PENNLP User's Guide (Version 2.2)

Michal Kočvara Michael Stingl
www.penopt.com

November 29, 2008

Contents

1 Installation

1.1 Unpacking

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

Win32 version
The distribution is packed in file `pennlp.zip`. Put this file in an arbitrary directory and extract the files by `PKZIP`.
In both cases, the directory PENNLP2.2 containing the following files and subdirectories will be created

LICENSE: file containing the PENNLP license agreement;

bin: directory containing the files
pennlp2.2(.exe), the binary executable with AMPL interface,
example.mod, a model file of a sample problem in AMPL format,
example.nl, an nl-file created by AMPL from `example.mod`;

c: directory containing the files
driver_nlp_c.c, a sample driver implemented in C,
pennlp.h, a header file to be included by C driver,
penout.c, a sample implementation of PENNLP output functions;
make_{CC}.txt, a makefile to build a sample C program;

cpp: directory containing the files
driver_nlp_cc.cpp, a sample driver implemented in C++,
pennlp.h, a header file to be included by C++ driver,
penout.cpp, a sample implementation of PENNLP output functions;
make_{CPP}.txt, a makefile to build a sample C++ program;

doc: directory containing this User's Guide;

fortran: directory containing the files
driver_nlp_f.f, a 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;

lib: directory containing the PENSDP library and ATLAS libraries;

matlab: directory containing the files
pennlpm.c, the MATLAB interface file,
penoutm.c, MATLAB version of `penout.c`,
make_pennlpm.m, M-file containing MEX link command,
nlp.m, f.m, df.m, hf.m, g.m dg.m, hg.m, M-files
defining a sample problem in PEN format.

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 `pennlpm.*`)

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 `pennlpm.*`)

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 `pennlpm.*`

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

```
make_pennlpm.
```

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

2 The problem

We solve optimization problems with nonlinear objective subject to nonlinear inequality and equality constraints:

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

Here f, g_i and h_i are C^2 functions from $\mathbb{R}^n$ to $\mathbb{R}$.

3 The algorithm

To simplify the presentation of the algorithm, *we only consider inequality constraints*. For the treatment of the equality constraints, see [1].
The algorithm is based on a choice of penalty/barrier function $\varphi_g : \mathbb{R} \to \mathbb{R}$ that penalizes the inequality constraints. This function satisfies a number of properties (see [1]) that guarantee that for any $p_i > 0,\ i = 1,\dots,m_g$, we have

$$g_i(x) \le 0 \iff p_i\varphi_g(g_i(x)/p_i) \le 0, \quad i = 1,\dots,m\,.$$

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

$$\begin{aligned} &\min_{x\in\mathbb{R}^n} f(x) \\ &\text{s.t.} \qquad p_i\varphi_g(g_i(x)/p_i) \le 0, \qquad i = 1,\dots,m_g\,. \end{aligned} \tag{NLP$_\phi$}$$

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

$$F(x,u,p) = f(x) + \sum_{i=1}^{m_g} u_i p_i \varphi_g(g_i(x)/p_i)\,; \tag{1}$$

here $u \in \mathbb{R}^{m_g}$ are Lagrange 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 $p_i^1 > 0$, $i = 1,\dots,m_g$. For $k = 1,2,\dots$ repeat till a stopping criterium is reached:*

$$\begin{aligned} &(i) && \text{Find } x^{k+1} \text{ such that } \|\nabla_x F(x^{k+1},u^k,p^k)\| \le K \\ &(ii) && u_i^{k+1} = u_i^k \varphi_g'(g_i(x^{k+1})/p_i^k), \quad i = 1,\dots,m_g \\ &(iii) && p_i^{k+1} < p_i^k,\ i = 1,\dots,m_g\,. \end{aligned}$$

The approximate unconstrained minimization in Step (i) is performed either by the Newton method with line-search or by one of two variants of the Trust Region method (for details, see [1]). The minimization is optionally stopped when either

$$\|\nabla_x F(x^{k+1},u^k,p^k)\|_2 \le \alpha$$

or

$$\|\nabla_x F(x^{k+1},u^k,p^k)\|_2 \le \alpha \cdot \|u_i^k - u_i^k\varphi_g'(g_i(x^{k+1})/p_i^k)\|_2$$

or

$$\|\nabla_x F(x^{k+1},u^k,p^k)\|_{H^{-1}} \le \alpha\|\nabla_x F(x^k,u^k,p^k)\|_{H^{-1}}$$

with optional parameter α; by default, $\alpha = 10^{-1}$.
The multipliers calculated in Step (ii) are restricted in order to satisfy:

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

with some positive $\mu \leq 1$; by default, $\mu = 0.3$.
The update of the penalty parameter p in Step (iii) is performed in the following way: During the first three iterations we do not update the penalty vector p at all. After this kind of "warm start", the penalty vector is updated by some constant factor dependent on the initial penalty parameter π. The penalty update is stopped, if p_{eps} (by default 10^{-6}) is reached.
Algorithm 3.1 is stopped when both of the inequalities holds:

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

where ϵ is by default 10^{-7} (parameter `precision`).

4 AMPL interface

4.1 Running PENNLP

AMPL is a comfortable modelling language for optimization problems. For a description of AMPL we refer to [2] or `www.ampl.com`.

PENNLP is called in the standard AMPL style, i.e., either by a sequence like

```
> model sample.mod;
> data sample.dat;
> options solver pennlp;
> options pennlp_options 'convex=1 outlev=2';  (for instance)
> solve;
```

within the AMPL environment or from the command line by

```
> pennlp sample.nl 'convex=1 outlev=2'
```

A sample nl-file named `sample.nl` is included in directory `bin`.

4.2 Program options

The options are summarized in Table 1.

Recommendations

- Whenever you know that the problem is convex, use `convex=1`.
- When you have problems with convergence of the algorithm, try to
  - decrease `pinit`, e.g., `pinit=0.01` (This should be particulary helpful for nonconvex problems, if an initial guess of the solution is available).
  - increase (decrease) `uinit`, e.g., `uinit=10000`.
  - switch to Trust Region algorithm by `ncmode=1`
  - decrease `alpha`, e.g., `alpha=1e-3`
  - change stopping criterion for inner loop by setting `nwtstopcrt=1`

PENNLP–AMPL options

option	meaning	default
`alpha`	stopping parameter α for the Newton/Trust region method in the inner loop	1.0E-1
`alphaupd`	update of α	1.0e0
`autoini`	automatic initialization of multipliers 0 . . . off 1 . . . nonlinear (nonconvex) mode 2 . . . lp/qp mode	1
`autoscale`	automatic scaling 0 . . . on 1 . . . off	0
`cgtolmin`	minimum tolerance for the conjugate gradient algorithm	5.0e-2
`cgtolup`	update of tolerance for the conjugate gradient algorithm	1.0e0
`cmaxnzs`	tuning parameter for Hessian assembling in "nwtmode" (put > 0 to switch it on)	-1
`convex`	convex problem? 0 . . . generally nonconvex 1 . . . convex	0
`eqltymode`	initialization of equality multipliers 0 . . . two inequalities, symmetric 1 . . . two inequalities, unsymmetric 2 . . . augmented lagrangian 3 . . . direct 4 . . . direct (only nonlinear equalities)	3
`filerep`	output to file 0 . . . no 1 . . . yes	0
`hessianmode`	check density of the Hessian 0. . . automatic 1 . . . dense	0
`ignoreinit`	ignore initial solutions 0. . . do not ignore 1 . . . do ignore	0
`maxit`	maximum number of outer iterations	100
`mu`	restriction factor μ of multiplier update	0.5
`ncmode`	nonconvex mode 0. . . Modified Newton 1. . . Trust region	0
`nwtiters`	maximum number of iterations in the inner loop (Newton or Trust region method)	100
`nwtmode`	linear system solver 0. . . Cholesky method 1. . . conjugate gradient method 2. . . conjugate gradient method with approximate Hessian calculation 1. . . conjugate gradient method to dual system	0
`nwtstopcrit`	stopping criterium for the inner loop 0. . . $\|\nabla L(x^{k+1})\|_2 < \alpha$ 1. . . $\|\nabla L(x^{k+1})\|_2 < \alpha \cdot \|u_i^k - u_i^{k+1}\|_2$ 2. . . $\|\nabla L(x^{k+1})\|_{H^{-1}} < \alpha \cdot \|\nabla L(x^k)\|_{H^{-1}}$	0
`objno`	objective number in the AMPL .mod file	1
`ordering`	ordering for MA57	4

PENNLP–AMPL options *(cont.)*

`outlev`	output level 0 . . . silent mode 1 . . . brief output 2 . . . full output	1
`penalty`	penalty function 0. . . logarithmic barrier + quadratic penalty 1 . . . reciprocal barrier + quadratic penalty	0
`penup`	penalty update	0.5
`penupmode`	penalty update is performed: 0 . . . adaptively 1 . . . after each outer iteration	0
`peps`	minimal penalty	1.0E-7
`pinit`	initial penalty	1.0E0
`pivtol`	pivot tolerance for MA27/57	1.0E-2
`precision`	required final precision	1.0E-7
`precKKT`	required final precision of the KKT conditions	1.0E-5
`precond`	preconditioner type 0. . . no preconditioner 1. . . diagonal 2. . . L-BFGS 3. . . approximate inverse 4 . . . symmetric Gauss-Seidel	0
`timing`	timing destination 0 . . . no 1 . . . stdout 2 . . . stderr 3 . . . both	0
`uinit`	initial multiplier scaling factor	1.0
`uinitbox`	initial multiplier scaling factor for box constraints	1.0
`uinitnc`	initial multiplier scaling factor for nonlinear constraints	1.0
`umin`	minimal multiplier	1.0E-10
`usebarrier`	Use (mod.) barrier approach for boxes? 0. . . no 1. . . barrier 2. . . strict modified barrier	0
`version`	report PENNLP version 0 . . . yes 1 . . . no	0
`wantsol`	solution report without -AMPL. Sum of 0 . . . do not write .sol file 1 . . . write .sol file 2 . . . print primal variable 4 . . . print dual variable 8 . . . do not print solution message	0

5 C/C++/FORTRAN interface

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

The interfaces described below require the following reformulation of problem (NLP):

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \; & f(x) \\ \text{s.t.} \quad & lbv_i \le x_i \le ubv_i, \qquad i = 1,\dots,n \\ & lbc_i \le g_i(x) \le ubc_i, \qquad i = 1,\dots,m \end{aligned} \tag{NLP'}$$

The constraints are divided into box constraints and general constraints now. Each variable and constraint function is bounded from below and from above. If constraint i is an equality constraint, the user should set $lbc_i = ubc_i$. If no lower or upper bound is applied to a variable or constraint, the corresponding bound should be set to a value smaller than or equal to `-1.0e38` for lower bounds and larger than or equal to `1.0e38` for upper bounds.

For illustration we will use the following sample problem:

$$\begin{aligned} \min_{x\in\mathbb{R}^3} \; & x_1^2 + 4x_2^2 - x_3^2 + x_1x_2 - 2x_1x_3 \\ \text{s.t.} \quad & x_1^2 + x_2^2 + x_3^2 \ge 4 \\ & 2x_1 + 6x_2 + 4x_3 = 24 \\ & x_1, x_2, x_3 \ge 0 \end{aligned} \tag{NLP1}$$

5.1 Calling PENNLP from a C/C++ program

5.1.1 User provided functions

The user is required to provide six C functions. The names of the functions can be chosen by the user; here we use the following names:

`my_f` ... evaluates the objective function
`my_f_gradient` ... evaluates the gradient of objective function
`my_f_hessian` ... evaluates the Hessian of objective function
`my_g` ... evaluates the constraints
`my_g_gradient` ... evaluates the gradient of constraints
`my_g_hessian` ... evaluates the Hessian of constraints

The specification of the user-defined functions will be explained using the sample problem (NLP1).

`my_f`

```
void my_f (double *x, double* val) {
  static double value = 0;
  value = x[0]*x[0] + 4.*x[1]*x[1] - x[2]*x[2] +
          x[0]*x[1] - 2*x[0]*x[2];

  *val = value;
}
```

Arguments:

`x`	double array of length n storing current iterate x_{it} (input)
`val`	double array of length 1 storing $f(x_{it})$ (output)

Description:

- Array Compute $f(x_{it})$ and store it in `*val`.

Note:

- Array `x` should not be modified by the user.

`my_f_gradient`

```
void my_f_gradient (double *x, int* nnz,
                    int* ind, double* val) {
  ind[0] = 1;
  ind[1] = 2;
  ind[2] = 3;

  val[0] = 2.*x[0] + x[1] - 2.*x[2];
  val[1] = 8.*x[1] + x[0];
  val[2] = -2.*x[2] - 2.*x[0];

  *nnz = 3;
}
```

Arguments:

`x`	double array of length n storing current iterate x_{it} (input)
`nnz`	double array of length 1 storing number of non-zeros of ∇f (output)
`ind`	double array of length `nnz` storing non-zero structure of ∇f (output)
`val`	double array of length `nnz` storing non-zero values of ∇f (output)

Description:

1. Compute $\nabla f(x_{it})$;
2. Assign non-zero structure to `ind` and the corresponding values to `val`.

Note:

- Array `x` should not be modified by the user;
- Non-zero indices should be zero based;
- Non-zero structure should be constant.

`my_f_hessian`

```
void my_f_hessian (double *x, int* nnz, int* row,
                   int* col, double* val) {
  row[0] = 1;
  row[1] = 2;
  row[2] = 2;
  row[3] = 3;
  row[4] = 3;
  col[0] = 1;
  col[1] = 1;
  col[2] = 2;
  col[3] = 1;
```

```
  col[4] = 3;
  val[0] = 2.;
  val[1] = 1.;
  val[2] = 8.;
  val[3] = -2.;
  val[4] = -2.;
  *nnz = 5;
}
```

Arguments:

`x`	double array of length n storing current iterate x_{it} (input)
`nnz`	double array of length 1 storing number of non-zeros of $\nabla^2 f$ (output)
`row`	double array of length `nnz` non-zero row indices of $\nabla^2 f$ (output)
`col`	double array of length `nnz` non-zero column indices of $\nabla^2 f$ (output)
`val`	double array of length `nnz` storing non-zero values of $\nabla^2 f$ (output)

Description:

1. Compute $\nabla^2 f(x_{it})$;
2. Assign non-zero structure to `row` and `col` and the corresponding values to `val`.

Note:

- Array `x` should not be modified by the user;
- Non-zero indices should be zero based;
- Non-zero structure should be constant;
- **Values should be assigned to lower triangular part of $\nabla^2 f(x_{it})$ only**.

`my_g`

```
void my_g (int* i, double *x, double* val) {
  static double value = 0;

  switch (*i) {
    case 0:
      value = x[0]*x[0] + x[1]*x[1] + x[2]*x[2] - 4.;
      break;
    case 1:
      value = 2.*x[0] + 6.*x[1] + 4.*x[2] - 24.;
      break;
  }

  *val = value;
}
```

Arguments:

`i`	integer array of length 1 storing constraint number (input)
`x`	double array of length n storing current iterate x_{it} (input)
`val`	double array of length 1 storing $f(x_{it})$ (output)

Description:

- Compute $g_i(x_{it})$ and store it in `*val`.

Note:

- Array `x` should not be modified by the user;
- Linear constraints should be specified **after** nonlinear constraints.

`my_g_gradient`

```
void my_g_gradient (int* i, double *x, int* nnz,
                    int* ind, double* val) {
  switch (*i) {
    case 0:
      val[0] = 2.*x[0];
      val[1] = 2.*x[1];
      val[2] = 2.*x[2];
      ind[0] = 1;
      ind[1] = 2;
      ind[2] = 3;
      *nnz = 3;
      break;
    case 1:
      val[0] = 2.;
      val[1] = 6.;
      val[2] = 4.;
      ind[0] = 1;
      ind[1] = 2;
      ind[2] = 3;
      *nnz = 3;
      break;
  }
}
```

Arguments:

`i`	integer array of length 1 storing constraint number (input)
`x`	double array of length n storing current iterate x_{it} (input)
`nnz`	double array of length 1 storing number of non-zeros of ∇g_i (output)
`ind`	double array of length `nnz` storing non-zero structure of ∇g_i (output)
`val`	double array of length `nnz` storing non-zero values of ∇g_i (output)

Description:

1. Compute $\nabla g_i(x_{it})$;
2. Assign non-zero structure to `ind` and the corresponding values to `val`.

Note:

- Array `x` should not be modified by the user;
- Linear constraints should be specified **after** nonlinear constraints;
- Non-zero indices should be zero based;
- Non-zero structure should be constant.

`my_g_hessian`

```
void my_g_hessian (int* i, double *x, int* nnz,
                   int* row, int* col, double* val) {
  switch (*i) {
    case 0:
      val[0] = 2.;
      val[1] = 2.;
      val[2] = 2.;
      row[0] = 1;
      row[1] = 2;
      row[2] = 3;
      col[0] = 1;
      col[1] = 2;
      col[2] = 3;
      *nnz = 3;
      break;
    case 1:
      *nnz = 0;
      break;
  }
}
```

Arguments:

`i`	integer array of length 1 storing constraint number (input)
`x`	double array of length n storing current iterate x_{it} (input)
`nnz`	double array of length 1 storing number of non-zeros of $\nabla^2 g_i$ (output)
`row`	double array of length `nnz` non-zero row indices of $\nabla^2 g_i$ (output)
`col`	double array of length `nnz` non-zero column indices of $\nabla^2 g_i$ (output)
`val`	double array of length `nnz` storing non-zero values of $\nabla^2 g_i$ (output)

Description:

1. Compute $\nabla^2 g_i(x_{it})$;
2. Assign non-zero structure to `row` and `col` and the corresponding values to `val`.

Note:

- Array `x` should not be modified by the user;
- Linear constraints should be specified **after** nonlinear constraints;
- Non-zero indices should be zero based;
- Non-zero structure should be constant;
- **Values should be assigned to lower triangular part of $\nabla^2 f(x_{it})$ only.**

5.1.2 The driver

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

First, the user must include the PENNLP header file and declare the user provided functions as follows:

```
#include "pennlp.h"

void my_f (double *x, double* val);
void my_f_gradient (double *x, int* nnz, int* ind,
                    double* val);
void my_f_hessian (double *x, int* nnz, int* row,
                   int* col, double* val);
void my_g (int* i, double *x, double* val);
void my_g_gradient (int* i, double *x, int* nnz,
                    int* ind, double*val);
void my_g_hessian (int* i, double *x, int* nnz,
                   int* row, int* col, double* val);
```

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

```
  static int status = 0;
  static int nvars = 0;
  static int nlin = 0;
  static int nconstr = 0;
  static int nnz_gradient = 0;
  static int nnz_hessian = 0;
  double *xinit = 0;
  double *uopt = 0;
  double* lbv = 0;
  double* ubv = 0;
  double* lbc = 0;
  double* ubc = 0;
  double doptions[13] = {1.0e-7, 1.0e-0, 1.0e-0, 1.0e-2, 5.0e-1, 1.0e-1,
                                 1.0e-6, 1.0e-12, 1.0e-1, 0.05, 1.0, 1.0, 1.0};
  int ioptions[17] = {100, 100, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 0, -1, 0, 1, 0};
  double *dresults;
  int *iresults;
```

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

```
 nvars, nlin, nconstr.
```

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

```
  /* Allocate memory for bounds and initial iterates */
  lbv = DOUBLE(nvars);
  ubv = DOUBLE(nvars);
  xinit = DOUBLE(nvars);
  lbc = DOUBLE(nconstr);
  ubc = DOUBLE(nconstr);
  uopt =  DOUBLE(nconstr);
```

Next, the problem data should be assigned to variables

```
  nnz_gradient, nnz_hessian
```

and arrays

```
  lbv, ubv, lbc, ubc, xinit, uopt.
```

Also, non-default options could be set now by changing entries in the arrays

```
ioptions, foptions.
```

The meaning of the input/output parameters and the options are explained in detail in section 5.3. Finally, the user should call PENNLP like

```
/* call PENNLP solver library */
status = pennlp(nvars, nlin, nconstr,
  nnz_gradient, nnz_hessian,
  lbv, ubv, lbc, ubc,
  xinit, uopt,
  my_f, my_f_gradient, my_f_hessian,
  my_g, my_g_gradient, my_g_hessian,
  ioptions, doptions,
  iresults, dresults) ;
```

and free the memory.

Sample implementations are included in the files `driver_nlp_c.c` in directory `c` and `driver_nlp_cc.cpp` in directory `cpp`.

5.2 Calling PENNLP from a FORTRAN program

5.2.1 User provided functions

The user is required to provide six FORTRAN subroutines. The names of these subroutines can be chosen by the user; here we use the following names:

`MY_F` ... evaluate the objective function

`MY_F_GRADIENT` ... evaluate the gradient of objective function

`MY_F_HESSIAN` ... evaluate the Hessian of objective function

`MY_G` ... evaluate the constraints

`MY_G_GRADIENT` ... evaluate the gradient of constraints

`MY_G_HESSIAN` ... evaluate the Hessian of constraints

The specification of the user-defined functions will be explained using the sample problem (NLP1).

`MY_F`

```
      SUBROUTINE MY_F (X, VAL)
      DOUBLE PRECISION X(*), VAL
C
      VAL = X(1)*X(1) + 4.D0*X(2)*X(2) - X(3)*X(3) +
     *      X(1)*X(2) - 2.D0*X(1)*X(3)
      RETURN
      END
```

Arguments:

`F`	`DOUBLE PRECISION` array of length n storing current iterate x_{it} (input)
`VAL`	`DOUBLE PRECISION` variable storing $f(x_{it})$ (output)

Description:

- Compute $f(x_{it})$ and store it in VAL.

Note:

- Array X should not be modified by the user.

MY_F_GRADIENT

```
      SUBROUTINE MY_F_GRADIENT (X, NNZ, IND, VAL)
      DOUBLE PRECISION X(*), VAL(*)
      INTEGER NNZ, IND(*)
C
      IND(1) = 1
      IND(2) = 2
      IND(3) = 3
C
      VAL(1) = 2.D0*X(1) + X(2) - 2.D0*X(3)
      VAL(2) = 8.D0*X(2) + X(1)
      VAL(3) = -2.D0*X(3) - 2.D0*X(1)
C
      NNZ = 3
      RETURN
      END
```

Arguments:

X	DOUBLE PRECISION array of length n storing current iterate x_{it} (input)
NNZ	DOUBLE PRECISION variable storing number of non-zeros of ∇f (output)
IND	DOUBLE PRECISION array of length NNZ storing non-zero structure of ∇f (output)
VAL	DOUBLE PRECISION array of length NNZ storing non-zero values of ∇f (output)

Description:

1. Compute $\nabla f(x_{it})$;
2. Assign non-zero structure to IND and the corresponding values to VAL.

Note:

- Array X should not be modified by the user;
- Non-zero indices should be zero based;
- Non-zero structure should be constant.

MY_F_HESSIAN

```
      SUBROUTINE MY_F_HESSIAN (X, NNZ, ROW, COL, VAL)
      DOUBLE PRECISION X(*), VAL(*)
      INTEGER NNZ, ROW(*), COL(*)
C
      ROW(1) = 1
      ROW(2) = 2
      ROW(3) = 2
      ROW(4) = 3
      ROW(5) = 3
C
      COL(1) = 1
      COL(2) = 1
      COL(3) = 2
      COL(4) = 1
      COL(5) = 3
C
      VAL(1) = 2.D0
      VAL(2) = 1.D0
      VAL(3) = 8.D0
      VAL(4) = -2.D0
      VAL(5) = -2.D0

      NNZ = 5
      RETURN
      END
```

Arguments:

x	DOUBLE PRECISION array of length n storing current iterate x_{it} (input)
nnz	DOUBLE PRECISION variable storing number of non-zeros of $\nabla^2 f$ (output)
row	DOUBLE PRECISION array of length nnz non-zero row indices of $\nabla^2 f$ (output)
col	DOUBLE PRECISION array of length nnz non-zero column indices of $\nabla^2 f$ (output)
val	DOUBLE PRECISION array of length nnz storing non-zero values of $\nabla^2 f$ (output)

Description:

1. Compute $\nabla^2 f(x_{it})$;
2. Assign non-zero structure to ROW and COL and the corresponding values to VAL.

Note:

- Array X should not be modified by the user;
- Non-zero indices should be zero based;
- Non-zero structure should be constant;
- **Values should be assigned to lower triangular part of $\nabla^2 f(x_{it})$ only.**

MY_G

```
      SUBROUTINE MY_G (I, X, VAL)
      DOUBLE PRECISION X(*), VAL
      INTEGER I
C
      IF (I .EQ. 0) THEN
        VAL = X(1)*X(1) + X(2)*X(2) + X(3)*X(3) - 4.D0
      ELSE
        VAL = 2.D0*X(1) + 6.D0*X(2) + 4.D0*X(3) - 24.D0
      ENDIF
      RETURN
      END
```

Arguments:

i	INTEGER variable storing constraint number (input)
x	DOUBLE PRECISION array of length n storing current iterate x_{it} (input)
val	DOUBLE PRECISION variable storing $f(x_{it})$ (output)

Description:

- Compute $g_i(x_{it})$ and store it in VAL.

Note:

- Array X should not be modified by the user.
- Linear constraints should be specified **after** nonlinear constraints.

MY_G_GRADIENT

```
      SUBROUTINE MY_G_GRADIENT (I, X, NNZ, IND, VAL)
      DOUBLE PRECISION X(*), VAL(*)
      INTEGER I, NNZ, IND(*)
C
      IF (I .EQ. 0) THEN
          VAL(1) = 2.D0*X(1)
          VAL(2) = 2.D0*X(2)
          VAL(3) = 2.D0*X(3)
          IND(1) = 1
          IND(2) = 2
          IND(3) = 3
          NNZ = 3
      ELSE
        VAL(1) = 2.D0
          VAL(2) = 6.D0
          VAL(3) = 4.D0
          IND(1) = 1
          IND(2) = 2
          IND(3) = 3
          NNZ = 3
        ENDIF
      RETURN
      END
```

Arguments:

`i`	`INTEGER` variable storing constraint number (input)
`x`	`DOUBLE PRECISION` array of length n storing current iterate x_{it} (output)
`nnz`	`DOUBLE PRECISION` variable storing number of non-zeros of ∇g_i (output)
`ind`	`DOUBLE PRECISION` array of length `NNZ` storing non-zero structure of ∇g_i (output)
`val`	`DOUBLE PRECISION` array of length `NNZ` storing non-zero values of ∇g_i (output)

Description

1. Compute $\nabla g_i(x_{it})$;
2. Assign non-zero structure to `IND` and the corresponding values to `VAL`.

Note:

- Array `X` should not be modified by the user;
- Linear constraints should be specified **after** nonlinear constraints;
- Non-zero indices should be zero based;
- Non-zero structure should be constant.

`MY_G_HESSIAN`

```
      SUBROUTINE MY_G_HESSIAN (I, X, NNZ, ROW, COL, VAL)
      DOUBLE PRECISION X(*), VAL(*)
      INTEGER I, NNZ, COL(*), ROW(*)
C
      IF (I .EQ. 0) THEN
          VAL(1) = 2.D0
          VAL(2) = 2.D0
          VAL(3) = 2.D0
          ROW(1) = 1
          ROW(2) = 2
          ROW(3) = 3
          COL(1) = 1
          COL(2) = 2
          COL(3) = 3
          NNZ = 3
      ELSE
          NNZ = 0
        ENDIF
      RETURN
      END
```

Arguments:

`i`	`INTEGER` variable storing constraint number (input) (input)
`x`	`DOUBLE PRECISION` array of length n storing current iterate x_{it} (output)
`nnz`	`DOUBLE PRECISION` variable storing number of non-zeros of $\nabla^2 g_i$ (output)
`row`	`DOUBLE PRECISION` array of length `NNZ` non-zero row indices of $\nabla^2 g_i$ (output)
`col`	`DOUBLE PRECISION` array of length `NNZ` non-zero column indices of $\nabla^2 g_i$ (output)
`val`	`DOUBLE PRECISION` array of length `NNZ` storing non-zero values of $\nabla^2 g_i$ (output)

Description:

1. Compute $\nabla^2 g_i(x_{it})$;
2. Assign non-zero structure to `ROW` and `COL` and the corresponding values to `VAL`.

Note:

- Array `X` should not be modified by the user;
- Linear constraints should be specified **after** nonlinear constraints;
- Non-zero indices should be zero based;
- Non-zero structure should be constant;
- **Values should be assigned to lower triangular part of $\nabla^2 f(x_{it})$ only.**

5.2.2 The driver

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

First, the user should declare the user provided functions as follows:

```
EXTERNAL MY_F, MY_F_GRADIENT, MY_F_HESSIAN
EXTERNAL MY_G, MY_G_GRADIENT, MY_G_HESSIAN
```

Now, given the (upper bounds on) values

```
NVARS1, NCONSTR1
```

of

```
NVARS, NCONSTR
```

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

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C DRIVER_NLP.F : 'FORTRAN' DRIVER FOR PENNLP            C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
      INTEGER STATUS, NVARS, NLIN, NCONSTR
      INTEGER NNZ_GRADIENT, NNZ_HESSIAN
      DOUBLE PRECISION XINIT(NVARS1), UOPT(NCONSTR1)
      DOUBLE PRECISION LBV(NVARS1), UBV(NVARS1)
      DOUBLE PRECISION LBC(NCONSTR1), UBC(NCONSTR1)
      DOUBLE PRECISION DOPTIONS(13)
      INTEGER IOPTIONS(17)
      DOUBLE PRECISION DRESULTS(5)
C
C SET DEFAULT OPTIONS
      IOPTIONS(1) = 100
      IOPTIONS(2) = 100
      IOPTIONS(3) = 0
      IOPTIONS(4) = 0
      IOPTIONS(5) = 0
      IOPTIONS(6) = 0
      IOPTIONS(7) = 1
      IOPTIONS(8) = 0
      IOPTIONS(9) = 0
      IOPTIONS(10) = 2
      IOPTIONS(11) = 0
      IOPTIONS(12) = 0
      IOPTIONS(13) = 0
      IOPTIONS(14) = -1
      IOPTIONS(15) = 0
      IOPTIONS(16) = 1
      IOPTIONS(17) = 0
      DOPTIONS(1) = 1.0D-7
      DOPTIONS(2) = 1.0D-0
      DOPTIONS(3) = 1.0D-0
      DOPTIONS(4) = 1.0D-2
      DOPTIONS(5) = 5.0D-1
      DOPTIONS(6) = 1.0D-1
      DOPTIONS(7) = 1.0D-6
      DOPTIONS(8) = 1.0D-12
      DOPTIONS(9) = 1.0D-1
      DOPTIONS(10) = 5.0D-2
      DOPTIONS(11) = 1.0D-0
      DOPTIONS(12) = 1.0D-0
      DOPTIONS(13) = 1.0D-0    ...
    ...
C
C CALL PENNLPF SOLVER LIBRARY
      CALL PENNLPF(NVARS, NLIN, NCONSTR,
     * NNZ_GRADIENT, NNZ_HESSIAN,
     * LBV, UBV, LBC, UBC,
     * XINIT, UOPT,
     * MY_F, MY_F_GRADIENT, MY_F_HESSIAN,
     * MY_G, MY_G_GRADIENT, MY_G_HESSIAN,
     * IOPTIONS, DOPTIONS,
```

```
     * IRESULTS, DRESULTS, STATUS)

      STOP
      END
```

The input/output parameters are explained below.

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

5.3 The input/output structure

`nvars`	number of variables
integer number	
`nconstr`	number of constraints (including linear)
integer number	
`nlin`	number of linear constraints
integer number	
`nnz_gradient`	maximal number of non-zero entries in user specified gradients
integer number	
`nnz_hessian`	maximal number of non-zero entries in user specified Hessians
integer number	
`lbv`	lower bounds on variables
double array	*length*: `nvars`
`ubv`	upper bounds on variables
double array	*length*: `nvars`
`lbc`	lower bounds on constraints
double array	*length*: `nconstr`
`ubc`	upper bounds on constraints
double array	*length*: `nconstr`
`x_init`	on entry: initial guess for the solution on exit: primal solution vector
double array	*length*: `nvars`
`u_opt`	on exit: optimal dual variable (multipliers)
double array	*length*: `nconstr`

`ioptions`	integer valued options (see below)
integer array	*length*: 14
`doptions`	real valued options (see below)
double array	*length*: 13
`iresults`	on exit: integer valued output information (see below) (Not referenced if `iresults = 0` on entry)
integer array	*length*: 4
`dresults`	on exit: real valued output information (see below) (Not referenced if `fresults = 0` on entry)
double array	*length*: 5
`status`	on exit: error flag (see below)
integer array	*length*: 1

IOPTIONS	name/value	meaning	default
ioptions(0)	maxit	maximum numbers of outer iterations	100
ioptions(1)	nwtiters	maximum number of iterations in inner loop	100
ioptions(2)	outlev 0/1 2/3	output level no output/only options are displayed brief output/full output	2
ioptions(3)	hessianmode 0/1	check density of hessian automatic/dense	0
ioptions(4)	autoscale 0/1	automatic scaling on/off	0
ioptions(5)	convex 0/1	convex problem ? no/yes	0
ioptions(6)	eqltymode 0 1 2 3	the way to treat equality constraints as two inequalities, unsymmetric initialization as two inequalities, symmetric initialization handled by standard augmented lagrangian direct handling (all equalities)	3
ioptions(7)	ignoreinit 0 1	ignore initial solutions ? do not ignore ignore	0
ioptions(8)	ncmode 0 1	nonconvex mode Modified Newton Trust Region	0
ioptions(9)	nwtstopcrit 0 1 2	stopping criterion for the inner loop $\|\nabla L(x^{k+1})\|_2 < \alpha$ $\|\nabla L(x^{k+1})\|_2 < \alpha \cdot \|u_i^k - u_i^{k+1}\|_2$ $\|\nabla L(x^{k+1})\|_{H^{-1}} < \alpha \cdot \|\nabla L(x^k)\|_{H^{-1}}$	2
ioptions(10)	penalty 0 1	penalty function logarithmic barrier + quadratic penalty reciprocal barrier + quadratic penalty	0
ioptions(11)	nwtmode 0 1 2 3	mode of solving the Newton system cholesky (standard) cg (with exact hessian) cg (with appr. hessian) cholesky (dual)	0
ioptions(12)	prec 0 1 2 3 4	preconditioner for the cg method no precond diagonal precond bfgs precond appr. inverse precond sgs precond	0
ioptions(13)	cmaxnzs -1 > 0	tuning parameter for Hessian assembling in nwt-mode 1-3 off on	-1
ioptions(14)	autoini 0 1 2	automatic initialization of multipliers off nonlinear (nonconvex) mode lp/qp mode	0
ioptions(15)	penup 0 1	penalty parameter update is performed adaptively after each outer iteration	1
ioptions(16)	usebarrier 0 1 2	box constraint mode no special treatment use (strict) barrier function use (strict) modified barrier function	0

DOPTIONS	name/value	meaning	default
doptions(0)	precision	required final precision	1.0e-7
doptions(1)	uinit	initial multilplier scaling factor	1.0
doptions(2)	pinit	initial penalty	1.0
doptions(3)	alpha	stopping parameter alpha for the Newton/Trust region method in the inner loop	0.01
doptions(4)	mu	restriction factor of multiplier update	0.5
doptions(5)	penup	penalty update	0.1
doptions(6)	peps	minimal penalty	1.0e-8
doptions(7)	umin	minimal multiplier	1.0e-12
doptions(8)	preckkt	precision of the KKT conditions	1.0e-1
doptions(9)	cgtolmin	minimum tolerance of the conjugate gradient algorithm	5.0e-2
doptions(10)	cgtolup	update of tolerance of the conjugate gradient algorithm	1.0e0
doptions(11)	uinitbox	initial multiplier box constraints	1.0e0
doptions(12)	uinitnc	initial multiplier nonlinear constraints	1.0e0

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

DRESULTS	meaning
dresults(0)	primal objective
dresults(1)	relative precision at x_{opt}
dresults(2)	feasibility at x_{opt}
dresults(3)	complementary slackness at x_{opt}
dresults(4)	gradient of augmented lagrangian at x_{opt}

STATUS	meaning
status = 0	converged: optimal solution
status = 1	converged: suboptimal solution (gradient large)
status = 2	converged: solution primal infeasible
status = 3	aborted: no progress, problem may be primal infeasible
status = 4	aborted: primal unbounded or initial multipliers too small
status = 5	aborted: iteration limit exceeded
status = 6	aborted: line search failure
status = 7	aborted: aborted: cholesky solver failed
status = 8	aborted: wrong parameters
status = 9	aborted: resource limit
status = 10	aborted: internal error, please contact PENOPT Gbr (contact @penopt.com)
status = 11	aborted: error in user's memory allocation
status = 12	aborted: error in user supplied routines

6 MATLAB interface

6.1 Calling PENNLPM from MATLAB

6.1.1 User provided functions

The user is required to provide six MATLAB functions. The names of the functions can be chosen by the user; here we use the following names:

`f` ... evaluates the objective function
`df` ... evaluates the gradient of objective function
`hf` ... evaluates the Hessian of objective function
`g` ... evaluates the constraints
`dg` ... evaluates the gradient of constraints
`hg` ... evaluates the Hessian of constraints

The specification of the user-defined functions will be explained using the sample problem (NLP1).

`f`

```
function [fx] = f(x)
fx = x(1)^2 + 4*x(2)^2 - x(3)^2 + x(1)*x(2) - 2*x(1)*x(3);
```

Arguments:

`x`	nx1 matrix storing current iterate x_{it} (input)
`fx`	variable storing $f(x_{it})$ (output)

Description:

- Compute $f(x_{it})$ and store it in `fx`.

Note:

- Matrix `x` should not be modified by the user.

`df`

```
function [nnz,ind,val] = df(x)
nnz = 3;
ind(1) = 1;
ind(2) = 2;
ind(3) = 3;
val(1) = 2.*x(1) + x(2) - 2*x(3);
val(2) = 8.*x(2) + x(1);
val(3) = -2.*x(3) - 2.*x(1);
```

Arguments:

`x`	nx1 matrix storing current iterate x_{it} (input)
`nnz`	variable storing number of non-zeros of ∇f (output)
`ind`	`nnz`x1 matrix storing non-zero structure of ∇f (output)
`val`	`nnz`x1 matrix storing non-zero values of ∇f (output)

Description:

1. Compute $\nabla f(x_{it})$;
2. Assign non-zero structure to `ind` and the corresponding values to `val`.

Note:

- Matrix `x` should not be modified by the user;
- Non-zero indices should be zero based;
- Non-zero structure should be constant.

`hf`

```
function [nnz,row, col, val] = hf(x)
nnz=5;
row(1) = 1;
row(2) = 2;
row(3) = 2;
row(4) = 3;
row(5) = 3;
col(1) = 1;
col(2) = 1;
col(3) = 2;
col(4) = 1;
col(5) = 3;
val(1) = 2.;
val(2) = 1.;
val(3) = 8.;
val(4) = -2.;
val(5) = -2.;
```

Arguments:

`x`	nx1 matrix storing current iterate x_{it} (input)
`nnz`	variable storing number of non-zeros of $\nabla^2 f$ (output)
`row`	`nnz`x1 matrix non-zero row indices of $\nabla^2 f$ (output)
`col`	`nnz`x1 matrix `nnz` non-zero column indices of $\nabla^2 f$ (output)
`val`	`nnz`x1 matrix `nnz` storing non-zero values of $\nabla^2 f$ (output)

Description:

1. Compute $\nabla^2 f(x_{it})$;
2. Assign non-zero structure to `row` and `col` and the corresponding values to `val`.

Note:

- Matrix `x` should not be modified by the user;
- Non-zero indices should be zero based;
- Non-zero structure should be constant;
- **Values should be assigned to lower triangular part of $\nabla^2 f(x_{it})$ only**.

`g`

```
function [gx] = g(i, x)

if i == 0
    gx = x(1)^2 + x(2)^2 + x(3)^2 - 4;
else
    gx = 2*x(1) + 6*x(2) + 4*x(3) - 24;
end;
```

Arguments:

`i`	variable storing constraint number (input)
`x`	nx1 matrix storing current iterate x_{it} (input)
`gx`	variable storing $f(x_{it})$ (output)

Description: Compute $g_i(x_{it})$ and store it in `gx`.

Note:

- Matrix `x` should not be modified by the user;
- Linear constraints should be specified **after** nonlinear constraints.

`dg`

```
function [nnz,ind, val] = dg(i, x)

if i == 0
   nnz=3;
   ind(1) = 1;
   ind(2) = 2;
   ind(3) = 3;
   val(1) = 2.*x(1);
   val(2) = 2.*x(2);
   val(3) = 2.*x(3);
else
   nnz=3;
   ind(1) = 1;
   ind(2) = 2;
   ind(3) = 3;
   val(1) = 2.;
   val(2) = 6.;
   val(3) = 4.;
end;
```

Arguments:

`i`	variable storing constraint number (input)
`x`	nx1 matrix storing current iterate x_{it} (input)
`nnz`	variable storing number of non-zeros of ∇g_i (output)
`ind`	`nnz`x1 matrix storing non-zero structure of ∇g_i (output)
`val`	`nnz`x1 matrix storing non-zero values of ∇g_i (output)

Description:

1. Compute $\nabla g_i(x_{it})$;
2. Assign non-zero structure to `ind` and the corresponding values to `val`.

Note:

- Matrix `x` should not be modified by the user;
- Linear constraints should be specified **after** nonlinear constraints;
- Non-zero indices should be zero based;
- Non-zero structure should be constant.

`hg`

```
function [nnz, row, col, val] = hg(i, x)

if i == 0
   nnz = 3;
   row(1) = 1;
   row(2) = 2;
   row(3) = 3;
   col(1) = 1;
   col(2) = 2;
   col(3) = 3;
   val(1) = 2.;
   val(2) = 2.;
   val(3) = 2.;
else
   nnz = 0;
end;
```

Arguments:

`i`	variable storing constraint number (input)
`x`	nx1 matrix storing current iterate x_{it} (input)
`nnz`	variable storing number of non-zeros of $\nabla^2 g_i$ (output)
`row`	`nnz`x1 matrix non-zero row indices of $\nabla^2 g_i$ (output)
`col`	`nnz`x1 matrix non-zero column indices of $\nabla^2 g_i$ (output)
`val`	`nnz`x1 matrix storing non-zero values of $\nabla^2 g_i$ (output)

Description:

1. Compute $\nabla^2 g_i(x_{it})$;
2. Assign non-zero structure to `row` and `col` and the corresponding values to `val`.

Note:

- Matrix `x` should not be modified by the user;
- Linear constraints should be specified **after** nonlinear constraints;
- Non-zero indices should be zero based;
- Non-zero structure should be constant;
- **Values should be assigned to lower triangular part of $\nabla^2 f(x_{it})$ only.**

6.1.2 The PENNLPM function call

In MATLAB, PENNLPM is called with the following arguments:

```
[f,x,u,status,iresults,dresults] = pennlpm(pen);
```

where

pen . . . the input structure described in the next section

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

status . . . exit information (see section 5.3)

iresults . . . a 4x1 matrix with elements as described in section 5.3

dresults . . . a 5x1 matrix with elements as described in section 5.3

6.2 The `pen` input structure in MATLAB

The user must create a MATLAB structure array with fields described in Section 5.3. Also the names of the user defined functions should be made known in this structure. For the sample problem (NLP1) the structure could look like follows:

```
Infinity = 1.0E38;
pen.nvars = 3;
pen.nlin = 1;
pen.nconstr = 2;
pen.nnz_gradient = 3;
pen.nnz_hessian = 5;
pen.lbv = [0., 0., 0.];
pen.ubv = [Infinity, Infinity, Infinity];
pen.lbc = [0, 0];
pen.ubc = [Infinity,0];
pen.xinit = [2., 2., 2.];
pen.my_f = 'f';
pen.my_f_gradient = 'df';
pen.my_f_hessian = 'hf';
pen.my_g = 'g';
pen.my_g_gradient = 'dg';
pen.my_g_hessian = 'hg';
pen.ioptions = [100 100 2 0 0 0 3 0 0 2 0 0 0 -1 0 1 0];
pen.doptions = [1.0E-7 1.0E0 1.0E-0 1.0E-2 5.0E-1 1.0E-1
         1.0E-6 1.0E-12 1.0E-1 5.0E-2 1.0E0 1.0E0 1.0E0];
```

A sample implementation is included in the files `nlp.m`, `f.m`, `df.m`, `hf.m`, `g.m`, `dg.m` and `hg.m` in directory `matlab`.

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.

[2] R. Fourer, D. M. Gay and B. W. Kernighan. AMPL—a modelling language for mathematical programming. *Scientific Press* , 1993.