switch from the Newton method to the Levenberg-Marquardt method. More precisely, one step of the minimization method in step (i) is defined as follows:

> Given a current iterate (x, U, p), compute the gradient g and Hessian H of F at x.
> Compute the minimal eigenvalue $\lambda_{\min}$ of H. If $\lambda_{\min} < 10^{-3}$, set
>
> $$\widehat{H}(\alpha) = H + (\lambda_{\min} + \alpha)I.$$
>
> Compute the search direction
>
> $$d(\alpha) = -\widehat{H}(\alpha)^{-1} g.$$
>
> Perform line-search in direction $d(\alpha)$. Denote the step-length by s.
> Set
>
> $$x_{\text{new}} = x + sd(\alpha).$$

Obviously, for a convex F, this is just a Newton step with line-search. For nonconvex functions, we can use a shift of the spectrum of H with a fixed parameter $\alpha = 10^{-3}$. As mentioned above, this approach works well on several nonconvex NLP problems but has serious difficulties with other problems. This idea needs more tuning and we believe that a nonconvex version of PENNON will be the subject of a future article.

6 Computational results

Since there are no standard test examples that combine NLP and SDP constraints, we will report on tests done separately for NLP and SDP problems.

To test PENNON on NLP problems, we have created an AMPL interface [7]. We used a test suite of convex quadratic programs by Maros and Mészáros[1] [12]. This set contains selected problems from the BRUNEL and CUTE collections plus some additional problems. We have chosen these test problems, as they were recently used for the benchmark of QP solvers by Mittelmann [13]. We should note that PENNON is not a specialized QP solver; on the contrary, it transforms the QP problem into a general nonlinear one. It does not use any special structure of the problem, like separability, either.

The second set of NLP examples, called `mater`, comes from structural optimization. It contains convex quadratically constrained quadratic problems.

The SDP version of PENNON was tested using two sets of problems: the SDPLIB collection of linear SDPs by Borchers [6]; the set of `mater` and `truss` examples from structural optimization. We describe the results of our testing of PENNON and three other SDP codes, namely CSDP-3.2 by Borchers [5],

[1] Available on-line at `ftp://ftp.sztaki.hu/pub/oplab/QPDATA`. Corresponding AMPL files are available at `ftp://plato.la.asu.edu/pub/ampl_files/qpdata_ampl`.

SDPT3-3.0 by Toh, Todd and Tütüncü [18], and DSDP-4.5 by Benson and Ye [4]. We have chosen these three codes as they were, at the moment of writing this article, the fastest ones in the independent tests performed by Mittelmann [13]. We used the default setting of parameters for all codes. We report on results obtained with PENNON-1.2, a version that is more efficient than the one tested in [10]. This is why the results differ (sometimes substantially) from those reported in [10].

Finally, we present results of selected problems from the DIMACS library [16] that combine SOCP and SDP constraints.

6.1 Convex quadratic programming

Table 1 shows number of successful runs for the problems from the BRUNEL and CUTE collections and additional problems denoted as MISC. The cause of failure was mainly lack of convergence. In some cases (the LISWET problems), the code converged too slowly, in other cases (POWELL20), it actually diverged.

Table 1: `QP` problems

set	problems	solved	failed	memory
BRUNEL	46	45	1	
CUTE	76	67	7	2
MISC	16	12		4

The next Table 2 gives CPU times and iteration counts for selected examples (following [13]).

Table 2: Computational results for selected QP problems using PENNON, performed on Pentium III-M (1000 MHz) with 512 KB memory running SuSE LINUX 7.3. The values in column "iter" denote number of outer iterations/number of Newton steps/number of line-search steps.

problem	CPU	iter
BOYD1	memory	
BOYD2	memory	
CONT-201	128	20/26/75
CONT-300	444	21/26/75
CVXQP1_M	4	14/42/135
CVXQP1_L	1707	19/75/192
CVXQP2_L	670	15/58/139
CVXQP3_L	4497	15/153/2064
EXDATA	memory	
LISWET1	376	98/1494/11262
LISWET10	failed	

6.2 `mater` problems

Program PENNON, both the NLP and SDP versions, was actually developed as a part of a software package MOPED for material optimization. The goal of this

package is to design optimal structures considered as two- or three-dimensional continuum elastic bodies where the design variables are the *material properties* which may vary from point to point. Our aim is to optimize not only the distribution of material but also the material properties themselves. We are thus looking for the ultimately best structure among all possible elastic continua, in a framework of what is now usually referred to as "free material design" (see [21] for details). After analytic reformulation and discretization by the finite element method, the problem reduces to a large-scale convex NLP problem

$$\min_{\alpha\in\mathbb{R},x\in\mathbb{R}^N} \left\{\alpha - c^T x \,|\, \alpha \geq x^T A_i x \text{ for } i = 1,\ldots,M\right\}$$

with positive semidefinite matrices A_i. Here M is the number of finite elements and N the number of degrees of freedom of the displacement vector. For real world problems one should work with discretizations of size $M \approx 20\,000$.

For the purpose of this paper we used an AMPL model of a typical material optimization problem. The examples presented in Table 3 differ in the number of variables and constraints but their character is the same. We must remark that a direct implementation of the problem (without the AMPL interface) leads to much more efficient solution, in terms of memory and CPU time.

Table 3: Computational results for mater problems using PENNON, performed on Pentium III-M (1000 MHz) with 512 KB memory running SuSE LINUX 7.3. "N" is the number of variables "M" the number of constraints. The values in column "iter" denote number of outer iterations/number of Newton steps/number of line-search steps.

problem	N	M	CPU	iter
mater-sl-1	162	64	<1	14/32/72
mater-sl-2	4050	1936	50	19/61/128
mater-sl-3	19602	9604	373	21/82/146
mater-sl-4	42050	20736	1073	20/91/194

6.3 SDPLIB

Due to space limitations, we do not present here the full SDPLIB results and select just several representative problems. Table 4 lists the selected SDPLIB problems, along with their dimensions, and the results for CSDP, SDPT3, DSDP, and PENNON[2]. All codes were used with standard setting; CSDP and PENNON were linked with ATLAS-BLAS library, SDPT3 (with HKM direction) ran under Matlab 6.1.

In several SDPLIB problems, CSDP, SDPT3 or DSDP are faster than PENNON. This is, basically, due to the number of Newton steps used by the particular algorithms. Since the complexity of Hessian assembling is about the same for all three codes, and the data sparsity is handled in a similar way, the main time difference is given by the number of Newton steps. While CSDP and SDPT3 need, in average, 15–30 steps, PENNON needs about 2–3 times more steps. Recall that this is due to the fact that PENNON is based on an

[2]This table is overtaken from `ftp://plato.la.asu.edu/pub/sdplib.txt` with a kind permission of the author.

Table 4: Selected SDPLIB problems and computational results using CSDP, SDPT3, and PENNON, performed on a Pentium II PC (450 MHz) with 512 KB memory running LINUX-2.4.14 and Matlab 6.1. "s" is the number of correct digits in the objective function.

	no. of	size of	CSDP		SDPT3		DSDP		PENNON	
problem	var.	matrix	CPU	s	CPU	s	CPU	s	CPU	s
arch8	174	335	34	7	63	7	26	5	42	6
control7	136	45	513	7	201	6	1101	5	448	7
control10	1326	150	2551	6	899	6	8531	5	6455	6
control11	1596	165	4404	6	1354	6	13830	5	10863	6
gpp250-4	250	250	45	7	43	7	29	6	34	6
gpp500-4	501	500	375	7	257	7	244	7	224	7
hinf15	91	37	1	5	11	5	1	2	11	3
mcp250-1	250	250	26	7	19	7	2	5	15	7
mcp500-1	500	500	171	7	435	7	14	5	79	7
qap9	748	82	27	7	28	4	69	5	40	5
qap10	1021	101	66	7	51	4	137	5	127	5
ss30	132	426	229	7	194	7	65	7	165	7
theta3	1106	150	70	7	61	7	94	7	126	7
theta4	1949	200	311	7	213	7	427	7	603	7
theta5	3028	250	956	7	647	7	1616	7	1693	7
theta6	4375	300	2629	7	1610	7	5379	7	5861	7
truss7	86	301	1	7	7	6	2	7	1	7
truss8	496	628	29	7	65	7	57	7	85	7
equalG11	801	801	1163	7	1027	7	1151	7	1010	6
equalG51	1001	1001	2342	7	2241	7	2899	7	3135	7
maxG11	800	800	573	7	310	7	44	7	213	6
maxG32	2000	2000	7755	7	4140	7	622	7	2690	7
maxG51	1001	1001	1295	7	1047	7	224	7	1630	7
qpG11	800	1600	3875	7	2579	7	98	7	544	7
qpG51	1000	2000	8087	7	5805	7	1263	7	3074	7

algorithm for general nonlinear convex problems and allows to solve larger class of problems. This is the price we pay for the generality. We believe that, in this light, the code is competitive.

The SDPLIB collection includes some very-large-scale problems, which are impossible to solve on a PC, due to memory limitations. We have solved these problems on an SGI Origin 3400 parallel computer with 56 GB main memory. We did not use any paralellization of PENNON but linked it with parallel BLAS and LAPACK libraries. The examples were solved using 4 processors. Table 5 shows the computational time needed by PENNON. We did not compare these times with any other SDP code; the purpose of these tests was to see whether PENNON is able to solve these large-scale problems and whether its behaviour deteriorates in any way. The results show that PENNON is very robust, at least for this group of problems. (To get a comparison with other computers, we included a smaller problem maxG51, contained in Table 4 above. Obviously, such comparison of a sequential and parallel computer gives only a very vague idea.).

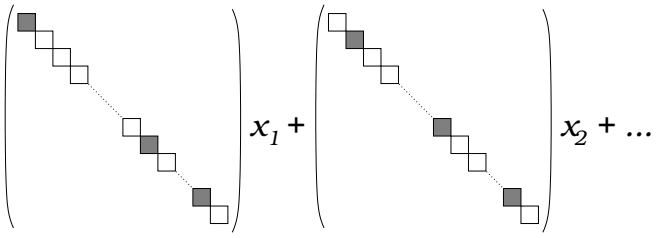
Table 5: Computational results on large SDPLIB problems using PENNON, performed on SGI Origin 3400 with 56 GB memory using 4 processors. The values in column "iter" denote number of outer iterations/number of Newton steps/number of line-search steps. "s" is the number of correct digits in the objective function.

			PENNON		
problem	n	m	time	s	iter
thetaG11	2401	801	991 (16m31s)	9	26/98/124
thetaG51	6910	1001	20277 (5h37m57s)	9	25/106/145
maxG51	1000	1000	554 (9m14s)	7	22/53/66
maxG55	5000	5000	120425 (33h27m5s)	7	22/50/65
maxG60	7000	7000	226586 (62h56m26s)	7	23/47/68

6.4 `mater` and `truss` problems

Next we present results of two sets of examples coming from structural optimization. The first set contains examples from free material optimization introduced above. While the single-load problem can be formulated as a convex NLP, the more realistic multiple-load is modeled by linear SDP as described in [1]. All examples solve the same problem (geometry, loads, boundary conditions) and differ only in the finite element discretization.

The linear matrix operator $\mathcal{A}(x) = \sum A_i x_i$ has the following structure: A_i are block diagonal matrices with many ($\sim$5 000) small (11×11–20×20) blocks. Moreover, only few (6–12) of these blocks are nonzero in any A_i, as schematically shown in the figure below.



As a result, the Hessian of the augmented Lagrangian associated with this problem is a large and sparse matrix. PENNON proved to be particularly efficient for this kind of problems, as shown in Table 7.

The following results are overtaken from Mittelmann [13] and were obtained[3] on Sun Ultra 60, 450 MHz with 2 GB memory, running Solaris 8. Table 6 shows the dimensions of the problems, together with the optimal objective value. Table 7 presents the test results for CSDP, SDPT3 and PENNON. It turned out that for this kind of problems, the code SeDuMi by Sturm [17] was rather competitive, so we included also this code in the table.

In the second set, we solve problems from truss topology design:

- `trto` are problems from single-load truss topology design. Normally formulated as LP, here reformulated as SDP for testing purposes (see, eg, [2, 11]).

[3]Except of mater-5 solved by CSDP and mater-6 solved by CSDP and SDPT3. These were obtained using Sun E6500, 400 MHz with 24 GB memory

Table 6: `mater` problems

problem	n	m	Optimal value
mater-3	1439	3588	-1.339163e+02
mater-4	4807	12498	-1.342627e+02
mater-5	10143	26820	-1.338016e+02
mater-6	20463	56311	-1.335387e+02

Table 7: Computational results for `mater` problems using SDPT3, CSDP, SeDuMi, and PENNON, performed on a Sun Ultra 60 (450 MHz) with 2 GB of memory running Solaris 8. "s" is the number of correct digits in the objective function.

	SDPT3		CSDP		SeDuMi		PENNON	
problem	CPU	s	CPU	s	CPU	s	CPU	digits
mater-3	569	7	129	8	54	11	48	10
mater-4	7675	5	2555	8	295	11	260	9
mater-5	40902	5	258391	8	692	10	600	8
mater-6	memory		memory		1957	8	1612	8

- `vibra` are single load truss topology problems with a vibration constraint. The constraint guarantees that the minimal self-vibration frequency of the optimal structure is bigger than a given value; see [9].
- `buck` are single load truss topology problems with linearized global buckling constraint. Originally a nonlinear matrix inequality, the constraint should guarantee that the optimal structure is mechanically stable (does not buckle); see [9].

All problems from this set are characterized by sparsity of the linear matrix operator $\mathcal{A}$. These problems are very difficult to solve. From all the tested SDP solvers, only SDPT3 and PENNON, followed by DSDP, could solve them efficiently. SeDuMi could only solve small problems and needed unacceptable length of time for larger ones. CSDP crashed on almost all problems from this set. In the following tables, we only show comparison of PENNON with SDPT3. Table 8 gives problem characteristics, while Table 9 presents results of the test runs.

When we started to test these examples with PENNON, we realized that the line-search procedure is very inefficient for this group of problems. It turned out that it is much more efficient to avoid line-search and do always a full Newton step. Results in Table 9 were obtained with this version of the code. This idea (to avoid line-search) is, however, not always a good one (in some SDPLIB examples), and this part of the algorithm needs certainly more attention and future development.

6.5 DIMACS

Finally, in Table 10 we present results of selected problems from the DIMACS collection. These are mainly SOCP problems, apart from `filter48-socp` that combines SOCP and SDP constraints. The results demonstrate that we can reach high accuracy even when working with the smooth reformulation of the

Table 8: `truss` problems. (e) denotes known exact optimal value; n is the number of variables, m the size of the matrix constraint; 25+36 means: matrix constraint of size 25 and 36 linear constraints

problem	n	m	Optimal value
trto1	36	25+36	1.1045 (e)
trto2	144	91+144	1.28 (e)
trto3	544	321+544	1.28 (e)
trto4	1200	673+1200	1.276582
trto5	3280	1761+3280	1.28 (e)
buck1	36	49+36	14.64192
buck2	144	193+144	292.3683
buck3	544	641+544	607.6055
buck4	1200	1345+1200	486.1421
buck5	3280	3521+3280	436.2292
vibra1	36	49+36	40.81901
vibra2	144	193+144	166.0153
vibra3	544	641+544	172.6130
vibra4	1200	1345+1200	165.6133
vibra5	3280	3521+3280	165.9029

SOCP constraints (see Section 5.1). The results also show the influence of linear constraints on the efficiency of the algorithm; cf. problems `nb` and `nb-L1`. This is due to the fact that, in our algorithm, the part of the Hessian corresponding to every (penalized) linear constraint is a dyadic, i.e., possibly full matrix. We are working on an approach that treats linear constraints separately.

Acknowledgment

The authors would like to thank Hans Mittelmann for his help when testing the code and for implementing PENNON on the NEOS server. This research was supported by BMBF project 03ZOM3ER. The first author was partly supported by grant No. 201/00/0080 of the Grant Agency of the Czech Republic.

References

[1] A. Ben-Tal, M. Kočvara, A. Nemirovski, and J. Zowe. Free material design via semidefinite programming. The multi-load case with contact conditions. *SIAM J. Optimization*, 9:813–832, 1997.

[2] A. Ben-Tal and A. Nemirovski. *Lectures on Modern Convex Optimization*. MPS-SIAM Series on Optimization. SIAM Philadelphia, 2001.

[3] A. Ben-Tal and M. Zibulevsky. Penalty/barrier multiplier methods for convex programming problems. *SIAM J. Optimization*, 7:347–366, 1997.

[4] S. J. Benson and Y. Ye. DSDP4 users manual. Report ANL/MCS-TM-248, Argonne National Laboratory, Argonne, 2002. Available at `http://www-unix.mcs.anl.gov/~benson/`.

Table 9: Computational results for `truss` problems using SDPT3 and PENNON, performed on a Pentium III PC (650 MHz) with 512 KB memory running SuSE LINUX 7.3. (S) means that the code failed with "Schur complement not positive definite" or "Lack of progress" but close to convergence, usually 5–6 digits accuracy. For PENNON, "iter" denotes number of outer iterations/number of Newtons steps. "s" is the number of correct digits in the objective function.

	SDPT3			PENNON		
problem	CPU	iter	s	CPU	iter	s
trto1	1	15	7	<1	20/49	8
trto2	8	22	7	5	22/83	8
trto3	102	26	6	89	25/113	8
trto4	777	35	4	486	28/100	8
trto5	7803	35(S)	4	8862	31/142	8
buck1	2	17	7	<1	23/40	7
buck2	15	21(S)	7	8	29/82	7
buck3	298	37(S)	5	164	38/108	7
buck4	1602	35	6	992	50/109	7
buck5	19124	42(S)	4	14577	50/136	7
vibra1	2	14	7	<1	25/61	7
vibra2	18	25	7	7	30/76	7
vibra3	718	36	5	163	35/107	7
vibra4	2910	44	5	818	37/90	8
vibra5	21494	47(S)	5	18571	42/178	8

[5] B. Borchers. CSDP, a C library for semidefinite programming. *Optimization Methods and Software*, 11:613–623, 1999. Available at `http://www.nmt.edu/~borchers/`.

[6] 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/`.

[7] R. Fourer, D. M. Gay, and B. W. Kerningham. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, 1993.

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

[9] M. Kočvara. On the modelling and solving of the truss design problem with global stability constraints. *Struct. Multidisc. Optimization*, 2002. In print.

[10] M. Kočvara and M. Stingl. PENNON—a generalized augmented Lagrangian method for semidefinite programming. Preprint 286, Institute of Applied Mathematics, University of Erlangen, 2001.

[11] M. Kočvara and J. Zowe. How mathematics can help in design of mechanical structures. In D.F. Griffiths and G.A. Watson, editors, *Numerical Analysis 1995*, pages 76–93. Longman, Harlow, 1996.

Table 10: Computational results on DIMACS problems using PENNON, performed on a Pentium III PC (650 MHz) with 512 KB memory running SuSE LINUX 7.3. Notation like [793x3] indicates that there were 793 (semidefinite, second-order, linear) blocks, each a symetric matrix of order 3.

					PENNON	
problem	n	SDP blocks	SO blocks	lin. blocks	CPU	s
nb	123	–	[793x3]	4	60	7
nb-L1	915	–	[793x3]	797	141	7
nb-L2	123	–	[1677,838x3]	4	100	8
nb-L2-bessel	123	–	[123,838x3]	4	90	8
qssp30	3691	–	[1891x4]	2	10	6
qssp60	14581	–	[7381x4]	2	55	5
nql30	3680	–	[900x3]	3602	17	4
filter48-socp	969	48	49	931	283	6

[12] I. Maros and C. Mészáros. A repository of convex quadratic programming problems. *Optimization Methods and Software*, 11&12:671–681, 1999.

[13] H. Mittelmann. Benchmarks for optimization software. Available at `http://plato.la.asu.edu/bench.html`.

[14] L. Mosheyev and M. Zibulevsky. Penalty/barrier multiplier algorithm for semidefinite programming. *Optimization Methods and Software*, 13:235–261, 2000.

[15] E. Ng and B. W. Peyton. Block sparse cholesky algorithms on advanced uniprocessor computers. *SIAM J. Scientific Computing*, 14:1034–1056, 1993.

[16] G. Pataki and S. Schieta. The DIMACS library of mixed semidefinite-quadratic-linear problems. Available at `http://dimacs.rutgers.edu/challenges/seventh/instances`.

[17] J. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optimization Methods and Software*, 11 & 12:625–653, 1999. Available at `http://fewcal.kub.nl/sturm/`.

[18] R.H. Tütüncü, K.C. Toh, and M.J. Todd. SDPT3 — A MATLAB software package for semidefinite-quadratic-linear programming, Version 3.0. Available at `http://www.orie.cornell.edu/~miketodd/todd.html`, School of Operations Research and Industrial Engineering, Cornell University, 2001.

[19] M. Zibulevsky. New penalty/barrier and lagrange multiplier approach for semidefinite programming. Research Report 5/95, Optimization Laboratory, Technion, Israel, 1995.

[20] M. Zibulevsky. *Penalty/barrier multiplier methods for large-scale nonlinear and semidefinite programming.* PhD thesis, Technion—Israel Institute of Technology, Haifa, 1996.

[21] J. Zowe, M. Kočvara, and M. Bendsøe. Free material optimization via mathematical programming. *Mathematical Programming, Series B*, 79:445–466, 1997.