

Jan Fiala · Michal Kočvara · Michael Stingl

PENLAB: A MATLAB solver for nonlinear semidefinite optimization^{*}

Received: date / Revised version: date

Abstract. PENLAB is an open source software package for nonlinear optimization, linear and nonlinear semidefinite optimization and any combination of these. It is written entirely in MATLAB. PENLAB is a young brother of our code PENNON [?] and of a new implementation from NAG [?]: it can solve the same classes of problems and uses the same algorithm. Unlike PENNON, PENLAB is open source and allows the user not only to solve problems but to modify various parts of the algorithm. As such, PENLAB is particularly suitable for teaching and research purposes and for testing new algorithmic ideas.

In this article, after a brief presentation of the underlying algorithm, we focus on practical use of the solver, both for general problem classes and for specific practical problems.

1. Introduction

Many problems in various scientific disciplines, as well as many industrial problems lead to (or can be advantageously formulated) as nonlinear optimization problems with semidefinite constraints. These problems were, until recently, considered numerically unsolvable, and researchers were looking for other formulations of their problem that often lead only to approximation (good or bad) of the true solution. This was our main motivation for the development of PENNON [?], a code for nonlinear optimization problems with matrix variables and matrix inequality constraints.

Apart from PENNON, other concepts for the solution of nonlinear semidefinite programs are suggested in literature; see [?] for a discussion on the classic augmented Lagrangian method applied to nonlinear semidefinite programs, [?, ?, ?] for sequential semidefinite programming algorithms and [?] for a smoothing type algorithm. However, to our best knowledge, none of these algorithmic concepts lead to a publicly available code yet.

In this article, we present PENLAB, a younger brother of PENNON and a new implementation from NAG. PENLAB can solve the same classes of problems, uses the same algorithm and its behaviour is very similar. However, its performance is relatively

Jan Fiala: The Numerical Algorithms Group Ltd, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, UK, e-mail: jan@nag.co.uk

Michal Kočvara: School of Mathematics, University of Birmingham, Birmingham B15 2TT, UK and Institute of Information Theory and Automation, Academy of Sciences of the Czech Republic, Pod vodárenskou věží 4, 18208 Praha 8, Czech Republic, e-mail: m.kocvara@bham.ac.uk

Michael Stingl: Applied Mathematics II, University of Erlangen-Nuremberg, Nögelsbachstr. 49b, 91052 Erlangen, Germany, e-mail: stingl@am.uni-erlangen.de

^{*} The research of MK was partly supported by the Grant Agency of the Czech Republic through project GAP201-12-0671.

limited in comparison to [?] and [?], due to MATLAB implementation. On the other hand, PENLAB is open source and allows the user not only to solve problems but to modify various parts of the algorithm. As such, PENLAB is particularly suitable for teaching and research purposes and for testing new algorithmic ideas.

After a brief presentation of the underlying algorithm, we focus on practical use of the solver, both for general problem classes and for specific practical problems, namely, the nearest correlation matrix problem with constraints on condition number, the truss topology problem with global stability constraint and the static output feedback problem. More applications of nonlinear semidefinite programming problems can be found, for instance, in [?,?,?].

PENLAB is distributed under GNU GPL license and can be downloaded from <http://web.mat.bham.ac.uk/kocvara/penlab>.

We use standard notation: Matrices are denoted by capital letters (A, B, X, \dots) and their elements by the corresponding small-case letters ($a_{ij}, b_{ij}, x_{ij}, \dots$). For vectors $x, y \in \mathbb{R}^n$, $\langle x, y \rangle := \sum_{i=1}^n x_i y_i$ denotes the inner product. \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 $\langle A, B \rangle_{\mathbb{S}^m} := \text{Tr}(AB)$. When the dimensions of A and B are known, we will often use notation $\langle A, B \rangle$, same as for the vector inner product. Notation $A \preceq B$ for $A, B \in \mathbb{S}^m$ means that the matrix $B - A$ is positive semidefinite. If A is an $m \times n$ matrix and a_j its j -th column, then $\text{vec } A$ is the $mn \times 1$ vector

$$\text{vec } A = (a_1^T \ a_2^T \ \dots \ a_n^T)^T.$$

Finally, for $\Phi : \mathbb{S}^m \rightarrow \mathbb{S}^m$ and $X, Y \in \mathbb{S}^m$, $D\Phi(X; Y)$ denotes the directional derivative of Φ with respect to X in direction Y .

2. The problem

We intend to solve optimization problems with a nonlinear objective subject to nonlinear inequality and equality constraints and nonlinear matrix inequalities (NLP-SDP):

$$\begin{aligned} & \min_{x \in \mathbb{R}^n, Y_1 \in \mathbb{S}^{p_1}, \dots, Y_k \in \mathbb{S}^{p_k}} f(x, Y) \\ & \text{subject to} \quad g_i(x, Y) \leq 0, & i = 1, \dots, m_g \\ & \quad h_i(x, Y) = 0, & i = 1, \dots, m_h \\ & \quad \mathcal{A}_i(x, Y) \preceq 0, & i = 1, \dots, m_A \\ & \quad \underline{\lambda}_i I \preceq Y_i \preceq \bar{\lambda}_i I, & i = 1, \dots, k. \end{aligned} \quad (1)$$

Here

- $x \in \mathbb{R}^n$ is the vector variable;
- $Y_1 \in \mathbb{S}^{p_1}, \dots, Y_k \in \mathbb{S}^{p_k}$ are the matrix variables, k symmetric matrices of dimensions $p_1 \times p_1, \dots, p_k \times p_k$;
- we denote $Y = (Y_1, \dots, Y_k)$;
- f, g_i and h_i are C^2 functions from $\mathbb{R}^n \times \mathbb{S}^{p_1} \times \dots \times \mathbb{S}^{p_k}$ to \mathbb{R} ;
- $\underline{\lambda}_i$ and $\bar{\lambda}_i$ are the lower and upper bounds, respectively, on the eigenvalues of Y_i , $i = 1, \dots, k$;
- $\mathcal{A}_i(x, Y)$ are twice continuously differentiable nonlinear matrix operators from $\mathbb{R}^n \times \mathbb{S}^{p_1} \times \dots \times \mathbb{S}^{p_k}$ to $\mathbb{S}^{p_{A_i}}$ where $p_{A_i}, i = 1, \dots, m_A$, are positive integers.

3. The algorithm

The basic algorithm used in this article is based on the nonlinear rescaling method of Roman Polyak [?] and was described in detail in [?] and [?]. Here we briefly recall it and stress points that will be needed in the rest of the paper.

The algorithm is based on a choice of penalty/barrier functions $\varphi : \mathbb{R} \rightarrow \mathbb{R}$ that penalize the inequality constraints and $\Phi : \mathbb{S}^p \rightarrow \mathbb{S}^p$ penalizing the matrix inequalities. These functions satisfy a number of properties (see [?, ?]) that guarantee that for any $\pi > 0$ and $\Pi > 0$, we have

$$z(x) \leq 0 \iff \pi \varphi(z(x)/\pi) \leq 0, \quad z \in C^2(\mathbb{R}^n \rightarrow \mathbb{R})$$

and

$$Z \preceq 0 \iff \Pi \Phi(Z/\Pi) \preceq 0, \quad Z \in \mathbb{S}^p.$$

This means that, for any $\pi > 0$, $\Pi > 0$, problem (??) has the same solution as the following “augmented” problem

$$\begin{aligned} & \min_{x \in \mathbb{R}^n, Y_1 \in \mathbb{S}^{p_1}, \dots, Y_k \in \mathbb{S}^{p_k}} f(x, Y) \\ & \text{subject to} \quad \varphi_\pi(g_i(x, Y)) \leq 0, & i = 1, \dots, m_g \\ & \quad \Phi_\Pi(\mathcal{A}_i(x, Y)) \preceq 0, & i = 1, \dots, m_A \\ & \quad \Phi_\Pi(\underline{\lambda}_i I - Y_i) \preceq 0, & i = 1, \dots, k \\ & \quad \Phi_\Pi(Y_i - \bar{\lambda}_i I) \preceq 0, & i = 1, \dots, k \\ & \quad h_i(x, Y) = 0, & i = 1, \dots, m_h, \end{aligned} \quad (2)$$

where we have used the abbreviations $\varphi_\pi = \pi\varphi(\cdot/\pi)$ and $\Phi_\Pi = \Pi\Phi(\cdot/\Pi)$.

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

$$\begin{aligned} F(x, Y, u, \Xi, \underline{U}, \overline{U}, v, \pi, \Pi) \\ = f(x, Y) + \sum_{i=1}^{m_g} u_i \varphi_\pi(g_i(x, Y)) + \sum_{i=1}^{m_A} \langle \Xi_i, \Phi_\Pi(\mathcal{A}_i(x, Y)) \rangle \\ + \sum_{i=1}^k \langle \underline{U}_i, \Phi_\Pi(\underline{\lambda}_i I - Y_i) \rangle + \sum_{i=1}^k \langle \overline{U}_i, \Phi_\Pi(Y_i - \overline{\lambda}_i I) \rangle + v^\top h(x, Y); \quad (3) \end{aligned}$$

here $u \in \mathbb{R}^{m_g}$, $\Xi = (\Xi_1, \dots, \Xi_{m_A})$, $\Xi_i \in \mathbb{S}^{p_{A_i}}$, and $\underline{U} = (\underline{U}_1, \dots, \underline{U}_k)$, $\overline{U} = (\overline{U}_1, \dots, \overline{U}_k)$, $\underline{U}_i, \overline{U}_i \in \mathbb{S}^{p_i}$, are Lagrange multipliers associated with the standard and the matrix inequality constraints, respectively, and $v \in \mathbb{R}^{m_h}$ is the vector of Lagrangian multipliers associated with the equality constraints.

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

Algorithm 1 Let x^1, Y^1 and $u^1, \Xi^1, \underline{U}^1, \overline{U}^1, v^1$ be given. Let $\pi^1 > 0$, $\Pi^1 > 0$ and $\alpha^1 > 0$. For $\ell = 1, 2, \dots$ repeat till a stopping criterium is reached:

- (i) Find $x^{\ell+1}, Y^{\ell+1}$ and $v^{\ell+1}$ such that

$$\begin{aligned} \|\nabla_{x,Y} F(x^{\ell+1}, Y^{\ell+1}, u^\ell, \Xi^\ell, \underline{U}^\ell, \overline{U}^\ell, v^{\ell+1}, \pi^\ell, \Pi^\ell)\| &\leq \alpha^\ell \\ \|h(x^{\ell+1}, Y^{\ell+1})\| &\leq \alpha^\ell \end{aligned}$$
- (ii) $u_i^{\ell+1} = u_i^\ell \varphi_{\pi^\ell}(g_i(x^{\ell+1}, Y^{\ell+1}))$, $i = 1, \dots, m_g$
 $\Xi_i^{\ell+1} = D_{\mathcal{A}} \Phi_{\Pi^\ell}(\mathcal{A}_i(x^{\ell+1}, Y^{\ell+1}); \Xi_i^\ell)$, $i = 1, \dots, m_A$
 $\underline{U}_i^{\ell+1} = D_{\mathcal{A}} \Phi_{\Pi^\ell}((\underline{\lambda}_i I - Y_i^{\ell+1}); \underline{U}_i^\ell)$, $i = 1, \dots, k$
 $\overline{U}_i^{\ell+1} = D_{\mathcal{A}} \Phi_{\Pi^\ell}((Y_i^{\ell+1} - \overline{\lambda}_i I); \overline{U}_i^\ell)$, $i = 1, \dots, k$
- (iii) $\pi^{\ell+1} < \pi^\ell$, $\Pi^{\ell+1} < \Pi^\ell$, $\alpha^{\ell+1} < \alpha^\ell$.

In Step (i) we attempt to find an approximate solution of the following system (in x, Y and v):

$$\begin{aligned} \nabla_{x,Y} F(x, Y, u, \Xi, \underline{U}, \overline{U}, v, \pi, \Pi) &= 0 \\ h(x, Y) &= 0, \end{aligned} \quad (4)$$

where the penalty parameters π, Π , as well as the multipliers $u, \Xi, \underline{U}, \overline{U}$ are fixed. In order to solve it, we apply the damped Newton method. Descent directions are calculated utilizing the MATLAB command `ldl` that is based on the factorization routine MA57, in combination with an inertia correction strategy described in [?]. In the forthcoming release of PENLAB, we will also apply iterative methods, as described in [?]. The step length is derived using an augmented Lagrangian merit function defined as

$$F(x, Y, u, \Xi, \underline{U}, \overline{U}, v, \pi, \Pi) + \frac{1}{2\mu} \|h(x, Y)\|_2^2$$

along with an Armijo rule.

If there are no equality constraints in the problems, the unconstrained minimization in Step (i) is performed by the modified Newton method with line-search (for details, see [?]).

The multipliers calculated in Step (ii) are restricted in order to satisfy:

$$\mu < \frac{u_i^{\ell+1}}{u_i^\ell} < \frac{1}{\mu}$$

with some positive $\mu \leq 1$; by default, $\mu = 0.3$. A similar restriction procedure can be applied to the matrix multipliers $\underline{U}^{\ell+1}$, $\overline{U}^{\ell+1}$ and Ξ ; see again [?] for details.

The penalty parameters π , Π in Step (iii) are updated by some constant factor dependent on the initial penalty parameters π^1 , Π^1 . The update process is stopped when π_{eps} (by default 10^{-6}) is reached.

Algorithm ?? is stopped when a criterion based on the KKT error is satisfied and both of the inequalities holds:

$$\begin{aligned} \frac{|f(x^\ell, Y^\ell) - F(x^\ell, Y^\ell, u^\ell, \Xi^\ell, \underline{U}^\ell, \overline{U}^\ell, v^\ell, \pi^\ell, \Pi^\ell)|}{1 + |f(x^\ell, Y^\ell)|} &< \epsilon \\ \frac{|f(x^\ell, Y^\ell) - f(x^{\ell-1}, Y^{\ell-1})|}{1 + |f(x^\ell, Y^\ell)|} &< \epsilon, \end{aligned}$$

where ϵ is by default 10^{-6} .

3.1. Choice of φ and Φ

To treat the standard NLP constraints, we use the penalty/barrier function proposed by Ben-Tal and Zibulevsky [?]:

$$\varphi_{\bar{\tau}}(\tau) = \begin{cases} \tau + \frac{1}{2} \tau^2 & \text{if } \tau \geq \bar{\tau} \\ -(1 + \bar{\tau})^2 \log\left(\frac{1 + 2\bar{\tau} - \tau}{1 + \bar{\tau}}\right) + \bar{\tau} + \frac{1}{2} \bar{\tau}^2 & \text{if } \tau < \bar{\tau}; \end{cases} \quad (5)$$

by default, $\bar{\tau} = -\frac{1}{2}$.

The penalty function Φ_Π of our choice is defined as follows (here, for simplicity, we omit the variable Y):

$$\Phi_\Pi(\mathcal{A}(x)) = -\Pi^2(\mathcal{A}(x) - \Pi I)^{-1} - \Pi I. \quad (6)$$

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

$$\mathcal{Z}(x) = -(\mathcal{A}(x) - \Pi I)^{-1} \quad (7)$$

we have (see [?]):

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

3.2. Strictly feasible constraints

In certain applications, some of the bound constraints must remain strictly feasible for all iterations because, for instance, the objective function may be undefined at infeasible points (see examples in Section ??). To be able to solve such problems, we treat these inequalities by a classic barrier function. In case of matrix variable inequalities, we split Y in non-strictly feasible matrix variables Y_1 and strictly feasible matrix variables Y_2 , respectively, and define the augmented Lagrangian

$$\tilde{F}(x, Y_1, Y_2, u, \Xi, \underline{U}, \overline{U}, v, \pi, \Pi, \kappa) = F(x, Y_1, u, \Xi, \underline{U}, \overline{U}, v, \pi, \Pi) + \kappa \Phi_{\text{bar}}(Y_2), \quad (8)$$

where Φ_{bar} can be defined, for example for the constraint $Y_2 \succeq 0$, by

$$\Phi_{\text{bar}}(Y_2) = -\log \det(Y_2).$$

Strictly feasible variables x are treated in a similar manner. Note that, while the penalty parameter π may be constant from a certain index $\bar{\ell}$ (see again [?] for details), the barrier parameter κ is required to tend to zero with increasing ℓ .

4. The code

PENLAB is a free open-source MATLAB implementation of the algorithm described above. The main attention was given to clarity of the code rather than tweaks to improve its performance. This should allow users to better understand the code and encourage them to edit and develop the algorithm further. The code is written entirely in MATLAB with an exception of two mex-functions that handles the computationally most intense task of evaluating the second derivative of the Augmented Lagrangian and a sum of multiple sparse matrices (a slower non-mex alternative is provided as well). The solver is implemented as a MATLAB handle class and thus it should be supported on all MATLAB versions starting from R2008a.

PENLAB is distributed under GNU GPL license and can be downloaded from <http://web.mat.bham.ac.uk/kocvara/penlab>. The distribution package includes the full source code and precompiled mex-functions, PENLAB User's Guide and also an internal (programmer's) documentation which can be generated from the source code. Many examples provided in the package show various ways of calling PENLAB and handling NLP-SDP problems.

4.1. Usage

The source code is divided between a class `penlab` which implements Algorithm 1 and handles generic NLP-SDP problems similar to formulation (??) and interface routines providing various specialized inputs to the solver. Some of these are described in Section ??.

The user needs to prepare a MATLAB structure (here called `penm`) which describes the problem parameters, such as number of variables, number of constraints, lower and upper bounds, etc. Some of the fields are shown in Table ??, for a complete list see the PENLAB User's Guide. The structure is passed to `penlab` which returns the initialized problem instance:

```
>> problem = penlab(penm);
```

The solver might be invoked and results retrieved, for example, by calling

```
>> problem.solve()
>> problem.x
```

The point `x` or option settings might be changed and the solver invoked again. The whole object can be cleared from the memory using

```
>> clear problem;
```

Table 1. Selection of fields of the MATLAB structure `penm` used to initialize PENLAB object. Full list is available in PENLAB User's Guide.

field name	meaning
<code>Nx</code>	dimension of vector x
<code>NY</code>	number of matrix variables Y
<code>Y</code>	cell array of length <code>NY</code> with a nonzero pattern of each of the matrix variables
<code>lbY</code>	<code>NY</code> lower bounds on matrix variables (in spectral sense)
<code>ubY</code>	<code>NY</code> upper bounds on matrix variables (in spectral sense)
<code>NANLN</code>	number of nonlinear matrix constraints
<code>NALIN</code>	number of linear matrix constraints
<code>lbA</code>	lower bounds on all matrix constraints
<code>ubA</code>	upper bounds on all matrix constraints

4.2. Callback functions

The principal philosophy of the code is similar to many other optimization codes—we use callback functions (provided by the user) to compute function values and derivatives of all involved functions.

For a generic problem, the user must define nine MATLAB callback functions: `objfun`, `objgrad`, `objhess`, `confun`, `congrad`, `conhess`, `mconfun`, `mcongrad`, `mconhess` for function value, gradient, and Hessian of the objective function, (standard) constraints and matrix constraint. If one constraint type is not present, the corresponding callbacks need not be defined. Let us just show the parameters of the most complex callbacks for the matrix constraints:

```

function [Ak, userdata] = mconfun(x,Y,k,userdata)
function [dAki,userdata] = mcongrad(x,Y,k,i,userdata)
function [ddAki,j, userdata] = mconhess(x,Y,k,i,j,userdata)

```

Here x, Y are the current values of the (vector and matrix) variables. Parameter k stands for the constraint number. Because every element of the gradient and the Hessian of a matrix function is a matrix, we compute them (the gradient and the Hessian) element-wise (parameters i, j). The outputs $A_k, dA_{ki}, ddA_{ki,j}$ are symmetric matrices saved in sparse MATLAB format.

Finally, `userdata` is a MATLAB structure passed through all callbacks for user's convenience and may contain any additional data needed for the evaluations. It is unchanged by the algorithm itself but it can be modified in the callbacks by user. For instance, some time-consuming computation that depends on x, Y, k but is independent of i can be performed only for $i = 1$, the result stored in `userdata` and recalled for any $i > 1$ (see, e.g., Section ??, example Truss Design with Buckling Constraint).

4.3. Mex files

Despite our intentions to use only pure Matlab code, two routines were identified to cause a significant slow-down and therefore their m-files were substituted with equivalent mex-files. The first one computes linear combination of a set of sparse matrices, e.g., when evaluating $A_i(x)$ for polynomial matrix inequalities, and is based on ideas from [?]. The second one evaluates matrix inequality contributions to the Hessian of the augmented Lagrangian (??) when using penalty function (??).

The latter case reduces to computing $z_\ell = \langle T A_k U, A_\ell \rangle$ for $\ell = k, \dots, n$ where $T, U \in \mathbb{S}^m$ are dense and $A_\ell \in \mathbb{S}^m$ are sparse with potentially highly varying densities. Such expressions soon become challenging for nontrivial m and can easily dominate the whole Algorithm ?. Note that the problem is common even in primal-dual interior point methods for SDPs and have been studied in [?]. We developed a relatively simple strategy which can be viewed as an evolution of the three computational formulae presented in [?] and offers a minimal number of multiplications while keeping very modest memory requirements. We refer to it as a *look-ahead strategy with caching*. It can be described as follows:

Algorithm 2 Precompute a set \mathcal{J} of all nonempty columns across all $A_\ell, \ell = k, \dots, n$ and a set \mathcal{I} of nonempty rows of A_k (look-ahead). Reset flag vector $c \leftarrow 0$, set $z = 0$ and $v = w = 0$. For each $j \in \mathcal{J}$ perform:

1. compute selected elements of the j -th column of $A_k U$, i.e.,

$$v_i = \sum_{\alpha=1}^m (A_k)_{i\alpha} U_{\alpha j} \text{ for } i \in \mathcal{I},$$
2. for each A_ℓ with nonempty j -th column go through its nonzero elements $(A_\ell)_{ij}$ and
 - (a) if $c_i < j$ compute $w_i = \sum_{\alpha \in \mathcal{I}} T_{i\alpha} v_\alpha$ and set $c_i \leftarrow j$ (caching),
 - (b) update trace, i.e., $z_\ell = z_\ell + w_i (A_\ell)_{ij}$.

5. Gradients and Hessians of matrix valued functions

There are several concepts of derivatives of matrix functions; they, however, only differ in the ordering of the elements of the resulting “differential”. In PENLAB, we use the following definitions of the gradient and Hessian of matrix valued functions.

Definition 1. Let F be a differentiable $m \times n$ real matrix function of an $p \times q$ matrix of real variables X . The (i, j) -th element of the gradient of F at X is the $m \times n$ matrix

$$[\nabla F(X)]_{ij} := \frac{\partial F(X)}{\partial x_{ij}}, \quad i = 1, \dots, p, j = 1, \dots, q. \quad (9)$$

Definition 2. Let F be a twice differentiable $m \times n$ real matrix function of an $p \times q$ matrix of real variables X . The $(ij, k\ell)$ -th element of the Hessian of F at X is the $m \times n$ matrix

$$[\nabla^2 F(X)]_{ij, k\ell} := \frac{\partial^2 F(X)}{\partial x_{ij} \partial x_{k\ell}}, \quad i, k = 1, \dots, p, j, \ell = 1, \dots, q. \quad (10)$$

In other words, for every pair of variables x_{ij} , $x_{k\ell}$, elements of X , the second partial derivative of $F(X)$ with respect to these variables is the $m \times n$ matrix $\frac{\partial^2 F(X)}{\partial x_{ij} \partial x_{k\ell}}$.

How to compute these derivatives, i.e., how to define the callback functions? In Appendix A, we summarize basic formulas for the computation of derivatives of scalar and matrix valued functions of matrices.

For low-dimensional problems, the user can utilize MATLAB’s Symbolic Toolbox. For instance, for $F(X) = XX$, the commands

```
>> A=sym('X',[2,2]);
>> J=jacobian(X*X,X(:));
>> H=jacobian(J,X(:));
```

generate arrays J and H such that the i -th column of J is the vectorized i -th element of the gradient of $F(X)$; similarly, the k -th column of H , $k = (i-1)n^2 + j$ for $i, j = 1, \dots, n^2$ is the vectorized (i, j) -th element of the Hessian of $F(X)$. Clearly, the dimension of the matrix variable is fixed and for a different dimension we have to generate new formulas. Unfortunately, this approach is useless for higher dimensional matrices (the user is invited to use the above commands for $F(X) = X^{-1}$ with $X \in \mathbb{S}^5$ to see the difficulties). However, one can always use symbolic computation to check validity of general dimension independent formulas on small dimensional problems.

6. Pre-programmed interfaces

PENLAB distribution contains several pre-programmed interfaces for standard optimization problems with standard inputs. For these problems, the user does not have to create the `penm` object, nor the callback functions.

6.1. Nonlinear optimization with AMPL input

PENLAB can read optimization problems that are defined in and processed by AMPL [?]. AMPL contains routines for automatic differentiation, hence the gradients and Hessians in the callbacks reduce to calls to appropriate AMPL routines.

Assume that nonlinear optimization problem is processed by AMPL, so that we have the corresponding `.nl` file, for instance `chain.nl`, stored in directory `datafiles`. All the user has to do to solve the problem is to call the following three commands:

```
>> penm = nlp_define('datafiles/chain100.nl');
>> problem = penlab(penm);
>> problem.solve();
```

6.2. Linear semidefinite programming

Assume that the data of a linear SDP problem is stored in a MATLAB structure `sdpdata`. Alternatively, such a structure can be created by the user from SDPA input file [?]. For instance, to read problem `arch0.dat-s` stored in directory `datafiles`, call

```
>> sdpdata = readsdp('datafiles/control11.dat-s');
```

To solve the problem by PENLAB, the user just has to call the following sequence of commands:

```
>> penm = sdp_define(sdpdata);
>> problem = penlab(penm);
>> problem.solve();
```

6.3. Bilinear matrix inequalities

We want to solve an optimization problem with quadratic objective and constraints in the form of bilinear matrix inequalities:

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & \frac{1}{2} x^T H x + c^T x \\ \text{subject to} \quad & b_{\text{low}} \leq Bx \leq b_{\text{up}} \\ & Q_0^i + \sum_{k=1}^n x_k Q_k^i + \sum_{k=1}^n \sum_{\ell=1}^n x_k x_\ell Q_{k\ell}^i \succcurlyeq 0, \quad i = 1, \dots, m. \end{aligned} \tag{11}$$

The problem data should be stored in a simple format explained in PENLAB User's Guide. All the user has to do to solve the problem is to call the following sequence of commands:

```
>> load datafiles/bmi_example;
>> penm = bmi_define(bmidata);
>> problem = penlab(penm);
>> problem.solve();
```

6.4. Polynomial matrix inequalities

We want to solve an optimization problem with constraints in the form of polynomial matrix inequalities:

$$\begin{aligned} & \min_{x \in \mathbb{R}^n} \frac{1}{2} x^T H x + c^T x \\ & \text{subject to } b_{\text{low}} \leq Bx \leq b_{\text{up}} \\ & \mathcal{A}_i(x) \succcurlyeq 0, \quad i = 1, \dots, m \end{aligned} \quad (12)$$

with

$$\mathcal{A}_i(x) = \sum_j x^{(\kappa^i(j))} Q_j^i$$

where $\kappa^i(j)$ is a multi-index of the i -th constraint with possibly repeated entries and $x^{(\kappa^i(j))}$ is a product of elements with indices in $\kappa^i(j)$.

For example, for

$$\mathcal{A}(x) = Q_1 + x_1 x_3 Q_2 + x_2 x_4^3 Q_3$$

the multi-indices are $\kappa(1) = \{0\}$ (Q_1 is an absolute term), $\kappa(2) = \{1, 3\}$ and $\kappa(3) = \{2, 4, 4, 4\}$.

Assuming now that the problem is stored in a structure `pmidata` (as explained in PENLAB User's Guide), the user just has to call the following sequence of commands:

```
>> load datafiles/pmi_example;
>> penm = pmi_define(pmidata);
>> problem = penlab(penm);
>> problem.solve();
```

7. Examples

All MATLAB programs and data related to the examples in this section can be found in directories `examples` and `applications` of the PENLAB distribution.

7.1. Correlation matrix with the constrained condition number

We consider the problem of finding the nearest correlation matrix ([?]):

$$\begin{aligned} & \min_X \sum_{i,j=1}^n (X_{ij} - H_{ij})^2 \\ & \text{subject to} \\ & X_{ii} = 1, \quad i = 1, \dots, n \\ & X \succeq 0. \end{aligned} \quad (13)$$

In addition to this standard setting of the problem, let us bound the condition number of the nearest correlation matrix by adding the constraint

$$\text{cond}(X) = \kappa.$$

We can formulate this constraint as

$$I \preceq \tilde{X} \preceq \kappa I \quad (14)$$

the variable transformation

$$\tilde{X} = \zeta X.$$

After the change of variables, and with the new constraint, the problem of finding the nearest correlation matrix with a given condition number reads as follows:

$$\begin{aligned} \min_{\zeta, \tilde{X}} \quad & \sum_{i,j=1}^n \left(\frac{1}{\zeta} \tilde{X}_{ij} - H_{ij} \right)^2 \\ \text{subject to} \quad & \tilde{X}_{ii} - \zeta = 0, \quad i = 1, \dots, n \\ & I \preceq \tilde{X} \preceq \kappa I \end{aligned} \quad (15)$$

The new problem now has the NLP-SDP structure of (??).

We will consider an example based on a practical application from finances; see [?]. Assume that we are given a 5×5 correlation matrix. We now add a new asset class, that means, we add one row and column to this matrix. The new data is based on a different frequency than the original part of the matrix, which means that the new matrix is no longer positive definite:

$$H_{\text{ext}} = \begin{pmatrix} 1 & -0.44 & -0.20 & 0.81 & -0.46 & -0.05 \\ -0.44 & 1 & 0.87 & -0.38 & 0.81 & -0.58 \\ -0.20 & 0.87 & 1 & -0.17 & 0.65 & -0.56 \\ 0.81 & -0.38 & -0.17 & 1 & -0.37 & -0.15 \\ -0.46 & 0.81 & 0.65 & -0.37 & 1 & -0.08 \\ -0.05 & -0.58 & -0.56 & -0.15 & 0.08 & 1 \end{pmatrix}.$$

When solving problem (??) by PENLAB with $\kappa = 10$, we get the solution after 11 outer and 37 inner iterations. The optimal value of ζ is 3.4886 and, after the back substitution $X = \frac{1}{\zeta} \tilde{X}$, we get the nearest correlation matrix

$$X = \begin{pmatrix} 1.0000 & -0.3775 & -0.2230 & 0.7098 & -0.4272 & -0.0704 \\ -0.3775 & 1.0000 & 0.6930 & -0.3155 & 0.5998 & -0.4218 \\ -0.2230 & 0.6930 & 1.0000 & -0.1546 & 0.5523 & -0.4914 \\ 0.7098 & -0.3155 & -0.1546 & 1.0000 & -0.3857 & -0.1294 \\ -0.4272 & 0.5998 & 0.5523 & -0.3857 & 1.0000 & -0.0576 \\ -0.0704 & -0.4218 & -0.4914 & -0.1294 & -0.0576 & 1.0000 \end{pmatrix}$$

with eigenvalues

$$\text{eigenvals} = \begin{pmatrix} 0.2866 & 0.2866 & 0.2867 & 0.6717 & 1.6019 & 2.8664 \end{pmatrix}$$

and the condition number equal to 10, indeed.

Gradients and Hessians What are the first and second partial derivatives of functions involved in problem (??)? The constraint is linear, so the answer is trivial here, and we can only concentrate on the objective function

$$f(z, \tilde{X}) := \sum_{i,j=1}^n (z\tilde{X}_{ij} - H_{ij})^2 = \langle z\tilde{X} - H, z\tilde{X} - H \rangle, \quad (16)$$

where, for convenience, we introduced a variable $z = \frac{1}{\zeta}$.

Theorem 1. Let x_{ij} and h_{ij} , $i, j = 1, \dots, n$ be elements of \tilde{X} and H , respectively. For the function f defined in (??) we have the following partial derivatives:

- (i) $\nabla_z f(z, \tilde{X}) = 2\langle \tilde{X}, z\tilde{X} - H \rangle$
- (ii) $\left[\nabla_{\tilde{X}} f(z, \tilde{X}) \right]_{ij} = 2z(zx_{ij} - h_{ij}), \quad i, j = 1, \dots, n$
- (iii) $\nabla_{z,z}^2 f(z, \tilde{X}) = 2\langle \tilde{X}, \tilde{X} \rangle$
- (iv) $\left[\nabla_{z,\tilde{X}}^2 f(z, \tilde{X}) \right]_{ij} = \left[\nabla_{\tilde{X},z}^2 f(z, \tilde{X}) \right]_{ij} = 4zx_{ij} - 2h_{ij}, \quad i, j = 1, \dots, n$
- (v) $\left[\nabla_{\tilde{X},\tilde{X}}^2 f(z, \tilde{X}) \right]_{ij,k\ell} = 2z^2 \text{ for } i = k, j = \ell \text{ and zero otherwise } (i, j, k, \ell = 1, \dots, n).$

The proof follows directly from formulas in Appendix A.

PENLAB distribution This problem is stored in directory `applications/CorrMat` of the PENLAB distribution. To solve the above example and to see the resulting eigenvalues of X , run in its directory

```
>> penm = corr_define;
>> problem = penlab(penm);
>> problem.solve();
>> eig(problem.Y{1}*problem.x)
```

7.2. Truss topology optimization with stability constraints

In truss optimization we want to design a pin-jointed framework consisting of m slender bars of constant mechanical properties characterized by their Young's modulus E . We will consider trusses in a d -dimensional space, where $d = 2$ or $d = 3$. The bars are jointed at \tilde{n} nodes. The system is under load, i.e., forces $f_j \in \mathbb{R}^d$ are acting at some nodes j . They are aggregated in a vector f , where we put $f_j = 0$ for nodes that are not under load. This external load is transmitted along the bars causing displacements of the nodes that make up the displacement vector u . Let p be the number of fixed nodal coordinates, i.e., the number of components with prescribed discrete homogeneous Dirichlet boundary condition. We omit these fixed components from the problem formulation reducing thus the dimension of u to

$$n = d \cdot \tilde{n} - p.$$

Analogously, the external load f is considered as a vector in \mathbb{R}^n .

The design variables in the system are the bar volumes x_1, \dots, x_m . Typically, we want to minimize the weight of the truss. We assume to have a unique material (and thus density) for all bars, so this is equivalent to minimizing the volume of the truss, i.e., $\sum_{i=1}^m x_i$. The optimal truss should satisfy mechanical equilibrium conditions:

$$K(x)u = f; \quad (17)$$

here

$$K(x) := \sum_{i=1}^m x_i K_i, \quad K_i = \frac{E_i}{\ell_i^2} \gamma_i \gamma_i^\top \quad (18)$$

is the so-called stiffness matrix, E_i the Young modulus of the i th bar, ℓ_i its length and γ_i the n -vector of direction cosines.

We further introduce the compliance of the truss $f^\top u$ that indirectly measures the stiffness of the structure under the force f and impose the constraints

$$f^\top u \leq \gamma.$$

This constraint, together with the equilibrium conditions, can be formulated as a single linear matrix inequality ([?])

$$\begin{pmatrix} K(x) & f \\ f^\top & \gamma \end{pmatrix} \succeq 0.$$

The minimum volume single-load truss topology optimization problem can then be formulated as a linear semidefinite program:

$$\begin{aligned} \min_{x \in \mathbb{R}^m} \quad & \sum_{i=1}^m x_i \\ \text{subject to} \quad & \begin{pmatrix} K(x) & f \\ f^\top & \gamma \end{pmatrix} \succeq 0 \\ & x_i \geq 0, \quad i = 1, \dots, m. \end{aligned} \quad (19)$$

We further consider the constraint on the global stability of the truss. The meaning of the constraint is to avoid global buckling of the optimal structure. We consider the simplest formulation of the buckling constraint based on the so-called linear buckling assumption [?]. As in the case of free vibrations, we need to constrain eigenvalues of the generalized eigenvalue problem

$$K(x)w = \lambda G(x)w, \quad (20)$$

in particular, we require that all eigenvalues of (??) lie outside the interval $[0,1]$. The so-called geometry stiffness matrix $G(x)$ depends, this time, nonlinearly on the design variable x :

$$G(x) = \sum_{i=1}^m G_i(x), \quad G_i(x) = \frac{E x_i}{\ell_i^d} (\gamma_i^\top K(x)^{-1} f) (\delta_i \delta_i^\top + \eta_i \eta_i^\top). \quad (21)$$

Vectors δ, η are chosen so that γ, δ, η are mutually orthogonal. (The presented formula is for $d = 3$. In the two-dimensional setting the vector η is not present.) To simplify the notation, we denote

$$\Delta_i = \delta_i \delta_i^T + \eta_i \eta_i^T.$$

It was shown in [?] that the eigenvalue constraint can be equivalently written as a nonlinear matrix inequality

$$K(x) + G(x) \succcurlyeq 0 \quad (22)$$

that is now to be added to (??) to get the following nonlinear semidefinite programming problem. Note that x_i are requested to be strictly feasible.

$$\min_{x \in \mathbb{R}^m} \sum_{i=1}^m x_i \quad (23)$$

subject to

$$\begin{pmatrix} K(x) & f \\ f^T & \gamma \end{pmatrix} \succeq 0$$

$$K(x) + G(x) \succcurlyeq 0$$

$$x_i > 0, \quad i = 1, \dots, m$$

Gradients and Hessians Let $M : \mathbb{R}^m \rightarrow \mathbb{R}^{n \times n}$ be a matrix valued function assigning each vector ξ a matrix $M(\xi)$. We denote by $\nabla_k M$ the partial derivative of $M(\xi)$ with respect to the k -th component of vector ξ .

Lemma 1 (based on [?]). *Let $M : \mathbb{R}^m \rightarrow \mathbb{R}^{n \times n}$ be a symmetric matrix valued function assigning each $\xi \in \mathbb{R}^m$ a nonsingular ($n \times n$) matrix $M(\xi)$. Then (for convenience we omit the variable ξ)*

$$\nabla_k M^{-1} = -M^{-1}(\nabla_k M)M^{-1}.$$

If M is a linear function of ξ , i.e., $M(\xi) = \sum_{i=1}^m \xi_i M_i$ with symmetric positive semidefinite $M_i, i = 1, \dots, m$, then the above formula simplifies to

$$\nabla_k M^{-1} = -M^{-1} M_k M^{-1}.$$

Theorem 2 ([?]). *Let $G(x)$ be given as in (??). Then*

$$[\nabla G]_k = \frac{E}{\ell_k^3} \gamma_k^T K^{-1} f \Delta_k - \sum_{j=1}^m \frac{E t_j}{\ell_j^3} \gamma_j^T K^{-1} K_k K^{-1} f \Delta_j$$

and

$$\begin{aligned} [\nabla^2 G]_{k\ell} = & -\frac{E}{\ell_k^3} \gamma_k^T K^{-1} K_\ell K^{-1} f \Delta_k - \frac{E}{\ell_\ell^3} \gamma_\ell^T K^{-1} K_k K^{-1} f \Delta_\ell \\ & - \sum_{j=1}^m \frac{E t_j}{\ell_j^3} \gamma_j^T K^{-1} K_\ell K^{-1} K_k K^{-1} f \Delta_j \\ & - \sum_{j=1}^m \frac{E t_j}{\ell_j^3} \gamma_j^T K^{-1} K_k K^{-1} K_\ell K^{-1} f \Delta_j. \end{aligned}$$

Example Consider the standard example of a laced column under axial loading (example `tim` in the PENLAB collection). Due to symmetry, we only consider one half of the column, as shown in Figure ??(top-left); it has 19 nodes and 42 potential bars, so $n = 34$ and $m = 42$. The column dimensions are 8.5×1 , the two nodes on the left-hand side are fixed and the “axial” load applied at the column tip is $(0, -10)$. The upper bound on the compliance is chosen as $\gamma = 1$.

Assume first that $x_i = 0.425, i = 1, \dots, m$, i.e., the volumes of all bars are equal and the total volume is 17.85. The values of x_i were chosen such that the truss satisfies the compliance constraint: $f^\top u = 0.9923 \leq \gamma$. For this truss, the smallest nonnegative eigenvalue of (??) is equal to 0.7079 and the buckling constraint (??) is not satisfied. Figure ??(top-right) shows the corresponding buckling mode (eigenvector associated with this eigenvalue). Let us now solve the truss optimization problem *without* the

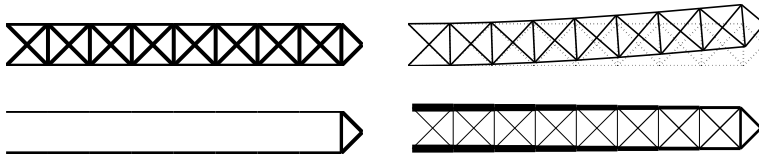


Fig. 1. Truss optimization with stability problem: initial truss (top-left); its buckling mode (top-right); optimal truss without stability constraint (bottom-left); and optimal stable truss (bottom-right)

stability constraint (??). We obtain the design shown in Figure ??(bottom-left). This truss is much lighter than the original one ($\sum_{i=1}^m x_i = 9.388$), it is, however, extremely unstable under the given load, as (??) has a zero eigenvalue.

When solving the truss optimization problem *with* the stability constraint (??) by PENLAB, we obtain the design shown in Figure ??(bottom-right). This truss is still significantly lighter than the original one ($\sum_{i=1}^m x_i = 12.087$), but it is now stable under the given load. To solve the nonlinear SDP problem, PENLAB needed 18 global and 245 Newton iterations and 212 seconds of CPU time, 185 of which were spent in the Hessian evaluation routines.

PENLAB distribution Directories `applications/TTO` and `applications/TTObuckling` of the PENLAB distribution contain the problem formulation and many examples of trusses. To solve the above example with the buckling constraint, run

```
>> solve_ttob('GEO/tim.geo')
```

in directory `TTObuckling`.

7.3. Static output feedback

Given a linear system with $A \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times m}$, $C \in \mathbb{R}^{p \times n}$

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx\end{aligned}$$

we want to stabilize it by static output feedback $u = Ky$. That is, we want to find a matrix $K \in \mathbb{R}^{m \times p}$ such that the eigenvalues of the closed-loop system $A + BKC$ belong to the left half-plane.

The standard way how to treat this problem is based on the Lyapunov stability theory. It says that $A + BKC$ has all its eigenvalues in the open left half-plane if and only if there exists a symmetric positive definite matrix P such that

$$(A + BKC)^T P + P(A + BKC) \prec 0. \quad (24)$$

Hence, by introducing the new variable, the Lyapunov matrix P , we can formulate the SOF problem as a feasibility problem for the bilinear matrix inequality (??) in variables K and P . As typically $n > p, m$ (often $n \gg p, m$), the Lyapunov variable dominates here, although it is just an auxiliary variable and we do not need to know its value at the feasible point. Hence a natural question arises whether we can avoid the Lyapunov variable in the formulation of the problem. The answer was given in [?] and lies in the formulation of the problem using polynomial matrix inequalities.

Let $k = \text{vec } K$. Define the characteristic polynomial of $A + BKC$:

$$q(s, k) = \det(sI - A - BKC) = \sum_{i=0}^n q_i(k) s^i,$$

where $q_i(k) = \sum_{\alpha} q_{i\alpha} k^{\alpha}$ and $\alpha \in \mathbb{N}^{mp}$ are all monomial powers. The *Hermite stability criterion* says that the roots of $q(s, k)$ belong to the stability region D (in our case the left half-plane) if and only if

$$H(q) = \sum_{i=0}^n \sum_{j=0}^n q_i(k) q_j(k) H_{ij} \succ 0.$$

Here the coefficients H_{ij} depend on the stability region only (see, e.g., [?]). For instance, for $n = 3$, we have

$$H(q) = \begin{pmatrix} 2q_0q_1 & 0 & 2q_0q_3 \\ 0 & 2q_1q_2 - 2q_0q_3 & 0 \\ 2q_0q_3 & 0 & 2q_2q_3 \end{pmatrix}.$$

The Hermite matrix $H(q) = H(k)$ depends polynomially on k :

$$H(k) = \sum_{\alpha} H_{\alpha} k^{\alpha} \succ 0 \quad (25)$$

where $H_{\alpha} = H_{\alpha}^T \in \mathbb{R}^{n \times n}$ and $\alpha \in \mathbb{N}^{mp}$ describes all monomial powers.

Theorem 3 ([?]). *Matrix K solves the static output feedback problem if and only if $k = \text{vec } K$ satisfies the polynomial matrix inequality (??).*

In order to solve the strict feasibility problem (??), we can solve the following optimization problem with a polynomial matrix inequality

$$\begin{aligned} \max_{k \in \mathbb{R}^{mp}, \lambda \in \mathbb{R}} \quad & \lambda - \mu \|k\|^2 \\ \text{subject to} \quad & H(k) \succ \lambda I. \end{aligned} \quad (26)$$

Here $\mu > 0$ is a parameter that allows us to trade off between feasibility of the PMI and a moderate norm of the matrix K , which is generally desired in practice.

COMPlib examples In order to use PENLAB for the solution of SOF problems (??), we have developed an interface to the problem library COMPlib [?]¹. Table ?? presents the results of our numerical tests. We have only solved COMPlib problems of small size, with $n < 10$ and $mp < 20$. The reason for this is that our MATLAB implementation of the interface (building the matrix $H(k)$ from COMPlib data) is very time-consuming. For each COMPlib problem, the table shows the degree of the matrix polynomial, problem dimensions n and mp , the optimal λ (the negative largest eigenvalue of the matrix K), the CPU time and number of Newton iterations/linesearch steps of PENLAB. The final column contains information about the solution quality. “F” means failure of PENLAB to converge to an optimal solution. The plus sign “+” means that PENLAB converged to a solution which does not stabilize the system and “0” is used when PENLAB converged to a solution that is on the boundary of the feasible domain and thus not useful for stabilization. The reader can see that PENLAB can solve all problems apart from AC7, NN10, NN13 and NN14; these problems are, however, known to be very ill-conditioned and could not be solved via the Lyapunov matrix approach either (see [?]). Notice that the largest problems with polynomials of degree up to 8 did not cause any major difficulties to the algorithm.

PENLAB distribution The related MATLAB programs are stored in directory `applications/SOF` of the PENLAB distribution. To solve, for instance, example AC1, run

```
>> sof('AC1');
```

COMPlib program and library must be installed on user’s computer.

8. PENLAB versus PENNON (MATLAB versus C)

The obvious concern of any user will be, how fast (or better, how slow) is the MATLAB implementation and if it can solve any problems of non-trivial size. The purpose of this section is to give a very rough comparison of PENLAB and PENNON, i.e., the MATLAB and C implementation of the same algorithm. The reader should, however, not make any serious conclusion from the tables below, for the following reasons:

¹ The authors would like to thank Didier Henrion, LAAS-CNRS Toulouse, for developing a substantial part of this interface.

Table 2. mmm

Problem	degree	n	mp	λ_{opt}	CPU (sec)	iter	remark
AC1	5	5	9	$-0.871 \cdot 10^0$	2.2	27/30	
AC2	5	5	9	$-0.871 \cdot 10^0$	2.3	27/30	
AC3	4	5	8	$-0.586 \cdot 10^0$	1.8	37/48	
AC4	2	4	2	$0.245 \cdot 10^{-2}$	1.9	160/209	+
AC6	4	7	8	$-0.114 \cdot 10^4$	1.2	22/68	
AC7	2	9	2	$-0.102 \cdot 10^3$	0.9	26/91	
AC8	2	9	5	$0.116 \cdot 10^0$	3.9	346/1276	F
AC11	4	5	8	$-0.171 \cdot 10^5$	2.3	65/66	
AC12	6	4	12	$0.479 \cdot 10^0$	12.3	62/73	+
AC15	4	4	6	$-0.248 \cdot 10^{-1}$	1.2	25/28	
AC16	4	4	8	$-0.248 \cdot 10^{-1}$	1.2	23/26	
AC17	2	4	2	$-0.115 \cdot 10^2$	1.0	19/38	
HE1	2	4	2	$-0.686 \cdot 10^2$	1.0	22/22	
HE2	4	4	4	$-0.268 \cdot 10^0$	1.6	84/109	
HE5	4	8	8	$0.131 \cdot 10^2$	1.9	32/37	+
REA1	4	4	6	$-0.726 \cdot 10^2$	1.4	33/35	
REA2	4	4	4	$-0.603 \cdot 10^2$	1.3	34/58	
DIS1	8	8	16	$-0.117 \cdot 10^2$	137.6	30/55	
DIS2	4	3	4	$-0.640 \cdot 10^1$	1.6	59/84	
DIS3	8	6	16	$-0.168 \cdot 10^2$	642.3	66/102	
MFP	3	4	6	$-0.370 \cdot 10^{-1}$	1.0	20/21	
TF1	4	7	8	$-0.847 \cdot 10^{-8}$	1.7	27/31	0
TF2	4	7	6	$-0.949 \cdot 10^{-7}$	1.3	19/23	0
TF3	4	7	6	$-0.847 \cdot 10^{-8}$	1.6	28/38	0
PSM	4	7	6	$-0.731 \cdot 10^2$	1.1	17/39	
NN1	2	3	2	$-0.131 \cdot 10^0$	1.2	32/34	0
NN3	2	4	1	$0.263 \cdot 10^2$	1.0	31/36	+
NN4	4	4	6	$-0.187 \cdot 10^2$	1.2	33/47	
NN5	2	7	2	$0.137 \cdot 10^2$	1.5	108/118	+
NN8	3	3	4	$-0.103 \cdot 10^1$	1.0	19/29	
NN9	4	5	6	$0.312 \cdot 10^1$	1.6	64/97	+
NN10	6	8	9	$0.409 \cdot 10^4$	18.3	300/543	F
NN12	4	6	4	$0.473 \cdot 10^1$	1.4	47/58	+
NN13	4	6	4	$0.279 \cdot 10^{12}$	2.2	200/382	F
NN14	4	6	4	$0.277 \cdot 10^{12}$	2.3	200/382	F
NN15	3	3	4	$-0.226 \cdot 10^0$	1.0	15/14	
NN16	7	8	16	$-0.623 \cdot 10^3$	613.3	111/191	
NN17	2	3	2	$0.931 \cdot 10^{-1}$	1.0	25/26	+

- Both implementations slightly differ. This can be seen on the different numbers of iterations needed to solve single examples.
- The difference in CPU timing very much depends on the type of the problem. For instance, some problems require multiplications of sparse matrices with dense ones—in this case, the C implementation will be much faster. On the other hand, for some problems most of the CPU time is spent in the dense Cholesky factorization which, in both implementations, relies on LAPACK routines and thus the running time may be comparable.
- The problems were solved using an Intel i7 processor with two cores. The MATLAB implementation used both cores to perform *some* commands, while the C implementation only used one core. This is clearly seen, e.g., example lame_emd10 in Table ??.

- For certain problems (such as mater2 in Table ??), most of the CPU time of PENLAB is spent in the user defined routine for gradient evaluation. For linear SDP, this only amounts to reading the data matrices, in our implementation elements of a two-dimensional cell array, from memory. Clearly, a more sophisticated implementation would improve the timing.

For all calculations, we have used a notebook running Windows 7 (32 bit) on Intel Core i7 CPU M620@2.67GHz with 4GB memory and MATLAB 7.7.0.

8.1. Nonlinear programming problems

We first solved selected examples from the COPS collection [?] using AMPL interface. These are medium size examples mostly coming from finite element discretization of optimization problems with PDE constraints. Table ?? presents the results.

Table 3. Selected COPS examples. CPU time is given in seconds. Iteration count gives the number of the global iterations in Algorithm ?? and the total number of steps of the Newton method.

problem	vars	constr.	constraint type	PENNON		PENLAB	
				CPU	iter.	CPU	iter.
elec200	600	200	=	40	81/224	31	43/135
chain800	3199	2400	=	1	14/23	6	24/56
pinene400	8000	7995	=	1	7/7	11	17/17
channel800	6398	6398	=	3	3/3	1	3/3
torsion100	5000	10000	<	1	17/17	17	26/26
bearing100	5000	5000	<	1	17/17	13	36/36
lane_emd10	4811	21	<	217	30/86	64	25/49
dirichlet10	4491	21	<	151	33/71	73	32/68
henon10	2701	21	<	57	49/128	63	76/158
minsurf100	5000	5000	box	1	20/20	97	203/203
gasoil400	4001	3998	= & box	3	34/34	13	59/71
duct15	2895	8601	= & <	6	19/19	9	11/11
tri_turtle	3578	3968	< & box	3	49/49	4	17/17
marine400	6415	6392	< & box	2	39/39	22	35/35
steering800	3999	3200	< & box	1	9/9	7	19/40
methanol400	4802	4797	< & box	2	24/24	16	47/67
catmix400	4398	3198	< & box	2	59/61	15	44/44

8.2. Linear semidefinite programming problems

We solved selected problems from the SDPLIB collection (Table ??) and Topology Optimization collection (Table ??); see [?,?]. The data of all problems were stored in SDPA input files [?]. Instead of PENNON, we have used its clone PENSDP that directly reads the SDPA files and thus avoid repeated calls of the call back functions. The difference between PENNON and PENSDP (in favour of PENSDP) would only be significant in the mater2 example with many small matrix constraints.

Table 4. Selected SDPLIB examples. CPU time is given in seconds. Iteration count gives the number of the global iterations in Algorithm ?? and the total number of steps of the Newton method.

problem	vars	constr.	constr. size	PENSDP		PENLAB	
				CPU	iter.	CPU	iter.
control3	136	2	30	1	19/103	20	22/315
maxG11	800	1	1600	18	22/41	186	18/61
qpG11	800	1	1600	43	22/43	602	18/64
ss30	132	1	294	20	23/112	17	12/63
theta3	1106	1	150	11	15/52	61	14/48

Table 5. Selected TOPO examples. CPU time is given in seconds. Iteration count gives the number of the global iterations in Algorithm ?? and the total number of steps of the Newton method.

problem	vars	constr.	constr. size	PENSDP		PENLAB	
				CPU	iter.	CPU	iter.
buck2	144	2	97	2	23/74	22	18/184
vibra2	144	2	97	2	34/132	35	20/304
shmup2	200	2	441	65	24/99	172	26/179
mater2	423	94	11	2	20/89	70	12/179

A. Appendix: Differential calculus for functions of symmetric matrices

Matrix differential calculus—derivatives of functions depending on matrices—is a topic covered in several papers; see, e.g., [?, ?, ?, ?] and the book [?]. The notation and the very definition of the derivative differ in these papers. Hence, for reader’s convenience, we will give a basic overview of the calculus for some typical (in semidefinite optimization) functions of matrices.

For a matrix X (whether symmetric or not), let x_{ij} denote its (i, j) -th element. Let further E_{ij} denote a matrix with all elements zero except for a unit element in the i -th row and j -th column (the dimension of E_{ij} will be always clear from the context). Our differential formulas are based on Definitions ?? and ??, hence we only need to find the partial derivative of a function $F(X)$, whether matrix or scalar valued, with respect to a single element x_{ij} of X .

A.1. Matrix valued functions

Let F be a differentiable $m \times n$ real matrix function of an $p \times q$ matrix of real variables X . Table ?? gives partial derivatives of $F(X)$ with respect to x_{ij} , $i = 1, \dots, p$, $j = 1, \dots, q$ for some most common functions. In this table, E_{ij} is always of the same dimension as X . To compute other derivatives, we may use the following result on the chain rule.

Theorem 4. *Let F be a differentiable $m \times n$ real matrix function of an $p \times q$ matrix Y that itself is a differentiable function G of an $s \times t$ matrix of real variables X , that is $F(Y) = F(G(X))$. Then*

$$\frac{\partial F(G(X))}{\partial x_{ij}} = \sum_{k=1}^p \sum_{\ell=1}^q \frac{\partial F(Y)}{\partial y_{k\ell}} \frac{\partial [G(X)]_{k\ell}}{\partial x_{ij}}. \quad (27)$$

Table 6.

$F(X)$	$\frac{\partial F(X)}{\partial x_{ij}}$	Conditions
X	E_{ij}	
X^T	E_{ji}	
AX	AE_{ij}	$A \in \mathbb{R}^{m \times p}$
XA	$E_{ij}A$	$A \in \mathbb{R}^{m \times p}$
XX	$E_{ij}X + XE_{ij}$	
$X^T X$	$E_{ji}X + X^T E_{ij}$	
XX^T	$E_{ij}X^T + XE_{ji}$	
X^s	$\sum_{k=1}^s X^{k-1} E_{ij} X^{s-k}$	X square, $s = 1, 2, \dots$
X^{-1}	$-X^{-1} E_{ij} X^{-1}$	X nonsingular

In particular, we have

$$\frac{\partial(G(X)H(X))}{\partial x_{ij}} = \frac{\partial G(X)}{\partial x_{ij}} H(X) + G(X) \frac{\partial H(X)}{\partial x_{ij}} \quad (28)$$

$$\frac{\partial(G(X))^{-1}}{\partial x_{ij}} = -(G(X))^{-1} \frac{\partial G(X)}{\partial x_{ij}} (G(X))^{-1}. \quad (29)$$

We finish this section with the all important theorem on derivatives of functions of *symmetric* matrices.

Theorem 5. Let F be a differentiable $n \times n$ real matrix function of a symmetric $m \times m$ matrix of real variables X . Denote Z_{ij} be the (i, j) -th element of the gradient of $F(X)$ computed by the general formulas in Table ?? and Theorem ?. Then

$$[\nabla F(X)]_{i,i} = Z_{ii}$$

and

$$[\nabla F(X)]_{i,j} = Z_{ij} + Z_{ji} \quad \text{for } i \neq j.$$

Example Let $X = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix}$ and $F(X) = X^2 = \begin{pmatrix} x_{11}^2 + x_{12}x_{21} & x_{11}x_{12} + x_{12}x_{22} \\ x_{11}x_{21} + x_{21}x_{22} & x_{12}x_{21} + x_{22}^2 \end{pmatrix}$.

Then

$$\nabla F(X) = \begin{bmatrix} \begin{pmatrix} 2x_{11} & x_{12} \\ x_{21} & 0 \end{pmatrix} & \begin{pmatrix} x_{21} & x_{11} + x_{22} \\ 0 & x_{21} \end{pmatrix} \\ \begin{pmatrix} x_{12} & 0 \\ x_{11} + x_{22} & x_{12} \end{pmatrix} & \begin{pmatrix} 0 & x_{12} \\ x_{21} & 2x_{22} \end{pmatrix} \end{bmatrix}$$

(a 2×2 array of 2×2 matrices). If we now assume that X is symmetric, i.e. $x_{12} = x_{21}$, we get

$$\nabla F(X) = \begin{bmatrix} \begin{pmatrix} 2x_{11} & x_{21} \\ x_{21} & 0 \end{pmatrix} & \begin{pmatrix} 2x_{21} & x_{11} + x_{22} \\ x_{11} + x_{22} & 2x_{21} \end{pmatrix} \\ \begin{pmatrix} 2x_{21} & x_{11} + x_{22} \\ x_{11} + x_{22} & 2x_{21} \end{pmatrix} & \begin{pmatrix} 0 & x_{21} \\ x_{21} & 2x_{22} \end{pmatrix} \end{bmatrix}.$$

We can see that we could obtain the gradient for the symmetric matrix using the general formula in Table ?? together with Theorem ??.

Notice that if we simply replaced each x_{12} in $\nabla F(X)$ by x_{21} (assuming symmetry of X), we would get an *incorrect* result

$$\nabla F(X) = \begin{bmatrix} \begin{pmatrix} 2x_{11} & x_{21} \\ x_{21} & 0 \end{pmatrix} & \begin{pmatrix} x_{21} & x_{11} + x_{22} \\ 0 & x_{21} \end{pmatrix} \\ \begin{pmatrix} x_{21} & 0 \\ x_{11} + x_{22} & x_{21} \end{pmatrix} & \begin{pmatrix} 0 & x_{21} \\ x_{21} & 2x_{22} \end{pmatrix} \end{bmatrix}.$$

A.2. Scalar valued functions

Table ?? shows derivatives of some most common scalar valued functions of an $m \times n$ matrix X .

Table 7.

$F(X)$	equivalently	$\frac{\partial F(X)}{\partial x_{ij}}$	Conditions
$\text{Tr } X$	$\langle I, X \rangle$	δ_{ij}	
$\text{Tr } AX^T$	$\langle A, X \rangle$	$A_{i,j}$	$A \in \mathbb{R}^{m \times n}$
$a^T X a$	$\langle aa^T, X \rangle$	$a_i a_j$	$a \in \mathbb{R}^n, m = n$
$\text{Tr } X^2$	$\langle X, X \rangle$	$2X_{j,i}$	$m = n$

Let Φ and Ψ be functions of a square matrix variable X . The following derivatives of composite functions allow us to treat many practical problems (Table ??). We can

Table 8.

$F(X)$	equivalently	$\frac{\partial F(X)}{\partial x_{ij}}$	Conditions
$\text{Tr } A\Phi(X)$	$\langle A, \Phi(X) \rangle$	$\langle A, \frac{\partial \Phi(X)}{\partial x_{ij}} \rangle$	
$\text{Tr } \Phi(X)^2$	$\langle \Phi(X), \Phi(X) \rangle$	$2\langle \Phi(X), \frac{\partial \Phi(X)}{\partial x_{ij}} \rangle$	
$\text{Tr } (\Phi(X)\Psi(X))$	$\langle \Phi(X), \Psi(X) \rangle$	$\langle \Phi(X), \frac{\partial \Psi(X)}{\partial x_{ij}} \rangle + \langle \frac{\partial \Phi(X)}{\partial x_{ij}}, \Psi(X) \rangle$	

use it, for instance, to get the following two results for a $n \times n$ matrix X and $a \in \mathbb{R}^n$:

$$\begin{aligned} \frac{\partial}{\partial x_{ij}}(a^T X^{-1} a) &= \frac{\partial}{\partial x_{ij}} \langle aa^T, X^{-1} \rangle \\ &= -\langle aa^T, X^{-1} E_{ij} X^{-1} \rangle \\ &= -a^T X^{-1} E_{ij} X^{-1} a, \end{aligned}$$

in particular,

$$\frac{\partial}{\partial x_{ij}} \text{Tr } X^{-1} = \frac{\partial}{\partial x_{ij}} \langle I, X^{-1} \rangle = -\langle I, X^{-1} E_{ij} X^{-1} \rangle = -\text{Tr}(X^{-1} E_{ij} X^{-1}).$$

Recall that for a *symmetric* $n \times n$ matrix X , the above two formulas would change to

$$\frac{\partial}{\partial x_{ij}} (a^T X^{-1} a) = -a^T (Z_{ij} + Z_{ij}^T - \text{diag} Z_{ij}) a$$

and

$$\frac{\partial}{\partial x_{ij}} \text{Tr } X^{-1} = -\text{Tr}(Z_{ij} + Z_{ij}^T - \text{diag} Z_{ij})$$

with

$$Z_{ij} = X^{-1} E_{ij} X^{-1}.$$

A.3. Second-order derivatives

To compute the second-order derivatives of functions of matrices, we can simply apply the formulas derived in the previous sections to the elements of the gradients. Thus we get, for instance,

$$\frac{\partial^2 X^2}{\partial x_{ij} \partial x_{kl}} = \frac{\partial}{\partial x_{kl}} (E_{ij} X + X E_{ij}) = E_{ij} E_{kl} + E_{kl} E_{ij}$$

or

$$\frac{\partial^2 X^{-1}}{\partial x_{ij} \partial x_{kl}} = \frac{\partial}{\partial x_{kl}} (-X^{-1} E_{ij} X^{-1}) = X^{-1} E_{ij} X^{-1} E_{kl} X^{-1} + X^{-1} E_{kl} X^{-1} E_{ij} X^{-1}$$

for the matrix valued functions, and

$$\frac{\partial^2}{\partial x_{ij} \partial x_{kl}} \langle \Phi(X), \Phi(X) \rangle = 2 \left\langle \frac{\partial \Phi(X)}{\partial x_{kl}}, \frac{\partial \Phi(X)}{\partial x_{ij}} \right\rangle + 2 \left\langle \Phi(X), \frac{\partial^2 \Phi(X)}{\partial x_{ij} \partial x_{kl}} \right\rangle$$

for scalar valued matrix functions. Other formulas easily follow.