PENBMI User's Guide (Version 2.1)

Michal Kočvara Michael Stingl
PENOPT GbR
www.penopt.com

March 5, 2006

Contents

Changes to old versions

Changes to version 1.1

- Iterative solvers of the Newton equation added
- Optional approximate Hessian of the Augmented Lagrangian
- Changes in handling the parameter `ALPHA` with the goal to increase the final precision

Changes to version 2.0

- New option for Newton system solution added which, in connection with the CG method, requires no storage of the full Hessian (option `NWT_SYS_MODE=3`)
- Minor bugs fixed

1 Installation

1.1 Unpacking

UNIX versions
The distribution is packed in file `penbmi.tar.gz`. Put this file in an arbitrary directory. After uncompressing the file to `penbmi.tar` by command `gunzip penbmi.tar.gz`, the files are extracted by command `tar -xvf penbmi.tar`.

Win32 version
The distribution is packed in file `penbmi.zip`. Put this file in an arbitrary directory and extract the files by `PKZIP`.
In both cases, the directory PENBMI2.1 containing the following files and subdirectories will be created

LICENSE: a file containing the PENBMI license agreement;

c: a directory containing the files

driver_bmi_c.c, a sample driver implemented in C,
penbmi.h, a header file to included by C driver,
penout.c, sample implementation of Penbmi output functions;
make_{CC}.txt, a makefile to build a sample C program;

doc: a directory containing this User's Guide and a User's Guide for the MATLAB program `penfeas_lmi.m`;

fortran: a directory containing the files

driver_bmi_f.f, sample driver implemented in FORTRAN,
penout.c, see above;
penout.o(bj), precompiled version of `penout.c`;
make_{FC}.txt, a makefile to build a sample FORTRAN program;

libs: a directory containing the Penbmi library and Atlas libraries;

matlab: a directory containing the files

penbmim.c, the MATLAB interface file,
penoutm.c, MATLAB version of `penout.c`,
make_penbmim.m, M-file containing MEX link command,
bmi.m, M-file containing a sample problem in PEN format,
LQ.m, M-file containing a sample problem coded using YALMIP,
penfeas_bmi.m, checks feasibility of a system of BMIs;

1.2 Compilation

Requirements

For successful compilation and linkage, depending on the operating system and the program to be created, the following software packages have to be installed:

UNIX versions

- `gcc` compiler package (C driver program)
- `g77` compiler package (FORTRAN driver program)

- MATLAB version 5.0 or later including MEX compiler package and `gcc` compiler package (MATLAB dynamic link library `penbmim.*`)

Win32 version

- VISUAL C++ version 6.0 or later (C driver program)
- VISUAL FORTRAN version 6.0 or later (FORTRAN driver program)
- MATLAB version 5.0 or later including MEX compiler package and VISUAL C++ version 6.0 or later (MATLAB dynamic link library `penbmim.*`)

To build a C driver program

UNIX versions

Go to directory `c` and invoke Makefile by command

```
make -f make_gcc.txt.
```

Win32 version

Go to directory `c` and invoke Makefile by command

```
nmake -f make_vc.txt.
```

To build a FORTRAN driver program

UNIX versions

Go to directory `fortran` and invoke Makefile by command

```
make -f make_g77.txt.
```

Win32 version

Go to directory `fortran` and invoke Makefile by command

```
nmake -f make_df.txt.
```

To build a MATLAB dynamic link library `penbmim.*`

Start MATLAB, go to directory `matlab` and invoke link command by

```
make_penbmim.
```

In case the user wants to use his/her own LAPACK, BLAS or ATLAS implementations, the makefiles resp. M-files in directories `c`, `fortran` and `matlab` have to be modified appropriately.

2 The problem

We solve the SDP problem with quadratic objective function and linear and bilinear matrix inequality constraints:

$$\min_{x\in\mathbb{R}^n} \frac{1}{2}x^TQx + f^Tx \tag{1}$$

$$\text{s.t.} \quad \sum_{k=1}^{n} b_k^i x_k \le c^i, \qquad i = 1,\ldots,m_\ell$$

$$A_0^i + \sum_{k=1}^{n} x_k A_k^i + \sum_{k=1}^{n}\sum_{\ell=1}^{n} x_k x_\ell K_{k\ell}^i \preccurlyeq 0, \qquad i = 1,\ldots,m\,.$$

The linear part of the matrix constraints can be written as one constraint with block diagonal matrices as follows:

$$\begin{pmatrix} A_0^1 & & & \\ & A_0^2 & & \\ & & \ddots & \\ & & & A_0^m \end{pmatrix} + \begin{pmatrix} A_1^1 & & & \\ & A_1^2 & & \\ & & \ddots & \\ & & & A_1^m \end{pmatrix} x_1 + \begin{pmatrix} A_2^1 & & & \\ & A_2^2 & & \\ & & \ddots & \\ & & & A_2^m \end{pmatrix} x_2 + \ldots + \begin{pmatrix} A_n^1 & & & \\ & A_n^2 & & \\ & & \ddots & \\ & & & A_n^m \end{pmatrix} x_n$$

Here we use the abbreviations $n := \texttt{vars}$, $m_\ell := \texttt{constr}$, and $m := \texttt{mconstr}$.
In the same way we can write down the bilinear part of the matrix constraints.

3 The algorithm

For a detailed description of the algorithm, see [1].
The algorithm is based on a choice of penalty/barrier function Φ_P that penalizes the inequality constraints. This function satisfies a number of properties that guarantee that for any $P > 0$, we have

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

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

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \ & \frac{1}{2}x^T Q x + f^T x \qquad (\text{SDP}_\Phi) \\ \text{s.t.} \ & \Phi_P(\mathcal{A}(x)) \preccurlyeq 0\,. \end{aligned}$$

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

$$F(x,U,P) = \frac{1}{2}x^T Q x + f^T x + \langle U, \Phi_P\left(\mathcal{A}(x)\right)\rangle_{\mathbb{S}_d}\,; \qquad (2)$$

here d is the dimension of the matrix operator $\mathcal{A}$ and $U \in \mathbb{S}^d$ are Lagrangian multipliers associated with the inequality constraints.
The algorithm combines ideas of the (exterior) penalty and (interior) barrier methods with the Augmented Lagrangian method.

Algorithm 3.1 *Let x^1 and U^1 be given. Let $P^1 > 0$. For $k = 1, 2, \ldots$ repeat till a stopping criterium is reached:*

$$\begin{aligned} &(i) && \text{Find } x^{k+1} \text{ such that } \|\nabla_x F(x^{k+1}, U^k, P^k)\| \le K \\ &(ii) && U^{k+1} = D_{\mathcal{A}}\Phi_p(\mathcal{A}(x); U^k) \\ &(iii) && P^{k+1} < P^k\,. \end{aligned}$$

The details of the implementation can be found in [1].

4 C/C++/FORTRAN interface

PENBMI can also be called as a function (or subroutine) from a C or FORTRAN program. In this case, the user should link the PENBMI library to his/her program.

4.1 Calling PENBMI from a C/C++ program

For the implementation of a C interface the user should perform the following steps:

First, the user must include the PENBMI header file as in the following line:

```
#include "penbmi.h"
```

Second, the variables for the problem data have to be declared as in the following piece of code:

```
/* basic problem dimensions */
int vars = 0, constr = 0, mconstr = 0;
int luoutput = 0, lbi = 0, lai = 0, nzsai = 0;

/* error flag */
int inform = 0;

/* function value */
double fX = 0.;

/* array reserved for block sizes of matrix constraints */
int *msizes = 0;

/* array reserved for initial iterate (input)
   and optimal primal variables (output) */
double *x0 = 0;

/* array reserved for optimal dual variables (output) */
double *uoutput = 0;

/* objective */
double *fobj = 0;

/* objective, quadratic part */
int q_nzs = 0;
int *q_col = 0;
int *q_row = 0;
double *q_val = 0;

/* rhs of linear constraints */
double *ci = 0;

/* arrays for linear constraints */
int *bi_dim = 0, *bi_idx = 0;
double *bi_val = 0;

/* arrays for matrix constraints */
int *ai_dim = 0, *ai_idx = 0, *ai_nzs = 0, *ai_col = 0,
    *ai_row = 0;
```

```
double *ai_val = 0;

/* arrays for bilinear matrix constraints */
int    *ki_dim = 0, *ki_col = 0, *ki_row = 0,
       *ki_idx = 0, *ki_nzs = 0;
int    *kj_idx = 0;
double *ki_val = 0;

/* arrays reserved for results */
int iresults[4] = {0,0,0,0};
double fresults[3] = {0.0,0.0,0.0};

/* default options */
int ioptions[12] = {1,50,100,2,0,1,0,0,0,0,0,0};
double foptions[12] = {1.0,0.7,0.1,1.0E-7,1.0E-6,
                       1.0E-14,1.0E-2,1.1e0,
                       0.0,1.0,1.0E-7,5.0E-2};
```

Third, the user should specify the problem dimensions by assigning values to variables

```
vars, mconstr, constr,lbi, lai, nzsai, luoutput.
```

Using these numbers, the user should allocate memory as it is shown below:

```
msizes = INTEGER (mconstr);
x0 = DOUBLE(vars);
uoutput = DOUBLE(luoutput);
fobj = DOUBLE(vars);

if(q_nzs) {
  q_col = INTEGER(q_nzs);
  q_row = INTEGER(q_nzs);
  q_val = DOUBLE(q_nzs);
}

if(constr) {
  ci = DOUBLE(constr);
  bi_dim = INTEGER(constr);
  bi_idx = INTEGER(lbi);
  bi_val = DOUBLE(lbi);
}

if(mconstr) {
  ai_dim = INTEGER(mconstr);
  ai_idx = INTEGER(lai);
  ai_nzs = INTEGER(lai);
  ai_col = INTEGER(nzsai);
  ai_row = INTEGER(nzsai);
  ai_val = DOUBLE(nzsai);
  ki_dim = INTEGER(mconstr);
  ki_idx = INTEGER(lki);
  kj_idx = INTEGER(lki);
  ki_nzs = INTEGER(lki);
  ki_col = INTEGER(nzski);
  ki_row = INTEGER(nzski);
  ki_val = DOUBLE(nzski);
```

```
}
```

Next, the problem data should be assigned to arrays

```
x0, fobj, ci, x0, bi_dim, bi_idx, bi_val,
q_col, q_row, q_val,
ai_dim, ai_idx, ai_nzs, ai_val, ai_col, ai_row,
ki_dim, ki_idx, kj_idx, ki_nzs, ki_val, ki_col, ki_row.
```

and some non default options could be set by changing entries in the arrays

```
ioptions, foptions.
```

The meaning of the input/output parameters and the options are explained in detail in section 4.3. Finally, the user should call PENBMI like

```
/* Call penbmi */
  penbmi(vars, constr, mconstr, msizes,
  &fX, x0, 0, uoutput, fobj,
  q_nzs, q_col, q_row, q_val,
  ci, bi_dim, bi_idx, bi_val,
  ai_dim, ai_idx, ai_nzs,
  ai_val, ai_col, ai_row,
  ki_dim, ki_idx, kj_idx, ki_nzs,
  ki_val, ki_col, ki_row,
  ioptions, foptions,
  iresults, fresults, &inform);
```

and free memory.

A sample implementation is included in the file `driver_bmi_c.c` in directory `c`.

4.2 Calling PENBMI from a FORTRAN program

Given the (upper bounds on) values

```
vars1, mconstr1, constr1, q_nzs1
```

of

```
vars, mconstr, constr, q_nzs
```

and dimensions

```
lbi, lai, nzsai, lki, nzski, luoutput
```

of arrays

```
bi_idx, ai_idx, ai_col, uoutput
```

(either from outer call or declared as parameters), the user can call subroutine `pennlpf` as in the following piece of code:

```
integer vars, constr, mconstr, msizes(mconstr1)
integer q_nzs, q_col(q_nzs1), q_row(q_nzs1)
integer bi_dim(constr1), bi_idx(lbi)
integer ai_dim(mconstr1), ai_idx(lai), ai_nzs(lai)
integer ai_col(nzsai), ai_row(nzsai)
integer ki_dim(mconstr1), ki_idx(lki), kj_idx(lki)
integer ki_nzs(lki), ki_col(nzski), ki_row(nzski)
integer ioptions(12), iresults(4), info

double precision fx, x0(vars1), uoutput(luoutput), fobj
```

```
      double precision ci(constr1), bi_val(lbi), ai_val(nzsai)
      double precision q_val(q_nzs1), foptions(12), fresults(5)
        ...
        ...

      call penbmif (vars, constr, mconstr, msizes,
     *              fx, x0, 0, uoutput, fobj,
     *              q_nzs, q_col, q_row, q_val,
     *              ci, bi_dim, bi_idx, bi_val,
     *              ai_dim, ai_idx, ai_nzs,
     *              ai_val, ai_col, ai_row,
     *              ki_dim, ki_idx, kj_idx, ki_nzs,
     *              ki_val, ki_col, ki_row,
     *              ioptions, foptions,
     *              iresults, fresults, info)
```

The input/output parameters are explained below.

A sample implementation is included in the file `driver_bmi_f.f` in directory `fortran`.

4.3 The input/output structure

We assume that the linear constraint vectors b^i can be sparse, so we give them in standard sparse format. We further assume that the matrix constraints data can be sparse. Here we distinguish two cases: Some of the matrices A_k^i for the $k-$th constraint can be empty; so we only give those matrices (for each matrix constraint) that are nonempty. Each of the nonempty matrices A_k^i can still be sparse; hence they are given in sparse format (value, column index, row index). As all the matrices are symmetric, we only give the upper triangle. *All index arrays (i.e., *.idx, *.col, and *.row) are zero-based, as in the C language.*

`vars`	number of variables
integer number	
`constr`	number of linear constraints
integer number	
`mconstr`	number of matrix constraints (diagonal blocks in each A_k)
integer number	
`msizes`	sizes of the diagonal blocks $A_k^1, A_k^2, \ldots, A_k^{\texttt{mconstr}}$
integer array	*length*: `mconstr`
`fx`	on exit: objective function value
double array	*length*: 1
`x0`	on entry: initial guess for the solution on exit: solution vector x (Not referenced, if `x0 = 0` on entry)
double array	*length*: `vars`
`uoutput`	on exit: linear multipliers u_i ($i = 1, \ldots,$ `constr`) followed by upper triangular parts (stored row-wise) of matrix multipliers U^j ($j = 1, \ldots,$ `mconstr`) (Not referenced, if `uoutput = 0` on entry)
double array	*length*: `constr + msizes(1)*(msizes(1)+1)/2 + msizes(2)*(msizes(2)+1)/2 + ...+ msizes(mconstr)*(msizes(mconstr)+1)/2`

fobj	objective vector f in full format
double array	*length*: vars
q_nzs	number of nonzeros in the upper triangle of the objective matrix Q; if q_nzs=0, the objective is considered to be linear
integer number	
q_val	nonzero values in the upper triangle of Q
double array	*length*: q_nzs
q_col	column indices of nonzero values in the upper triangle of Q
integer array	*length*: q_nzs
q_row	row indices of nonzero values in the upper triangle of Q
integer array	*length*: q_nzs
ci	right-hand side vector of the linear constraint c in full format
double array	*length*: constr
bi_dim	number of nonzeros in vector b_i for each linear constraint
integer array	*length*: constr
bi_idx	indices of nonzeros in each vector b_i
integer array	*length*: bi_dim(1)+bi_dim(2)+...+bi_dim(constr)
bi_val	nonzero values in each vector b_i corresponding to indices in bi_idx
double array	*length*: bi_dim(1)+bi_dim(2)+...+bi_dim(constr)
ai_dim	number of nonzero blocks $A^i_0, A^i_1, \ldots, A^i_{\texttt{vars}}$ for each matrix constraint $i = 1, 2, \ldots,$ mconstr
integer array	*length*: mconstr
ai_idx	indices of nonzero blocks for each matrix constraint
integer array	*length*: ai_dim(1)+ai_dim(2)+...+ai_dim(mconstr)
ai_nzs	number of nonzero values in each nonzero block $A^i_{\texttt{ai\_idx(1)}}, A^i_{\texttt{ai\_idx(2)}}, \ldots, A^i_{\texttt{ai\_idx(ai\_dim(i))}}$ for each matrix constraint $i = 1, 2, \ldots,$ mconstr
integer array	*length*: ai_dim(1)+ai_dim(2)+...+ai_dim(mconstr)
ai_val	nonzero values in the upper triangle of each nonzero block $A^i_{\texttt{ai\_idx(1)}}, A^i_{\texttt{ai\_idx(2)}}, \ldots, A^i_{\texttt{ai\_idx(ai\_dim(i))}}$ for each matrix constraint $i = 1, 2, \ldots,$ mconstr
double array	*length*: ai_nzs(1)+ai_nzs(2)+...+ai_nzs(length(ai_nzs))
ai_col	column indices of nonzero values in the upper triangle of each nonzero block $A^i_{\texttt{ai\_idx(1)}}, A^i_{\texttt{ai\_idx(2)}}, \ldots, A^i_{\texttt{ai\_idx(ai\_dim(i))}}$ for each matrix constraint $i = 1, 2, \ldots,$ mconstr
integer array	*length*: ai_nzs(1)+ai_nzs(2)+...+ai_nzs(length(ai_nzs))
ai_row	row indices of nonzero values in the upper triangle of each nonzero block $A^i_{\texttt{ai\_idx(1)}}, A^i_{\texttt{ai\_idx(2)}}, \ldots, A^i_{\texttt{ai\_idx(ai\_dim(i))}}$ for each matrix constraint $i = 1, 2, \ldots,$ mconstr
integer array	*length*: ai_nzs(1)+ai_nzs(2)+...+ai_nzs(length(ai_nzs))

ki_dim	number of nonzero blocks $K^i_{k\ell}, k, \ell \in \{1, \ldots,$ vars$\}$, for each matrix constraint $i = 1, 2, \ldots,$ mconstr
integer array	*length*: mconstr
ki_idx	first indices of nonzero blocks for each matrix constraint
integer array	*length*: ki_dim(1)+ki_dim(2)+...+ki_dim(mconstr)
kj_idx	isecond indices of nonzero blocks for each matrix constraint
integer array	*length*: ki_dim(1)+ki_dim(2)+...+ki_dim(mconstr)
ki_nzs	number of nonzero values in each nonzero block $K^i_{\texttt{ki\_idx}(\alpha),\texttt{kj\_idx}(\alpha)}, \alpha = 1, \ldots,$ ki_dim(i), for each matrix constraint $i = 1, 2, \ldots,$ mconstr
integer array	*length*: ki_dim(1)+ki_dim(2)+...+ki_dim(mconstr)
ki_val	nonzero values in the upper triangle of each nonzero block $K^i_{\texttt{ki\_idx}(\alpha),\texttt{kj\_idx}(\alpha)}, \alpha = 1, \ldots,$ ki_dim(i), for each matrix constraint $i = 1, 2, \ldots,$ mconstr
double array	*length*: ki_nzs(1)+ki_nzs(2)+...+ki_nzs(length(ki_nzs))
ki_col	column indices of nonzero values in the upper triangle of each nonzero block $K^i_{\texttt{ki\_idx}(\alpha),\texttt{kj\_idx}(\alpha)}, \alpha = 1, \ldots,$ ki_dim(i), for each matrix constraint $i = 1, 2, \ldots,$ mconstr
integer array	*length*: ki_nzs(1)+ki_nzs(2)+...+ki_nzs(length(ki_nzs))
ki_row	row indices of nonzero values in the upper triangle of each nonzero block $K^i_{\texttt{ki\_idx}(\alpha),\texttt{kj\_idx}(\alpha)}, \alpha = 1, \ldots,$ ki_dim(i), for each matrix constraint $i = 1, 2, \ldots,$ mconstr
integer array	*length*: ki_nzs(1)+ki_nzs(2)+...+ki_nzs(length(ki_nzs))

ioptions	integer valued options (see below)
integer array	*length*: 12
foptions	real valued options (see below)
double array	*length*: 12
iresults	on exit: integer valued output information (see below) (Not referenced if iresults = 0 on entry)
integer array	*length*: 4
fresults	on exit: real valued output information (see below) (Not referenced if fresults = 0 on entry)
double array	*length*: 3
foptions	on exit: erro flag (see below)
integer array	*length*: 1

IOPTIONS	name/value	meaning	default
ioption(0)	DEF 0 1	 use default values for all options use user defined values	
ioption(1)	PBM_MAX_ITER	maximum number of iterations of the overall algorithm	50
ioption(2)	UM_MAX_ITER	maximum number of iterations for the unconstrained minimization	100
ioption(3)	OUTPUT 0 1 2 3	output level no output summary output brief output full output	1
ioption(4)	DENSE 0 1	check density of the Hessian automatic check. For very large problems with dense Hessian, this may lead to memory difficulties. dense Hessian assumed	0
ioption(5)	LS 0 1	linesearch in unconstrained minimization do not use linesearch use linesearch	0
ioption(6)	XOUT 0 1	write solution vector x in the output file no yes	0
ioption(7)	UOUT 0 1	write computed multipliers in the output file no yes	0
ioption(8)	NWT_SYS_MODE 0 1 2 3	mode of solution of the Newton system Cholesky method preconditioned conjugate gradient method preconditioned CG method with approximate Hessian calculation preconditioned CG method with exact Hessian not explicitly assembled	0
ioption(9)	PREC_TYPE 0 1 2 3 4	preconditioner type for the CG method no preconditioner diagonal LBFGS appoximate inverse symmetric Gauss-Seidel	0
ioption(10)	DIMACS 0 1	print DIMACS error measures no yes	0
ioption(11)	TR_MODE 0 1	Trust-Region mode modified Newton method Trust Region method	0

FOPTIONS	name/value	meaning	default
foption(0)	U0	scaling factor for linear constraints; must be positive	1.0
foption(1)	MU	restriction for multiplier update for linear constraints	0.7
foption(2)	MU2	restriction for multiplier update for matrix constraints	0.1
foption(3)	PBM_EPS	stopping criterium for the overall algorithm	1.0e-7
foption(4)	P_EPS	lower bound for the penalty parameters	1.0e-6
foption(5)	UMIN	lower bound for the multipliers	1.0e-14
foption(6)	ALPHA	stopping criterium for unconstrained minimization	1.0e-2
foption(7)	P0	initial penalty value; set it lower (0.01–0.1) to maintain feasibility when starting from a feasible point.	1.1
foptions(8)	PEN_UP	penalty update; when set to 0.0, it is computed automatically	0.0
foptions(9)	ALPHA_UP	update of α; should either be equal to 1.0 (= no update) or smaller than 1.0 (e.g. 0.7)	1.0
foptions(10)	PRECISION_2	precision of the KKT conditions	1.0e-7
foptions(11)	CG_TOL_DIR	stopping criterion for the conjugate gradient algorithm	5.0e-2

IRESULTS	meaning
iresults(0)	number of outer iterations
iresults(1)	number of Newton steps
iresults(2)	number of linesearch steps
iresults(3)	ellapsed time in seconds

FRESULTS	meaning
fresults(0)	relative precision at $x_{\rm opt}$
fresults(1)	feasibility of linear inequalities at $x_{\rm opt}$
fresults(2)	feasibility of matrix inequalities at $x_{\rm opt}$
fresults(3)	complementary slackness of linear inequalities at $x_{\rm opt}$
fresults(4)	complementary slackness of matrix inequalities at $x_{\rm opt}$

INFO	meaning
info = 0	No errors.
info = 1	Cholesky factorization of Hessian failed. The result may still be usefull.
info = 2	No progress in objective value, problem probably infeasible.
info = 3	Linesearch failed. The result may still be usefull.
info = 4	Maximum iteration limit exceeded. The result may still be usefull.
info = 5	Wrong input parameters (ioptions, foptions).
info = 6	Memory error.
info = 7	Unknown error, please contact PENOPT Gbr (contact @penopt.com).

5 MATLAB interface

5.1 Calling `penbmi` from MATLAB

In MATLAB, `penbmim` is called with the following arguments:

```
[f,x,u,iflag,niter,feas] = penbmim(pen);
```

where

pen . . . the input structure described above

f . . . the value of the objective function at the computed optimum

x . . . the value of the dual variable at the computed optimum

u . . . the value of the primal variable at the computed optimum

iflag . . . exit information

- 0 . . . No errors.
- 1 . . . Cholesky factorization of Hessian failed. The result may still be usefull.
- 2 . . . No progress in objective value, problem probably infeasible.
- 3 . . . Linesearch failed. The result may still be usefull.
- 4 . . . Maximum iteration limit exceeded. The result may still be usefull.
- 5 . . . Wrong input parameters (`ioptions`, `foptions`).
- 6 . . . Memory error.
- 7 . . . Unknown error, please contact PENOPT GbR (contact @penopt.com).

niter . . . a 4x1 matrix with elements

- niter(1) . . . number of outer iterations
- niter(2) . . . number of Newton steps
- niter(3) . . . number of linesearch steps
- niter(4) . . . ellapsed time in seconds

feas . . . a 3x1 matrix with stopping criteria values

- feas(1) . . . relative precision at $x_{\rm opt}$
- feas(2) . . . feasibility of linear inequalities at $x_{\rm opt}$
- feas(3) . . . feasibility of matrix inequalities at $x_{\rm opt}$
- feas(4) . . . complementary sleckness of linear inequalities at $x_{\rm opt}$
- feas(5) . . . complementary sleckness of matrix inequalities at $x_{\rm opt}$

5.2 `pen` input structure in MATLAB

The user must create a MATLAB structure array with fields described in Section 4.3.

Example 1. We have a linear objective function with $f = (1, 2, 3)^T$. Assume that we have no linear constraints. Assume further that we have two linear matrix inequality constraints, first of size (3x3), second of size (2x2). The first constraint contains full matrices, the second one diagonal matrices:

$$\begin{pmatrix} A_0^1 & & \\ & & \\ & & A_0^2 \end{pmatrix} + \begin{pmatrix} A_1^1 & & \\ & & \\ & & A_1^2 \end{pmatrix} x_1 + \begin{pmatrix} A_2^1 & & \\ & & \\ & & A_2^2 \end{pmatrix} x_2 + \begin{pmatrix} A_3^1 & & \\ & & \\ & & A_3^2 \end{pmatrix} x_3$$

The blocks A_k^1 have then 6 nonzero elements, block A_k^2 only two nonzero elements (recall that we only give elements of the upper triangular matrix). In this case

vars = 3
constr = 0
mconstr = 2
msizes = (3,2)
x0 = (0.0,0.0,0.0) (for example)
fobj = (1.0,2.0,3.0)
q_nzs = 0
ci = (0.0)
bi_dim = (0)
bi_idx = (0)
bi_val = (0.0)
ai_dim = (4,4)
ai_idx = (0,1,2,3,0,1,2,3)
ai_nzs = (6,6,6,6,2,2,2,2)
ai_val = $(A_0^1(1), \ldots, A_0^1(6),\ A_1^1(1), \ldots, A_1^1(6), \ldots\ldots,$
$A_0^2(1), A_0^2(2), A_1^2(1), A_1^2(2), A_2^2(1), A_2^2(2), A_3^2(1), A_3^2(2))$
ai_col = (0,1,2,1,2,2, 0,1,2,1,2,2,$\ldots\ldots$, 0,1,0,1,0,1,0,1)
ai_row = (0,0,0,1,1,2, 0,0,0,1,1,2,$\ldots\ldots$, 0,1,0,1,0,1,0,1)

Example 2. Consider a (fictitious) example with a linear-quadratic objective function with

$$f = (1,2,3)^T \quad \text{and} \quad Q = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}.$$

We have two linear constraints with

$$b_1 = (0,0,1)^T, \quad b_2 = (5,6,0)^T, \quad c = (3,-3)^T.$$

Assume further that we have two linear matrix inequality constraints, first of size (3x3), second of size (2x2). The first constraint contains sparse matrices, the second one diagonal matrices. Some of the matrices are empty, as shown below:

$$\left(\begin{array}{ccc|c} 0 & & & \\ & 0 & & \\ & & 0 & \\ \hline & & & \begin{matrix} 0 & \\ & 1 \end{matrix} \end{array}\right) + \left(\begin{array}{ccc|c} 2 & -1 & 0 & \\ & 2 & 0 & \\ & & 2 & \\ \hline & & & \begin{matrix} 1 & \\ & -1 \end{matrix} \end{array}\right) x_1$$
$$+ \left(\begin{array}{ccc|c} 0 & & & \\ & 0 & & \\ & & 0 & \\ \hline & & & \begin{matrix} 3 & \\ & -3 \end{matrix} \end{array}\right) x_2 + \left(\begin{array}{ccc|c} 2 & 0 & -1 & \\ & 2 & 0 & \\ & & 2 & \\ \hline & & & \begin{matrix} 0 & \\ & 0 \end{matrix} \end{array}\right) x_3$$

In this case

```
   vars = 3;
 constr = 2;
mconstr = 2;
 msizes = [3,2];
     x0 = [0.0,0.0,0.0] (for example);
   fobj = [1.0,2.0,3.0];
  q_nzs = 5;
 qi_col = [0, 1, 1, 2, 2];
 qi_row = [0, 0, 1, 1, 2];
 qi_val = [2.0, -1.0, 2.0, -1.0, 2.0];
     ci = [3.0 , -3.0];
 bi_dim = [1,2];
 bi_idx = [2,0,1];
 bi_val = [1.0, 5.0, 6.0];
 ai_dim = [2,3];
 ai_idx = [1,3,0,1,2];
 ai_nzs = [4,4,1,2,2];
 ai_val = [2.0,-1.0,2.0,2.0, 2.0,-1.0,2.0,2.0, 1.0, 1.0,-1.0, 3.0,-3.0];
 ai_col = [0,1,1,2, 0,2,1,2, 1, 0,1, 0,1)];
 ai_row = [0,0,1,2, 0,0,1,2, 1, 0,1, 0,1];
```

Example 3. We solve a problem in three variables with box constraints and one linear-bilinear matrix constraint of dimension three:

$$\begin{aligned} \min_{x\in\mathbb{R}^3} \; & x_3 \\ \text{s.t.} \quad & x_1 \in [-0.5, 2], \quad x_2 \in [-3, 7] \\ & A_0 + x_1 A_1 + x_2 A_2 + x_1 x_2 K_{12} - x_3 I \preccurlyeq 0 \end{aligned}$$

with

$$A_0 = \begin{pmatrix} -10 & -0.5 & -2 \\ -0.5 & 4.5 & 0 \\ -2 & 0 & 0 \end{pmatrix} \qquad A_1 = \begin{pmatrix} 9 & 0.5 & 0 \\ 0.5 & 0 & -3 \\ 0 & -3 & -1 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} -1.8 & -0.1 & -0.4 \\ -0.1 & 1.2 & -1 \\ -0.4 & -1 & 0 \end{pmatrix} \qquad K_{12} = \begin{pmatrix} 0 & 0 & 2 \\ 0 & -5.5 & 3 \\ 2 & 3 & 0 \end{pmatrix}$$

```
   vars = 3;
 constr = 4;
mconstr = 1;
 msizes = [3];
     x0 = [1.,0.,0.];
   fobj = [0,0,1];
     ci = [0.5,2,3,7];
 bi_dim = [1,1,1,1];
 bi_idx = [0,0,1,1];
 bi_val = [-1,1,-1,1];
 ai_dim = [4];
 ai_idx = [0,1,2,3];
 ai_nzs = [4,4,5,3];
 ai_val = [-10,-0.5,-2,4.5, 9, 0.5,-3,-1, -1.8,-0.1,-0.4,1.2,-1, -1,-1,-1];
 ai_col = [0,1,2,1, 0,1,2,2, 0,1,2,1,2, 0,1,2];
 ai_row = [0,0,0,1, 0,0,1,2, 0,0,0,1,1, 0,1,2];
 ki_dim = [1];
 ki_idx = [1];
 kj_idx = [2];
 ki_nzs = [3];
 ki_val = [2,-5.5,3];
 ki_col = [2,1,2];
 ki_row = [0,1,1];
```

Example 4. Consider the following bilinear matrix inequality arising often in Robust Control problems

$$PBK \preccurlyeq 0 \tag{3}$$

where B is a given matrix and P, K are *matrix variables*. Let us show how to write this inequality in the form used in (1), i.e., as a BMI with vector variables. Assume, for simplicity, that all matrices have dimension 2×2. Denote elements of a matrix by the corresponding lower-case letter, e.g.,

$$M = \begin{pmatrix} m_1 & m_3 \\ m_4 & m_2 \end{pmatrix}.$$

Then we have

$$M = \sum_{i=1}^{4} m_i Z_i \,,$$

with

$$Z_1 = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \quad Z_2 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \quad Z_3 = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad Z_4 = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}.$$

Thus we can write the original matrix inequality (3) (with matrix variables P, K) as a matrix inequality of type (1):

$$\sum_{i=1}^{4}\sum_{j=1}^{4} p_i k_j Z_i B Z_j \preccurlyeq 0$$

with vector variables

$$p = (p_1\ p_2\ p_3\ p_4)^T, \qquad k = (k_1\ k_2\ k_3\ k_4)^T.$$

6 PENBMI and YALMIP

Parser YALMIP[1] offers a simple and comfortable way of formulating LMI/BMI problems within MATLAB. PENBMI can then be used to solve the problem, without the need to set up the `pen` input structure. The example below shows how to formulate and solve a simple BMI problem.

Example 5. Assume we want to solve an LQ optimal feedback problem formulated as BMI:

$$\begin{aligned} \min_{P\in\mathbb{R}^{2\times 2}, K\in\mathbb{R}^{1\times 2}} \ & \operatorname{trace}(P) \\ \text{s.t.} \quad & (A+BK)^T P + P(A+BK) \prec -I_{2\times 2} - K^T K \\ & P \succ 0 \end{aligned}$$

with

$$A = \begin{pmatrix} -1 & 2 \\ -3 & -4 \end{pmatrix}, \qquad B = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Using YALMIP, the problems is formulated and solved by the following few lines

```
>> A = [-1 2;-3 -4]; B = [1;1];
>> P = sdpvar(2,2); K = sdpvar(1,2);
>> F = set(P>0)+set((A+B*K)'*P+P*(A+B*K) < -eye(2)-K'*K);
>> solvesdp(F,trace(P),sdpsettings('solver','penbmi'));
```

In fact, when PENBMI is the only installed BMI code, the last line can be replaced by

```
>> solvesdp(F,trace(P));
```

The PENBMI options can be changed using `sdpsetting`. For instance,

```
>> options=sdpsettings('penbmi.U0',1000,'penbmi.P0',1.1);
>> solvesdp(F,trace(P),options);
```

will set the `foptions` `U0` and `P0` to 1000 and 1.1, respectively.

The directory **matlab** contains a sample YALMIP code `LQ.m` (solving the above example) that includes all available options. For more details of using YALMIP, see the YALMIP manual.

[1] http://control.ee.ethz.ch/ joloef/yalmip.msql

7 General recommendations

7.1 Dense versus Sparse

For the efficiency of PENBMI, it is important to know if the problem has sparse or dense Hessian. The program can check this automatically. The check may take some time and memory, so if you know that the Hessian is dense (and this is the case of most problems), you can avoid the check. The check is switched on and off by parameter `CHECKDENSITY` in 'in.txt' (SDPA version) or by option `DENSE` in `ioptions` (MATLAB version). The default is 'on' (do the check).

7.2 High accuracy

The default values of the parameters in file `in.txt` are set to achieve relatively high accuracy in the primal and dual solutions. Typically, one obtains about 7 digits of accuracy in the objective function of the SDP problem, 1st-order optimality criteria and primal feasibility. To achieve even higher accuracy one can decrease parameter

`PRECISION2` to 1.0e-9.

This may lead to an inefficient code.

7.3 Cholesky versus iterative solvers

When computing the search direction in the Newton method, we have to solve a system of linear equations with a symmetric positive definite matrix (Hessian of the augmented Lagrangian). This system can either be solved by (possibly sparse) Cholesky decomposition or by the preconditioned conjugate gradient method. The choice is done by parameter

`USECG`

When this parameter is set to 1 or 2, CG method is used in connection with a preconditioner chosen by parameter

`PRECTYPE`

We recommend to use CG when *the number of variables is large and, in particular, when additionally the size of the matrix constraint is relatively small*. While PENSDP with CG can reach the same high accuracy as with Cholesky decomposition, the code becomes much more efficient when the required accuracy is decreased by setting

`PRECISION` to 1e-4

and

`PRECISION2` to 1e-2.

The choice of a preconditioner depends to a large extent on the problem. We recommend to try several preconditioners, in the order 4–1–2–3–0.

Warning: For certain kind of problems, the choice of CG method may lead to a rather inefficient code, compared to Cholesky decomposition.

7.4 Exact versus approximate Hessian

The use of the CG method allows us to use an exact or an approximate formula for the Hessian-vector product. This may be particularly efficient when the Hessian evaluation is expensive. Further, as the Hessian does not have to be stored, *the memory requirements of the approximate version are much smaller*. The option is chosen by setting

`NWT_SYS_MODE` to 2 (approximate Hessian-vector product)

`NWT_SYS_MODE` to 3 (exact Hessian-vector product)

The use of exact Hessian-vector product (option 3) is generally preferable.

As above, to increase the efficiency, we recommend to set

`PRECISION2` to 1e-2.

For

NWT_SYS_MODE set to 2 or 3,

only

PRECTYPE set to 0 or 2 is allowed.

Warning: For certain kind of problems, the choice of CG method with approximate Hessian may lead to a rather inefficient code, compared to Cholesky decomposition with exact Hessian.

References

[1] M. Kočvara and M. Stingl. PENNON—a code for convex nonlinear and semidefinite programming. *Optimization Methods and Software*, 8(3):317–333, 2003.