PENNON

A Generalized Augmented Lagrangian Method for Semidefinite Programming

Michal Kočvara*
Institute of Applied Mathematics, University of Erlangen
Martensstr. 3, 91058 Erlangen, Germany
kocvara@am.uni-erlangen.de

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

Abstract This article describes a generalization of the PBM method by Ben-Tal and Zibulevsky to convex semidefinite programming problems. The algorithm used is a generalized version of the Augmented Lagrangian method. We present details of this algorithm as implemented in a new code PENNON. 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 SDP codes are presented.

Keywords: semidefinite programming; cone programming; method of augmented Lagrangians

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 def-

*On leave from the Academy of Sciences of the Czech Republic inition 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.

Our aim in this paper is to generalize the PBM approach to convex semidefinite programming problems. The idea is to use the PBM penalty function to construct another function that penalizes matrix inequality constraints. We will show that a direct generalization of the method may lead to an inefficient algorithm and present an idea how to make the method efficient again. The idea is based on a special choice of the penalty function for matrix inequalities. We explain how 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. We introduce a new code PENNON, based on the generalized PBM algorithm, and give details of its implementation. The code is not only aimed at SDP problems but at general convex problems with a mixture of NLP, SOCP and SDP constraints. A generalization to nonconvex situation has been successfully tested for NLP problems. In the last section we present results of extensive numerical tests and comparison with other SDP codes. We will demonstrate that PENNON is particularly efficient when solving problems with sparse data structure and sparse Hessian.

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.

1. The problem and the method

Our goal is to solve problems of convex semidefinite programming, that is problems of the type

$$\min_{x\in\mathbb{R}^n} \left\{ b^T x : \mathcal{A}(x) \preccurlyeq 0 \right\} \tag{SDP}$$

where $b \in \mathbb{R}^n$ and $\mathcal{A} : \mathbb{R}^n \to \mathbb{S}^m$ is a convex operator. The basic idea of our approach is to generalize the PBM method, developed originally by Ben-Tal and Zibulevsky for convex NLPs, to problem (SDP). The method is based on a special choice of a one-dimensional penalty/barrier function φ that penalizes inequality constraints. Below we show how to use this function to construct another function Φ that penalizes the matrix inequality constraint in (SDP).

Let $\varphi : \mathbb{R} \to \mathbb{R}$ 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) $\lim_{t\to -\infty} \varphi'(t) = 0$,

Let further $A = S^T \Lambda S$, where $\Lambda = \mathrm{diag}\,(\lambda_1, \lambda_2, \ldots, \lambda_d)^T$, be an eigenvalue decomposition of a matrix A. Using φ, we define a *penalty function* $\Phi_p : \mathbb{S}^m \to \mathbb{S}^m$ as follows:

$$\Phi_p : A \longmapsto S^T \begin{pmatrix} p\varphi\left(\frac{\lambda_1}{p}\right) & 0 & \ldots & 0 \\ 0 & p\varphi\left(\frac{\lambda_2}{p}\right) & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \ldots & 0 & p\varphi\left(\frac{\lambda_d}{p}\right) \end{pmatrix} S\,, \tag{1}$$

where $p > 0$ is a given number.

From the definition of φ it follows that for any $p > 0$ we have

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

that means that, for any $p > 0$, problem (SDP) has the same solution as the following "augmented" problem

$$\min_{x\in\mathbb{R}^n} \left\{ b^T x : \Phi_p(\mathcal{A}(x)) \preccurlyeq 0 \right\} . \tag{SDP$_\Phi$}$$

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

$$F(x, U, p) = b^T x + \langle U, \Phi_p\left(\mathcal{A}(x)\right)\rangle_{\mathbb{S}_m} ; \tag{2}$$

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

We can now define the basic algorithm that combines ideas of the (exterior) penalty and (interior) barrier methods with the Augmented Lagrangian method.

Algorithm 1.1. *Let* x^1 *and* U^1 *be given. Let* $p^1 > 0$. *For* $k = 1, 2, \dots$ *repeat till a stopping criterium is reached:*

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

Details of the algorithm, the choice of initial values of x, U and p, the approximate minimization in step (i) and the update formulas, will be discussed in detail in subsequent sections. The next section concerns the choice of the penalty function Φ_p.

2. The choice of the penalty function Φ_p

As mentioned in the Introduction, Algorithm 1.1 is a generalization of the PBM method by Ben-Tal and Zibulevsky [3] (introduced for convex NLPs) to convex SDP problems. 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, $i = 1, \dots, 6$, is chosen so that (φ_1)–(φ_5) hold.

It turns out that function which work well in the NLP case may not be the best choice for SDP problems. The reason is twofold.

First, it may happen that, even if the function φ and the operator $\mathcal{A}$ are convex, the penalty function Φ_p may be *nonconvex*. For instance, function Φ_p defined through the right, quadratic branch of the quadratic-logarithmic function φ^{ql} is nonmonotone and its composition with a convex nonlinear operator $\mathcal{A}$ may result in a nonconvex function $\Phi_p(\mathcal{A}(x))$. Even for linear operator $\mathcal{A}$, $\Phi_p(\mathcal{A}(x))$ corresponding to φ^{ql} may be nonconvex. This nonconvexity may obviously bring difficulties to Algorithm 1.1 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 1.1 is performed be 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). The detail analysis of the algorithmic complexity will be given in Section 3. It is based on formulas for the first and second derivatives of Φ_p presented below.

Denote by $\triangle^i$ the divided difference of i-th order, $i = 1, 2$, defined by

$$\triangle^1\varphi(t_i, t_j) := \begin{cases} \dfrac{\varphi(t_i) - \varphi(t_j)}{t_i - t_j} & \text{for } t_i \neq t_j \\ \varphi'(t_i) & \text{for } t_i = t_j \end{cases}$$

and

$$\triangle^2\varphi(t_i, t_j, t_k) := \begin{cases} \dfrac{\triangle^1\varphi(t_i, t_j) - \triangle^1\varphi(t_i, t_k)}{t_j - t_k} & \text{for } t_j \neq t_k \\ \dfrac{\triangle^1\varphi(t_i, t_k) - \triangle^1\varphi(t_j, t_k)}{t_i - t_j} & \text{for } t_i \neq t_j, t_j = t_k \\ \varphi''(t_i) & \text{for } t_i = t_j = t_k\,. \end{cases}$$

Theorem 2.1. *Let $\mathcal{A} : \mathbb{R}^n \to \mathbb{S}^m$ be a convex operator. Let further Φ_p be a function defined by (1). Then for any $x \in \mathbb{R}^n$ the first and second partial derivatives of $\Phi_p(\mathcal{A}(x))$ are given by*

$$\begin{aligned} &\frac{\partial}{\partial x_i}\Phi_p(\mathcal{A}(x)) \\ &\quad = S\left(\left[\triangle^1\varphi(\lambda_r(x), \lambda_s(x))\right]_{r,s=1}^m \circ \left[S(x)^T\frac{\partial\mathcal{A}(x)}{\partial x_i}S(x)\right]\right)S^T \end{aligned} \tag{4}$$

$$\begin{aligned} &\frac{\partial^2}{\partial x_i \partial x_j}\Phi_p(\mathcal{A}(x)) \\ &\quad = 2S\Big(\sum_{k=1}^m [\triangle^2\varphi(\lambda_r(x), \lambda_s(x), \lambda_k(x))]_{r,s=1}^m \\ &\qquad \circ \left[S(x)^T\frac{\partial\mathcal{A}(x)}{\partial x_i}S(x)E_{kk}S(x)^T\frac{\partial\mathcal{A}(x)}{\partial x_j}S(x)\right]\Big)S^T\,. \end{aligned} \tag{5}$$

We can avoid the above mentioned drawbacks by a choice of the function φ. In particular, we search a function that allows for a "direct" computation of Φ_p and its first and second derivatives. The function of our choice is the reciprocal barrier function

$$\varphi^{\text{rec}}(t) = \frac{1}{t-1} - 1\,. \tag{6}$$

Theorem 2.2. *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

$$\Phi_p^{\mathrm{rec}}(\mathcal{A}(x)) = p^2 \mathcal{Z}(x) - pI \tag{7}$$

$$\frac{\partial}{\partial x_i}\Phi_p^{\mathrm{rec}}(\mathcal{A}(x)) = p^2 \mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_i}\mathcal{Z}(x) \tag{8}$$

$$\frac{\partial^2}{\partial x_i \partial x_j}\Phi_p^{\mathrm{rec}}(\mathcal{A}(x)) = p^2 \mathcal{Z}(x)\left(\frac{\partial \mathcal{A}(x)}{\partial x_i}\mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_j} - \frac{\partial^2 \mathcal{A}(x)}{\partial x_i \partial x_j} + \frac{\partial \mathcal{A}(x)}{\partial x_j}\mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_i}\right)\mathcal{Z}(x) \tag{9}$$

where

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

Furthermore, $\Phi_p^{\mathrm{rec}}(\mathcal{A}(x))$ is monotone and convex in x.

Proof. Let I_m denote the identity matrix of order m. Since $\mathcal{Z}(x)$ is differentiable and nonsingular at x we have

$$\begin{aligned} 0 &= \frac{\partial}{\partial x_i} I_m = \frac{\partial}{\partial x_i}\left[\mathcal{Z}(x)\mathcal{Z}^{-1}(x)\right] \\ &= \left[\frac{\partial}{\partial x_i}\mathcal{Z}(x)\right]\mathcal{Z}^{-1}(x) + \mathcal{Z}(x)\left[\frac{\partial}{\partial x_i}\mathcal{Z}^{-1}(x)\right], \end{aligned} \tag{10}$$

so the formula

$$\frac{\partial}{\partial x_i}\mathcal{Z}(x) = -\mathcal{Z}(x)\left[\frac{\partial}{\partial x_i}\mathcal{Z}^{-1}(x)\right]\mathcal{Z}(x) = -\mathcal{Z}(x)\left[\frac{\partial \mathcal{A}(x)}{\partial x_i}\right]\mathcal{Z}(x) \tag{11}$$

follows directly after multiplication of (10) by $\mathcal{Z}(x)$ and (8) holds. For the proof of (9) we differentiate the right hand side of (11)

$$\begin{aligned} \frac{\partial^2}{\partial x_i \partial x_j}\mathcal{Z} &= -\frac{\partial}{\partial x_i}\left(\mathcal{Z}(x)\left[\frac{\partial \mathcal{A}(x)}{\partial x_j}\right]\mathcal{Z}(x)\right) \\ &= -\left[\frac{\partial}{\partial x_i}\mathcal{Z}(x)\right]\frac{\partial \mathcal{A}(x)}{\partial x_j}\mathcal{Z}(x) - \mathcal{Z}(x)\left[\frac{\partial}{\partial x_i}\left(\frac{\partial \mathcal{A}(x)}{\partial x_j}\mathcal{Z}(x)\right)\right] \\ &= \mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_i}\mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_j}\mathcal{Z}(x) - \mathcal{Z}(x)\frac{\partial^2 \mathcal{A}(x)}{\partial x_i \partial x_j}\mathcal{Z}(x) \\ &\quad -\mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_j}\left[\frac{\partial}{\partial x_i}\mathcal{Z}(x)\right] \\ &= \mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_i}\mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_j}\mathcal{Z}(x) - \mathcal{Z}(x)\frac{\partial^2 \mathcal{A}(x)}{\partial x_i \partial x_j}\mathcal{Z}(x) \\ &\quad +\mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_j}\mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_i}\mathcal{Z}(x) \end{aligned}$$

and (9) follows. For the proof of convexity and monotonicity of Φ_p^{rec} we refer to [13]. $\square$

Using Theorem 2.2 we can compute the value of Φ_p^{rec} and its derivatives directly, without the need of eigenvalue decomposition of $\mathcal{A}(x)$. The "direct" formulas (8)–(9) are particularly simple for affine operator

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

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

3. Complexity

Computational complexity of Algorithm 1.1 is dominated by construction of the Hessian of the augmented Lagrangian (2). Our complexity analysis is therefore limited to this issue.

3.1. The general approach

As we can easily see from Theorem 2.1, the part of the Hessian corresponding to the inner product in formula (2) is given by

$$\left[\sum_{k=1}^{m} s_k^T \frac{\partial \mathcal{A}(x)}{\partial x_i} \left[S(x)\Big(Q_k \circ [S(x)^T U S(x)]\Big) S(x)^T\right] \frac{\partial \mathcal{A}(x)}{\partial x_j} s_k\right]_{i,j=1}^{n} \tag{12}$$

where Q_k denotes the matrix $[\Delta^2\varphi(\lambda_r(x), \lambda_s(x), \lambda_k(x))]_{r,s=1}^m$ and s_k is the k-th row of the matrix $S(x)$. Essentially, the construction is done in three steps, shown below together with their complexity:

- For all k compute matrices $S(x)\Big(Q_k \circ [S(x)^T U S(x)]\Big) S(x)^T \longrightarrow O(m^4)$.
- For all k, i compute vectors $s_k^T \dfrac{\partial \mathcal{A}(x)}{\partial x_i} \longrightarrow O(nm^3)$.
- Multiply and sum up expressions above $\longrightarrow O(m^3 n + m^2 n^2)$.

Consequently the Hessian assembling takes $O(m^4 + m^3 n + m^2 n^2)$ time. Unfortunately, if the constraint matrices $\frac{\partial \mathcal{A}(x)}{\partial x_i}$ are sparse, the complexity formula remains the same. This is due to the fact, that the matrices Q_k and $S(x)$ are generally dense, even if the matrix $\mathcal{A}(x)$ is very sparse.

3.2. Function Φ_p^{rec}

If we replace the general penalty function by the reciprocal function Φ_p^{rec} then, according to Theorem 2.2, the part of the Hessian corresponding to the inner product in formula (2) can be written as

$$\begin{aligned}&\left[\left\langle \mathcal{Z}(x)U\mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_i}\mathcal{Z}(x), \frac{\partial \mathcal{A}(x)}{\partial x_j}\right\rangle\right]_{i,j=1}^n \\ &+ \left[\left\langle \mathcal{Z}(x)U\mathcal{Z}(x), \frac{\partial^2 \mathcal{A}(x)}{\partial x_i \partial x_j}\right\rangle\right]_{i,j=1}^n \\ &+ \left[\left\langle \mathcal{Z}(x)U\mathcal{Z}(x)\frac{\partial \mathcal{A}(x)}{\partial x_j}\mathcal{Z}(x), \frac{\partial \mathcal{A}(x)}{\partial x_i}\right\rangle\right]_{i,j=1}^n .\end{aligned} \tag{13}$$

It is straightforward to see that the complexity of assembling of (13) is given by $O(m^3n+m^2n^2)$. In contrast to the general approach, for sparse constraint matrices with $O(1)$ entries, the complexity formula reduces to $O(m^2n+n^2)$.

4. The code PENNON

Algorithm 1.1 was implemented (mainly) in the C programming language and this implementation gave rise to a computer program called PENNON[1]. In this section we describe implementation details of this code.

4.1. Block diagonal structure

Many semidefinite constraints can be written in block diagonal form

$$\mathcal{A}(x) = \begin{pmatrix} \mathcal{A}_1(x) & & & & \\ & \mathcal{A}_2(x) & & & \\ & & \ddots & & \\ & & & \mathcal{A}_{k_s}(x) & \\ & & & & \mathcal{A}^l(x) \end{pmatrix} \preccurlyeq 0,$$

where $\mathcal{A}^l(x)$ is a diagonal matrix of order k_l, each entry of which has the form $a_i^T x - c_i$. Using this, we can reformulate the original problem

[1] `http://www2.am.uni-erlangen.de/~kocvara/pennon/`

(SDP) as

$$\begin{aligned}
&\min_{x\in\mathbb{R}^n} b^T x \\
&\text{s.t.}\quad \mathcal{A}_j(x) \preccurlyeq 0, \qquad j = 1,\dots,k_s, \\
&\qquad\;\; g_i(x) \le 0, \qquad i = 1,\dots,k_l,
\end{aligned}$$

where g_i, $i = 1,\dots,k$, are real valued affine linear functions. This is the formulation solved by our algorithm. The corresponding augmented Lagrangian can be written as follows:

$$F(x,U,u,p) = b^T x + \sum_{j=1}^{k_s} \langle U_j, \Phi_p(\mathcal{A}_j(x))\rangle_{\mathbb{S}^{m_j}} + \sum_{i=1}^{k_l} \langle u_i, \varphi_p(g_i(x))\rangle_{\mathbb{R}},$$

where $U = (U_1,\dots,U_k) \in \mathbb{S}^{m_1} \times \dots \times \mathbb{S}^{m_{k_s}}$ and $u = (u_1,\dots,u_{k_l}) \in \mathbb{R}^{k_l}$ are the Lagrangian multipliers and $p \in \mathbb{R}^{k_s} \times \mathbb{R}^{k_l}$ is the vector of penalty parameters associated with the inequality constraints .

4.2. Initialization

As we have seen in Theorem 2.2, our algorithm can start with an arbitrary primal variable $x \in \mathbb{R}^n$. Therefore we simply choose $x^0 = 0$. The initial values of the multipliers are set to

$$\begin{aligned}
U_j^0 &= \mu_j^s I_{m_j}, \quad j = 1,\dots,k_s, \\
u_i^0 &= \mu_i^l, \qquad\quad i = 1,\dots,k_l,
\end{aligned}$$

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

$$\mu_j^s = m_j \max_{1\le\ell\le n} \frac{1+|b_j|}{1+\left\|\frac{\partial\mathcal{A}(x)}{\partial x_\ell}\right\|}, \tag{14}$$

$$\mu_i^l = \max_{1\le\ell\le n} \frac{1+|b_i|}{1+\left\|\frac{\partial g(x)}{\partial x_\ell}\right\|}. \tag{15}$$

Furthermore, we calculate $\pi > 0$ so that

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

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

4.3. Unconstrained minimization

The tool used in step (i) of Algorithm 1.1 (approximate unconstrained minimization) is the modified Newton method combined with a cubic linesearch. In each step we calculate the search direction d by solving the Newton equation and find α_{max} so that the conditions

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

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

4.4. Update of multipliers

First we would like to motivate the multiplier update formula in Algorithm 1.1.

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} as in Algorithm 1.1 we have*

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

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

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

$$\nabla_x F(x, U, p) = b + \begin{pmatrix} \left\langle U, D_{\mathcal{A}}\Phi_p\left(\mathcal{A}(x); \frac{\partial \mathcal{A}(x)}{\partial x_1}\right)\right\rangle \\ \vdots \\ \left\langle U, D_{\mathcal{A}}\Phi_p\left(\mathcal{A}(x); \frac{\partial \mathcal{A}(x)}{\partial x_n}\right)\right\rangle \end{pmatrix}. \qquad (16)$$

It can be shown that (16) can be written as

$$b + \mathcal{A}^* D_{\mathcal{A}}\Phi_p\left(\mathcal{A}(x); U\right),$$

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

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

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

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

$$U^{k+1} = (p^k)^2 \mathcal{Z}(x) U^k \mathcal{Z}(x), \qquad (17)$$

where $\mathcal{Z}$ was defined in Theorem 2.2.

Numerical test 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:

1. Calculate U^{k+1} using the update formula in Algorithm 1.1.
2. Choose some positive $\lambda \leq 1$, typically 0.7.
3. If the eigenvalues $\lambda_{min}(U^k), \lambda_{max}(U^k), \lambda_{min}(U^{k+1})$ and $\lambda_{max}(U^{k+1})$ can be calculated in a reasonable amount of time, check the inequalities
$$\begin{aligned} \frac{\lambda_{max}(U^{k+1})}{\lambda_{max}(U^k)} &> \frac{1}{1-\lambda}\,, \\ \frac{\lambda_{min}(U^{k+1})}{\lambda_{min}(U^k)} &< 1-\lambda\,. \end{aligned}$$
4. If both inequalities hold, use the initial update formula. If at least one of the inequalities is violated or if calculation of the eigenvalues is too complex, update the current multiplier by
$$U^{new} = U^k + \lambda(U^{k+1} - U^k). \tag{18}$$

4.5. Stopping criteria and penalty update

When testing our algorithm we observed that Newton method needs many steps during the first global iterations. To improve this, we adopted the following strategy: During the first three iterations we do not update the penalty vector p at all. Furthermore, we stop the unconstrained minimization if $\|\nabla_x F(x, U, p)\|$ is smaller than some $\alpha_0 > 0$, which is not too small, typically 1.0. After this kind of "warm start", we change the stopping criterion for the unconstrained minimization to $\|\nabla_x F(x, U, p)\| \leq \alpha$, where in most cases $\alpha = 0.01$ is a good choice. Algorithm 1.1 is stopped if one of the inequalities holds:
$$\frac{|b^T x^k - F(x^k, U^k, p)|}{|b^T x^k|} < \epsilon\,, \qquad \frac{|b^T x^k - b^T x^{k-1}|}{|b^T x|} < \epsilon\,,$$
where ϵ is typically 10^{-7}.

4.6. Sparse linear algebra

Many semidefinite programs have very sparse data structure and therefore have to be treated by sparse linear algebra routines. In our implementation, we use sparse linear algebra routines to perform the following two tasks:

Construction of the Hessian. In each Newton step, the Hessian of the augmented Lagrangian has to be calculated. As we have seen in Section 3, 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 [6], where one can find the ideas we follow in our implementation.

Cholesky factorization. The second 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 [11], 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.

5. Remarks

5.1. SOCP problems

Let us recall that the PBM method was originally developed for large-scale NLP problems. Our generalized method can therefore naturally handle problems with both NLP and SDP constraints, whereas the NLP constraints are penalized by the quadratic–logarithmic function φ^{ql} from (3) and the augmented Lagrangian contains terms from both kind of constraints. The main change in Algorithm 1.1 is in step (ii), the multiplier update, that is now done separately for different kind of constraints.

The method 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

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

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 1.1. In PENNON we set $\varepsilon = p \cdot 10^{-6}$, where p is the penalty parameter in Algorithm 1.1. In this way, we obtain solutions of SOCP problems of high accuracy. This is demosntrated in Section 6.

5.2. Convex and nonconvex problems

We would like to emphasize that, although used only for linear SDP so far, Algorithm 1.1 is proposed for general convex problems. This should be kept in mind when comparing PENNON (on test sets of linear problems) with other codes that are based on genuine linear algorithms.

We can go even a step further and try to generalize Algorithm 1.1 to nonlinear *nonconvex* problems, whereas the nonconvexity can be both in the NLP and in the SDP constraint. Examples of nonconvex SDP problems can be found in [1, 8, 9]. How to proceed in this case? The idea is quite simple: we apply Algorithm 1.1 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 $\alpha = 10^{-3}$. This approach proved to work well on several nonconvex NLP problems and we have reasons to believe that it will work for nonconvex SDPs, too. Obviously, the fixed shift is just the simplest approach and one can use more sophisticated ones like a plane-search (w.r.t. α and s), as proposed in [8], or an approximate version of the trust-region algorithm.

5.3. Program MOPED

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 [16] for details). After analytic reformulation and discretization by the finite element method, the problem reduces to a large-scale NLP

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

where 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 $N, M \approx 20\,000$.

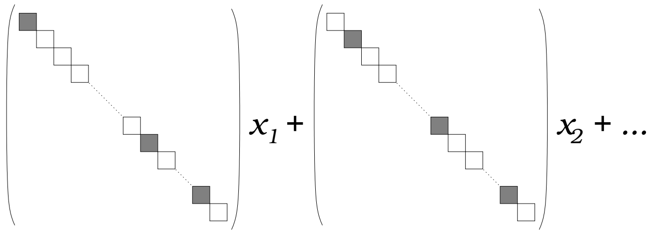
From practical application point of view ([7]), the *multiple-load* formulation of the free material optimization problem is much more important than the above one. Here we look for a structure that is stable with respect to a whole scenario of independent loads and which is the stiffest one in the worst-case sense. In this case, the original "min-max-max" formulation can be rewritten as a linear SDP of the following type (for details, see [2]):

$$\min_{\alpha\in\mathbb{R},x\in(\mathbb{R}^N)^L}\left\{\alpha-\sum_{\ell=1}^{L}(c^\ell)^T x^\ell \mid \mathcal{A}_i(\alpha,x)\succeq 0 \text{ for } i=1,\dots,M\right\};$$

here L is the number of independent load cases (usually 2–4) and $\mathcal{A}_i : \mathbb{R}^{NL+1}\to\mathbb{S}^d$ are linear matrix operators (where d is small). Written in a standard form (SDP), we get a problem with one linear matrix inequality

$$\min_{x\in(\mathbb{R}^n)^L}\left\{a^T x \mid \sum_{i=1}^{nL} x_i B_i\succeq 0\right\},$$

where B_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 B_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 the next section.

6. Computational results

Here we describe the results of our testing of PENNON and two other SDP codes, namely CSDP by Borchers [4] and SDPT3 by Toh, Todd and Tütüncü [15]. We have chosen these two codes as they were, in average, the fastest ones in the independent tests performed by Mittelmann [10]. We have used three sets of test problem: the SDPLIB collection of linear SDPs by Borchers [5]; the set of `mater` examples from multiple-load free material optimization (see Section 5.3); and selected problems from the DIMACS library [12] that combine SOCP and SDP constraints. We used the default setting of parameters for CSDP and SDPT3. PENNON, too, was tested with one setting of parameters for all the problems.

6.1. SDPLIB

Due to space (and memory) limitations, we do not present here the full SDPLIB results and select just several representative problems. Table 1 lists the selected SDPLIB problems, along with their dimensions.

We will present two tables with results obtained on two different computers. The reason for that is that CSDP implementation under LINUX seems to be relatively much faster than under Sun Solaris. On the other hand, we did not have a LINUX computer running MATLAB, hence the comparison with SDPT3 was done on a Sun workstation. Table 1 shows the results of CSDP and PENNON on a 650 MHz Pentium III with 512 KB memory running SuSE LINUX 7.3. PENNON was linked with the ATLAS library, while CSDP binary was taken from Borchers' homepage [4].

Table 1. Selected SDPLIB problems and computational results using CSDP and PENNON, performed on a Pentium III PC (650 MHz) with 512 KB memory running SuSE LINUX 7.3.

			CSDP		PENNON	
problem	n	m	CPU	digits	CPU	digits
arch8	174	335	25	7	79	6
control7	666	105	401	7	327	7
control10	1326	150	1981	6	3400	6
control11	1596	165	3514	6	6230	6
gpp250-4	250	250	33	7	25	7
gpp500-4	501	500	245	7	156	7
hinf15	91	37	1	5	5	3
mcp250-1	250	250	19	7	21	7
mcp500-1	500	500	117	7	175	7
qap9	748	82	21	7	35	5
qap10	1021	101	45	7	107	5
ss30	132	426	167	7	111	7
theta3	1106	150	47	7	97	7
theta4	1949	200	216	7	431	7
theta5	3028	250	686	7	1295	7
theta6	4375	300	1940	7	4346	7
truss7	86	301	1	7	1	7
truss8	496	628	19	7	130	7
equalG11	801	801	749	7	768	6
equalG51	1001	1001	1498	7	3173	7
maxG11	800	800	404	7	611	6
maxG32	2000	2000	5540	7	10924	7
maxG51	1001	1001	875	7	1461	7
qpG11	800	1600	2773	7	3886	7
qpG51	1000	2000	5780	7	7867	7

Table 2 gives results of SDPT3 and PENNON, obtained on Sun Ultra 10 with 384 MB of memory running Solaris 8. SDPT3 was used within MATLAB 6 and PENNON was linked with the ATLAS library.

Table 2. Selected SDPLIB problems and computational results using SDPT3 and PENNON, performed on a Sun Ultra 10 with 384 MB of memory running Solaris 8.

	SDPT3		PENNON	
problem	CPU	digits	CPU	digits
arch8	52	7	203	6
control7	263	6	652	7
control10	1194	6	7082	6
control11	1814	6	13130	6
gpp250-4	46	7	42	6
gpp500-4	266	7	252	7
hinf15	16	5	6	3
mcp250-1	24	7	38	7
mcp500-1	109	7	290	7
qap9	31	4	64	5
qap10	55	4	176	5
ss30	141	7	246	7
theta3	64	7	176	7
theta4	212	7	755	7
theta5	657	7	2070	7
truss7	10	6	2	7
truss8	62	7	186	7
equalG11	1136	7	1252	7
equalG51	2450	7	3645	7
maxG11	500	7	1004	7
maxG51	1269	7	2015	7
qpG11	3341	7	7520	7
qpG51	7525	7	13479	7

In most of the SDPLIB problems, SDPT3 and CSDP 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 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.

6.2. `mater` problems

Next we present results of the `mater` examples. These results are overtaken from Mittelmann [10] and were obtained[2] on Sun Ultra 60, 450 MHz with 2 GB memory, running Solaris 8. Table 4 shows the dimensions of the problems, together with the optimal objective value. Table 5 presents the test results for CSDP, SDPT3 and PENNON. It turned out that for this kind of problems, the code SeDuMi by Sturm [14] was rather competitive, so we included also this code in the table.

Table 3. `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 4. 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.

	SDPT3		CSDP		SeDuMi		PENNON	
problem	CPU	digits	CPU	digits	CPU	digits	CPU	digits
mater-3	718	7	129	8	59	11	50	10
mater-4	9544	5	2555	8	323	11	222	9
mater-5	51229	5	258391	8	738	10	630	8
mater-6	memory		memory		2532	8	1602	8

6.3. DIMACS

Finally, in Table 5 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 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

[2] 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

(penalized) linear constraint is a dyadic, i.e., possibly full matrix. We are working on an approach that treats linear constraints separately.

Table 5. 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	digits
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

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, F. Jarre, M. Kočvara, A. Nemirovski, and J. Zowe. Optimal design of trusses under a nonconvex global buckling constraint. *Optimization and Engineering*, 1:189–213, 2000.

[2] 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.

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

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

[5] 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/`.

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

[7] H.R.E.M. Hörnlein, M. Kočvara, and R. Werner. Material optimization: Bridging the gap between conceptual and preliminary design. *Aerospace Science and Technology*, 2001. In print.

[8] F. Jarre. An interior method for nonconvex semidefinite programs. *Optimization and Engineering*, 1:347–372, 2000.

[9] M. Kočvara. On the modelling and solving of the truss design problem with global stability constraints. *Struct. Multidisc. Optimization*, 2001. In print.

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

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

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

[13] M. Stingl. Konvexe semidefinite programmierung. Diploma Thesis, Institute of Applied Mathematics, University of Erlangen, 1999.

[14] 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/`.

[15] R.H. Tütütcü, 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.

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