Nonlinear Optimization Made Easier: A Tutorial for using the AMPL modelling language with NAG routines.

Jan Fiala*
The Numerical Algorithm Group, Ltd.

February 9, 2011

Abstract

Optimization, or Operational Research in general, nowadays plays an important role in our lives. No matter if you are a respected finance house or a student of mathematics, you have probably used some sort of optimization routines. The field itself has changed rapidly since linear programming was introduced in the mid 1940s. More powerful computers allowed us to consider much more realistic and complex models using sophisticated algorithms. Whereas the input for linear programming problems is relatively simple, it is a much more delicate task in the case of general nonlinear programming. One way to tackle it is to introduce a specialised language for the problem description. In this tutorial we will focus on a particular one called AMPL which we have equipped with two of our NAG solvers, namely E04UFF and E04UGF.

AMPL, A Mathematical Programming Language [3, 8], comes with several nice features packed together and it has earned considerable popularity. The language uses a common mathematical notation so it is easy to understand. Moreover, it includes an automatic differentiation package (a method to compute *exact* derivatives) therefore coding any derivatives can be completely avoided. This makes it ideal whenever you need to solve the problem fast. It is great for demonstrating or teaching as well as for rapid prototyping of mathematical models. Thus you can focus on optimization itself and not coding. It also unifies the interface for both setting the problems and solvers so you can test your solver on several problems or solve the same problem with several solvers without much effort. Thus a connection of AMPL and NAG software seems to be natural as it offers you the power of a modelling language together with NAG well-tuned solvers.

The tutorial is organised as follows. In the first two sections we introduce AMPL and a simple example written in the AMPL language. The 3rd section

*`jan@nag.co.uk`

focuses on the technicalitites of how to get the platform ready for computations. Invoking the solvers and setting optional parameters is the subject of sections 4 and 5. The next section covers more advanced elements of the language which are used in section 7 where a fictitious problem and its model development is presented. The rest of the document overviews building of an AMPL solver from source codes and concluding remarks.

Note: you can download all the files which will be discussed from the NAG website [14] so you can really get your hands into it and test it by yourself.

1 Introducing AMPL

As mentioned earlier, AMPL is a modelling language, a special computer language which allows you to describe various types of optimization problems, such as linear programming, integer programming or nonlinear programming to name a few. The syntax of the language is quite intuitive. The problem is written in algebraic notation based on common mathematical constructions which makes it easy to use even for a non-programmer.

AMPL is also a piece of software, effectively a platform, to load models in the AMPL language, translate them and send them to a compatible solver. The solver in return sends the results back to AMPL where you can display or further analyse them. In this way you can unify inputs for several optimization problems and for various solvers.

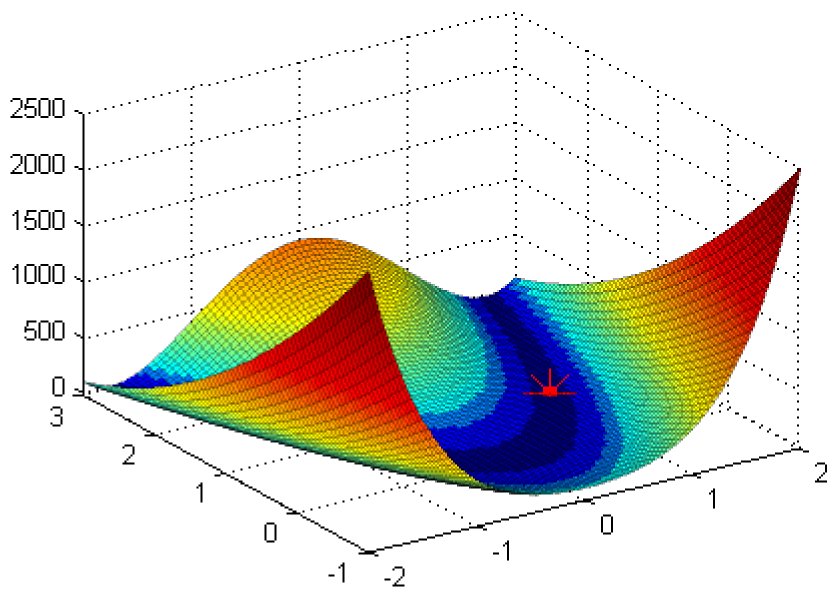
Note that AMPL is not the only environment for describing such problems. You can find several formats for linear or quadratic programming, however, noticably less for general mathematical programming problems. Probably the best known competitor to AMPL is the GAMS language [9].

AMPL started off at Bell Laboratories in the early 1990s and has been extensively developed since then. Nowadays AMPL offers even database access to feed data, definitions of multiple objective functions or create general user functions to expand the elements of the language. We will focus only on the basics of nonlinear programming (NLP) (possibly large-scale), however, if there is any interest, we can describe also how to incorporate black box functions from the NAG S chapter (e.g., Bessel functions) into AMPL or other topics.

2 Writing our first AMPL model

The language is very similar to natural "mathematical" language so there is no extra need to study anything and one can start writing AMPL models straight away. Let's imagine we would like to solve Rosenbrock's banana function [15]. The problem can be written as follows:

$$\min_{x,y\in\mathbb{R}} (1-x)^2 + 100(y-x^2)^2.$$



The global minimum is located at the point $(1,1)$ as indicated on the picture. To solve the problem in AMPL we need first to create an AMPL model. It is a pure text file, here called `rosenbrock.mod`, with the following content:

```
rosenbrock.mod
    var x;
    var y;

    minimize ros_func:
      100*(y - x^2)^2 + (1-x)^2;

    let x:=-1.2;
    let y:=1;
```

The notation is straightforward and self explanatory. The first two statements declare variable names, `minimize` defines our optimization task and objective function (here called `ros_func`). The last two lines are optional and suggest a starting point. Please notice that each statement has to be terminated by a semicolon.

Later on in other models we will define constraints, arrays of variables as well as parameters and data files but now let's solve Rosenbrock's problem.

3 Preparations – getting the AMPL platform ready

We will need two things to get the AMPL environment ready for our computations.

The first bit is an AMPL executable, the platform to load models and invoke solvers. This program is a commercial product. Unless you already have it you will need to download either a testing limited version (for models with up to 300 variables and 300 constraints), called *AMPL Student Edition*, or obtain an unrestricted trial version. For our testing purposes the first one will be sufficient. You can download it from [1] for several architectures, e.g., MS Windows, Linux, MacOS, Solaris. It comes as one gzipped executable file, for example, a direct link for MS Windows version is [2]. Please unzip it to your working directory using the command `gzip -d ampl.exe.gz` or similar. MS Windows users can download the gzip program from [11] or use products with GUI's like 7-zip [17].

The second thing is a solver compiled with an interface for AMPL input. We prepared two NAG routines with such support, E04UF [6] and E04UG [7]. E04UF is suitable for dense nonlinear optimization problems, E04UG for large-scale sparse ones.

Both can be downloaded from the NAG website [14]. They come as executables for MS Windows and Linux, already compiled for your convenience and ready to use. Note: These are intended only for demonstration purposes, not for commercial use. They have the same restrictions as the AMPL Student Edition, i.e., they will not solve models bigger then 300 variables or 300 constraints. In case you prefer a different architecture or an unrestricted version, we have included the source codes so that you could compile such a version with your Fortran or C NAG Library. Alternatively, you can write to us saying which NAG Library you have and we will do our best to help you. For building your own NAG-AMPL solver, see details in Section 9 focused on AMPL interfaces.

Please copy both solvers when you have them ready to your current working directory together with the AMPL executable. You can test them simply by running them without any arguments from your command prompt. You should see a message similar to the following one:

```
C:\honza\math\nag\ampl>e04uf
E04UF, NAG Fortran Library Mark 22
********************************************************
*** This routine is intended for demonstration only. ***
***          Commercial use is not permited.         ***
***     A limited version to problems with up to     ***
***        300 variables and 300 constraints.        ***
********************************************************
Licence: valid (demonstration only)

Based on library:
 *** Start of NAG Library implementation details ***

 Implementation title: NAG Fortran Library for Win32 Applications (DLL)
            Precision: FORTRAN double precision
         Product Code: FLDLL224ML
                 Mark: 22.2 (self-contained)

 *** End of NAG Library implementation details ***
```

```
No test problem set, terminating.
Call ‘e04uf -?‘ for help.
```

4 Loading models, invoking solvers and checking results in AMPL

Now we are ready to solve our first AMPL model, `rosenbrock.mod`. For simplicity, copy it to the same directory as everything else and call `ampl` from your command line to start the AMPL interpreter. You should see the AMPL command prompt

```
ampl:
```

AMPL is now ready for your input but don't forget to add semicolons at the end of each statement. If you forget the prompt will change into

```
ampl?
```

expecting further input (i.e., continuation of the previous line). It is OK just to type `';'` and AMPL will continue.

A typical AMPL session might look like this:

```
ampl: model "rosenbrock.mod";
ampl: option solver e04uf;
ampl: solve;
ampl: display x, y;
ampl: display ros_func;
```

This causes the model to be loaded into memory, and to instruct the solver `e04uf` to solve the problem. The variables and the final objective are then displayed.

If AMPL cannot find the file `rosenbrock.mod` in the current directory an error will be reported, for example:

```
ampl: model "missing_model.mod";
Can't find  file "missing_model.mod"
context:   >>> model "missing_model.mod" <<< ;
```

If this happens, please check the content of the current directory by using commands

```
ampl: shell 'dir';
ampl: shell 'ls';
```

for Windows and Linux/Unix users, respectively.

The consistency and syntax of the file is checked and any error reported. Typical problems might be undefined variables, incorrect spelling of function names or keywords, missing semicolons, brackets, parentheses, and other common programming mistakes. AMPL is also case sensitive, so users of FORTRAN are advised to be extra careful!

The second command chooses the solver, in this case `e04uf`. It means that when the time comes, AMPL will invoke any executable file of that name in the current working directory or in the `PATH` variable. As long as you keep the solver in the working directory and you tested it as suggested in Section 3, it should work fine. Note that Linux users might need to set the full name of the solver including the relative path, for example:

```
ampl: option solver './e04uf.exe';
```

When you hit `solve;` several things will happen. The model currently stored in the memory is translated to a binary file and your solver is invoked and fetched with the file. All the output of the solver is transferred back to the AMPL window so you can typically see the progress of the computation. When the solver computes the results, it generates a result file which is loaded back to AMPL.

To see the results, use the `display` command. It accepts as a parameter any variable names, function names or even mathematical expressions.

Please note that AMPL may shuffle the variables in the model or even eliminate some for efficiency reasons (see AMPL help for `presolve` phase, which can be deactivated). Thus if the solver prints the final values of the variables to the screen, they might be in a different order than the order you specified in the model file! That's why you should always use the `display` command and not rely on the automatic output from the solver.

At this stage you might need to delete the current model from the memory in order to load a new one. Failing to do so can cause the definitions of variables and functions to be combined and probably clash. The command is:

```
ampl: reset;
```

or you may finish the session and return to the command prompt by

```
ampl: quit;
```

If it is necessary to recall how the model loaded into memory looks, use the `show;` command

```
ampl: show;
```

to list all the elements of the model and use the same even to see how each bit is defined, for example:

```
ampl: show ros_func;
```

To resolve the model simply type

```
ampl: solve;
```

but note that the current values of the decision variables serve as a starting point so you may want to change them

```
ampl: let x:=10;
ampl: let y:=-5;
```

5 Optional solver settings

Many optimization solvers come with optional setting parameters to allow a user to fine tune an algorithm. The provided NAG solvers E04UF and E04UG certainly exploit this ability. Fortunately, AMPL offers a convenient way how to deal with them. A general option setting command is as follows,

```
ampl: option solver_options 'keyword1=option keyword2=option ...';
```

where `solver_options` is the name of the option setting string for the specific solver. For solver e04uf such a string is called `e04uf_options`, for e04ug it is `e04ug_options`. The string name is followed by a list of keywords and their values enclosed in apostrophes.

Solver E04UF has the following options implemented (ε stands for the *machine precision*, see [6]):

`feasibility`	Feasibility Tolerance (default $\sqrt{\varepsilon}$)
`maxit`	Major Iteration Limit (default 0, automatic)
`optimality`	Optimality Tolerance (default $\varepsilon^{0.8}$)
`precision`	Function Precision (default $\varepsilon^{0.9}$)
`print`	Print Level (default 10)

Such a list is shown whenever you call the solver with `'-='` as the only argument. Thus from AMPL environment:

```
ampl: shell 'e04uf -=';
```

A full explanation of the settings is presented in the official documentation of the routines [6, 7].

Let's say that we want to solve the previous model with looser precision tolerance and restricted output to the screen. Then call

```
ampl: option e04uf_options 'print=0 precision=1e-3';
```

before you call the `solve` command.

In this way you can easily adjust optional settings and recompute the model without recompiling anything. It is very useful for testing, particularly if you are checking the sensitivity of some parameters from the solver.

You can find below a full listing of the previous AMPL session.

```
ampl: model "rosenbrock.mod";
ampl: option solver e04uf;
ampl: option e04uf_options 'print=0 precision=1e-3';
ampl: solve;
E04UF, NAG Fortran Library Mark 22:
*****************************************************
*** This routine is intended for demonstration only. ***
***          Commercial use is not permited.        ***
***     A limited version to problems with up to     ***
***        300 variables and 300 constraints.        ***
********************************************************
print=0
precision=1e-3

 Calls to E04UEF
 ---------------
      Verify Level = -1
      Function Precision = 1.000000e-03
      Print Level = 0
Result OK
converged: optimal solution found
ampl: display x, y;
x = 1.00015
y = 1.00028

ampl: display ros_func;
ros_func = 4.79927e-08

ampl: quit
```

6 Enhancing AMPL models

In this section, we would like to introduce other elements of the AMPL language to allow users to create more realistic models. Later on we will also use a fictitious problem to demonstrate the biggest strength of AMPL – development of a new model.

Instead of a long theoretical introduction to the AMPL Language, let's jump straight into it and show everything important on an example. We choose a test problem number 72 from the Hock-Schittkowski test suite [12]. Please have a look below at the following constrained optimization problem and its AMPL model

$$\begin{aligned}
\min_{x\in\mathbb{R}^4} \quad & 1+\sum_{j=1}^{4} x_j \\
\text{subject to} \quad & \sum_{j=1}^{4} \frac{a_{i,j}}{x_j} \le b_i, \quad i=1,2 \\
& x_j >= 0.001, \quad j=1,\dots,4 \\
& x_1 \le 400000 \\
& x_2 \le 300000 \\
& x_3 \le 200000 \\
& x_4 \le 100000
\end{aligned} \tag{1}$$

where

$$A = \begin{pmatrix} 4 & 2.25 & 1 & 0.25 \\ 0.16 & 0.36 & 0.64 & 0.64 \end{pmatrix}, \qquad b = \begin{pmatrix} 0.0401 \\ 0.010085 \end{pmatrix}.$$

hs072.mod

```
# Hock-Schittkowski test problem #72
#
# W. Hock, K. Schittkowski, Test Examples for Nonlinear Programming
#   Codes, Lecture Notes in Economics and Mathematical Systems,
#   Springer, Vol. 187, 1981

var x {1..4} >= 0.001;

param a {1..2, 1..4};
param b {1..2};

minimize obj:  1 + sum {j in 1..4} x[j];
subject to constr {i in 1..2}:   sum {j in 1..4} a[i,j]/x[j]<=b[i];
subject to upperb {j in 1..4}:   x[j] <= (5-j)*1e5;

let x[1] := 1;
let x[2] := 1;
let x[3] := 1;
let x[4] := 1;

data;
param a:  1    2    3    4 :=
    1     4 2.25    1 0.25
    2  0.16 0.36 0.64 0.64  ;
param b :=
    1   0.0401
    2   0.010085  ;
```

Although there are many features of the language introduced at once (arrays, constants, constraints, etc.), it should be still intuitively understandable. We will look closer at them now.

Comments might be particularly useful if you work with more complicated models. Any line anywhere in the model starting with character `#` is considered as a comment and thus is ignored.

Scalar (decision) variables have been already introduced in `rosenbrock.mod`. Recall that they are declared via keyword `var` followed by a variable name, such as

```
var my_var;
```

If you wish you can add curly brackets defining a set of indices to transform a scalar variable into a (possibly multi-dimensional) array of variables. For example,

```
var my_var{1..2, -1..1};
```

declares a 2D array of 6 variables which can be referred as `my_var[1,-1]`, `my_var[1,0]`, `my_var[1,1]`, `my_var[2,-1]` and so on. We will see more than once that curly brackets in AMPL define a set of indices whereas square brackets refer to one particular element in the array. In our case line

```
var x {1..4} >= 0.001;
```

declares variables `x[1]` to `x[4]` with an additional bound below. It is a simple way to imply a box constraint to all variables at once. Such bounds can be included also later among other constraints; it is up to user preference.

Arrays and indexing of variables help substantially to simplify some of the expressions. Imagine that we want to sum all our variables. We can write either `x[1]+x[2]+x[3]+x[4]` or just:

```
sum {j in 1..4} x[j]
```

Note that there is a statement for multiplication as well:

```
prod {j in 1..4} x[j]
```

In addition to decision variables, one can also declare constants (parameters of the model). The syntax is the same as with variables, just use keyword `param` in place of `var` and the rest is unchanged. Parameters help in particular in two ways. Firstly, they simplify some of the mathematical expressions such as sums or products. Secondly, it is sometimes desirable to reuse the model for the same problem types. If numbers are kept outside the formulae and parameters are used instead, it is simple to swap the numerical part of the model (in AMPL called *data block*) and the rest will serve well.

Parameters must be initialized before the model can be solved. They are associated either immediately in the declaration or later in the data block. The first is useful if the parameters can be computed directly, for example, based on its indices or other parameters. The following example shows how a Hilbert matrix can be created:

```
param n;
param H {i in 1..n, j in 1..n} := 1/(i+j-1);
```

Whenever `n` is specified in the data block, matrix `H` will be updated automatically to the desired size. It is demonstrated in the following AMPL session:

```
ampl: param n;
ampl: param H {i in 1..n, j in 1..n} := 1/(i+j-1);
ampl: data;
ampl data: param n:=2;
ampl data: display H;

ampl: update data n;
ampl: data;
ampl data: param n:=4;
ampl data: display H;
```

In all other cases you can assign parameters in the data block by listing all their elements. There are various formats how to do it, see reference [8] for details. The two following examples define the same matrix. The first uses tabular data record as it is coded in `hs072.mod`, the other names all nonzero elements, each value preceded by its indices:

```
data;
param a:  1    2    3    4 :=
    1     4 2.25    1 0.25
    2  0.16 0.36 0.64 0.64
       ;
```

```
data;
param a:=
 1 1    4
 1 2    2.25
 1 3    1
 1 4    0.25
 2 1    0.16
 2 2    0.36
 2 3    0.64
 2 4    0.64
 ;
```

The data block starts with the `data;` statement and can be placed either in one file together with the rest of the model (see `hs072.mod`) or in a separate file (see Section 7) which makes swapping data for various problems even easier.

The objective function is given by the `minimize` (or `maximize`) keyword followed by its name and a function definition. The same notation is used for constraints as well. A constraint is expressed by the `subject to` (or just `s.t.`) keyword, its name and the constraint itself. It is also allowed to combine the curly brackets construct and specify a set of constraints in one go.

All the constructions of the AMPL language mentioned above should be enough to understand most AMPL models. If you look closer at the model `hs072.mod` it has 4 decision variables, 1 objective function, 2 nonlinear constraints and 8 simple bounds. The optimal solution is $x = (193.4071,\ 179.5475,\ 185.0186,\ 168.7062)$ with objective function value 727.67937 as reported in [12].

The problem can be invoked in the very same way as the Rosenbrock model:

```
ampl: reset;
ampl: model 'hs072.mod';
ampl: option solver 'e04uf';
ampl: solve;
ampl: display x;
ampl: display obj;
```

Note that E04UF reaches a good match with the reported values.

We might be interested in examining dual variables, slacks or bounds in more complicated models like this one. Simply add behind the name of a variable or constraint `.ub`, `.lb`, `.dual` and `.slack` to display upper bound, lower bound, dual variable and slack variable, respectively. For example, the results of the previous problem will look like:

```
ampl: display x.lb, x, x.ub, x.slack;
:   x.lb       x       x.ub    x.slack    :=
1   0.001   193.407   4e+05   193.406
2   0.001   179.547   3e+05   179.546
3   0.001   185.018   2e+05   185.017
4   0.001   168.707   1e+05   168.706
;
ampl: display constr.lb, constr, constr.ub, constr.dual;
:   constr.lb    constr    constr.ub constr.dual    :=
1   -Infinity    -7692.94   0.0401      -7692.94
2   -Infinity   -41466.8    0.010085   -41466.8
;
```

7 AMPL Case Study – Nonlinear data fit

It is no surprise that the place where AMPL helps most is prototyping a new model. Several changes might be required during the development either because new constraints need to be added to adjust the model closer to reality or simply because you obtained new data. We will demonstrate it on the following fictitious problem.

Imagine that you have an experiment in physics and you know that the measured data should behave as a convex parabolic function. In addition, you expect that the function should be increasing for positive inputs. We can formulate such a relation as a quadratic function with unknown quotients a, b, c with additional constraints on a to impose convexity and on b to limit us to functions increasing for $x > 0$. We will use least-squares for fitting the curve, e.g., minimization of the sum of squared errors. The problem can be written as:

$$\begin{array}{ll}
\displaystyle\min_{a,b,c\in\mathbb{R}} & \displaystyle\sum_{i=1}^{n}(f(x_i) - y_i)^2 \\
\text{subject to} & a \geq 0 \\
 & b \geq 0 \\
\text{where} & \\
 & f(x) = ax^2 + bx + c
\end{array}$$

and pairs $[x_i, y_i]$ are the points of the measurement. Let's use the following 8 points with 8 appropriate measured values as our data:

x_i	y_i
0.5	3.9802
1.0	7.3611
1.5	6.7004
2.0	11.0118
2.5	14.3323
3.0	22.3791
3.5	27.2975
4.0	36.0287

This problem is a typical example in which it can be advantageous to store data separately from the model. If a new measurement set is received we can simply swap data files without touching the model. The model in AMPL language and data file can be written as follows:

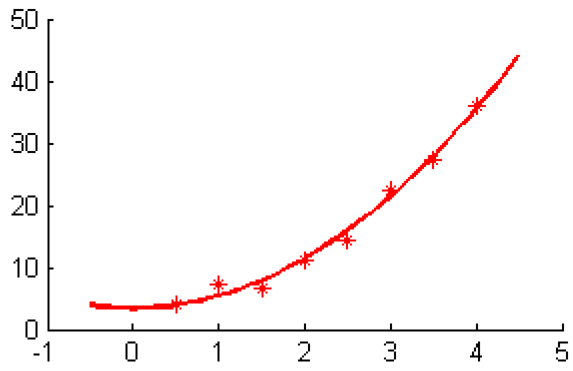
lsfit1.mod
```
param n >= 1;
          # number of measurements
param xpts{1..n};
          # measurement points (x_i)
param D{1..n};
          # Data to fit y_i

var a;
var b;
var c;

minimize obj:
   sum{i in 1..n}
      (a*xpts[i]^2+b*xpts[i]+c-D[i])^2;
subject to constr1:  a>=0;
subject to constr2:  b>=0;
```

lsfit1a.dat
```
data;
param n := 8;
param xpts :=
    1   0.5
    2   1.0
    3   1.5
    4   2.0
    5   2.5
    6   3.0
    7   3.5
    8   4.0 ;
param D :=
    1    3.9802
    2    7.3611
    3    6.7004
    4   11.0118
    5   14.3323
    6   22.3791
    7   27.2975
    8   36.0287 ;
```

We will need to add the command `data filename;` in our typical commands sequence to load the data file. The whole input looks like:

```
ampl: reset;
ampl: model 'lsfit1.mod';
ampl: data  'lsfit1a.dat';
ampl: option solver 'e04uf';
ampl: solve;
ampl: display a,b,c;
ampl: display obj;
```

The computed quotients are $a = 2.00052$, $b = 0$, $c = 3.38308$ with total error (our objective function) 9.615132. See a picture of the fit.



Now imagine that we would like to confirm the results so we will repeat the experiment two more times. We will obtain two more independent sets of data as it is summarised in the following table:

Points x_i	Set 1	Set 2	Set 3
0.5	3.9802	5.4776	3.0925
1.0	7.3611	8.1040	4.0135
1.5	6.7004	8.4392	6.1188
2.0	11.0118	10.748	9.9346
2.5	14.3323	14.8901	13.6188
3.0	22.3791	19.3112	20.0661
3.5	27.2975	23.7691	27.8293
4.0	36.0287	32.2135	37.9609

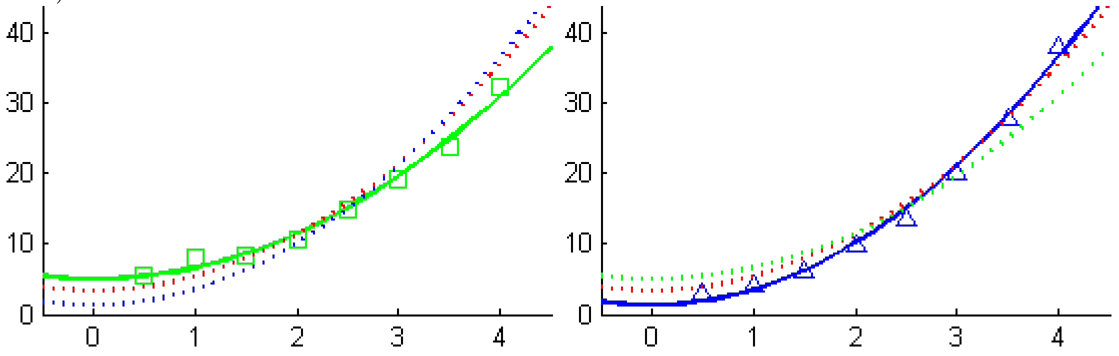
Let's create two more data files `lsfit1b.dat` and `lsfit1c.dat` with the new data and continue our AMPL session:

```
ampl: reset data;
ampl: data lsfit1b.dat;
ampl: solve;
ampl: display a,b,c;
ampl: display obj;
ampl: reset data;
ampl: data lsfit1c.dat;
ampl: solve;
ampl: display a,b,c;
ampl: display obj;
```

Note that you do not need to reload the whole model if you wish to change the data, just use the `reset data;` command. The results are summarised in the table below.

Parameter	Set 1	Set 2	Set 3
`a`	2.00052	1.61829	2.19829
`b`	0	0	0
`c`	3.38308	5.05247	1.31522
`obj`	9.615132	5.72935	7.27976

As we might have expected, each data set produced an individual solution which differs from the other (see graphs, dotted lines reflect the results of the other data sets).



Our new task will be a new model which can handle consistently all data sets so that we obtain one final curve. We could simply merge all the data points and create one big data set but it does not have to be the best solution in our case. Let's say that we want to keep the data from each experiment independent and that we want to minimize the maximal error of each set (so called, worst-case scenario). This can be written as

$$\begin{aligned} \min_{a,b,c\in\mathbb{R}} \ \max_{k=1,2,3} \ & \sum_{i=1}^{n}(f(x_i)-y_i^k)^2 \\ \text{subject to} \quad & a \geq 0 \\ & b \geq 0 \end{aligned}$$

where y_i^k for $k = 1, 2, 3$ are the measured data of each of the sets respectively. We can model it in AMPL by using an extra variable **alpha** which is the maximal error:

lsfit2.mod

```
param n >= 1;              # number of measurement points
param m >= 1;              # number of data sets
param xpts{1..n};          # measurement points (x_i)
param D{1..n,1..m};        # Data to fit y^k_i

var a;
var b;
var c;
var alpha;       # maximal error of the fit for a data set

minimize obj: alpha;
subject to err {j in 1..m}:
   sum{i in 1..n} (a*xpts[i]^2 + b*xpts[i] +c - D[i,j])^2 <= alpha;
subject to constr1:  a>=0;
subject to constr2:  b>=0;
```

lsfit2.dat

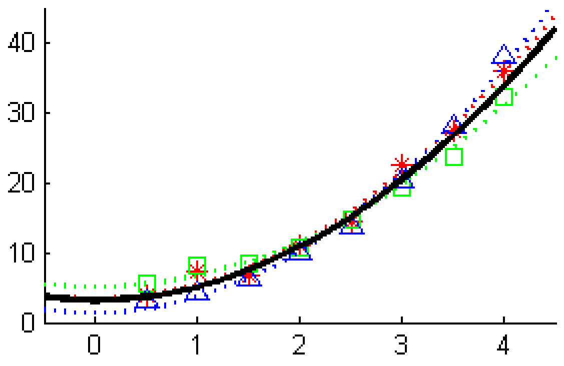
```
 data;
 param n := 8;
```

```
param m := 3;
param xpts :=
 1   0.5
 2   1.0
 3   1.5
 4   2.0
 5   2.5
 6   3.0
 7   3.5
 8   4.0  ;
param D:      1        2        3 :=
 1        3.9802   5.4776   3.0925
 2        7.3611   8.1040   4.0135
 3        6.7004   8.4392   6.1188
 4        11.0118  10.748   9.9346
 5        14.3323  14.8901  13.6188
 6        22.3791  19.3112  20.0661
 7        27.2975  23.7691  27.8293
 8        36.0287  32.2135  37.9609  ;
```

To continue our AMPL session and solve the problem, type as usual:

```
ampl: reset;
ampl: model lsfit2.mod;
ampl: data lsfit2.dat;
ampl: solve;
ampl: display a,b,c;
ampl: display alpha;
ampl: display {i in 1..m}: err[i]+alpha;
```

Note that the last command shows the actual fitting error of each of the data sets. The computed quotients are $a = 1.91428$, $b = 0$, $c = 3.14526$ with total error (`alpha`) 25.2803 and error of each of the fits (`err[i]+alpha`): 25.2803, 24.7906, 24.7699.



This small example should not demonstrate the best way of a nonlinear least-squares fit but our intention was to show one thing only – it is really easy to adjust your model written in a modelling language without much coding. If the model was written in a typical programming language the same changes would require much more effort, especially to incorporate the new objective function and constraints. With AMPL you can simply type it and let the NAG routines do the rest – deliver results. The results are what matter.

8 Additional models

The AMPL language has become de facto a standard interface among solvers for nonlinear programming problems. That is why you can find several test suites for benchmarking written in AMPL. The whole Hock-Schittkowski test set mentioned earlier can be found in AMPL format on a webpage of Professor Robert J. Vanderbei [13].

Another benchmarking suite called COPS (a large-scale Constrained Optimization Problem Set) can be downloaded from [5]. COPS consists of 22 difficult test cases for NLP arising in areas such as population dynamics, shape optimization or fluid dynamics. All can be obtained as AMPL models. This demonstrates well the flexibility of the AMPL language; it is also worth a look just for inspiration.

9 Technical notes

So far we have focused on the user's point of view – how to write a model and solve it in AMPL, however, we avoided describing how to build such a solver compatible with AMPL. Explaining it in detail is far beyond the scope of this text. We will discuss it only to a level that you could compile your own AMPL solver from the provided source codes. If you are interested in developing an AMPL interface by yourself, please follow [10] for details.

As was alluded to in Section 4, the AMPL platform communicates with any solver via files. Your model file is converted into a binary file and the solver gets its name as a command line argument together with optional parameters (string `solver_options`). Once the optimization run finishes, the solver generates another file with the results which is read in by AMPL.

If you want to create an AMPL-compatible solver you will need 3 main ingredients:

- the solver itself; a routine which is capable of solving your type of problems
- a set of routines which can read the binary file on the input, treat the model stored there and vice versa can generate the results file at the end
- a main routine which glues both together.

In our case the solvers are either E04UFF and E04UGF (from the NAG Fortran Library) or E04UCC and E04UGC (from the NAG C Library) depending on the library you have installed. The AMPL developers provide a library called *AMPL*

Solver Library (ASL) which takes care of the input, output and evaluation of the functions in the model. You can download its source codes free of charge from [4]. Finally, files `e04uff_ampl.c`, `e04ugf_ampl.c`, `e04ucc_ampl.c` and `e04ugc_ampl.c` create the intermediate layer between ASL and the NAG Library. They are available on the NAG website [14].

You will need to download and build ASL. There are scripts `configure` and `configurehere` to choose the right `makefile` for your platform. Please follow the readme file and instructions in the makefile for building the library. If you want you can test ASL with some small example programs [16]. It should indicate whether the ASL build was successful or not.

There are several makefiles ready to compile together your NAG Library, ASL build and one of the supplied routines. For example, `Makefile.fldll224ml.cl` is intended for NAG Fortran Library FLDLL224ML (Mark 22, Windows DLL 32-bit) and Microsoft C++ compiler (`cl`). Before you use it it is necessary to edit the appropriate lines of the file defining the path to the NAG Library and ASL, for example,

```
NAG_LIBDIR = C:\math\NAG\FL22\fldll224ml
AMPL_LIBDIR = C:\honza\math\ampl\amplsolver\sys.cl
```

If you prefer you can edit other parts as well (e.g., compiler flags), however, it should work as it is. The table below summarizes the systems we have tested.

NAG Library code	System	Compilers
CLDLL084ZL	Win32	cl
CLL6A08DGL	Linux64	gcc
CLL6A09DHL	Linux64	gcc
CLLUX08DCL	Linux32	icc
CLLUX08DGL	Linux32	gcc
CLW3209DAL	Win32	cl, gcc
FLDLL214AL	Win32	cl, gcc
FLDLL224ML	Win32	cl, gcc
FLL3A21DFL	Linux32	gcc
FLL3A22DFL	Linux32	gcc
FLL6A22DFL	Linux64	gcc
FLL6I22DCL	Linux64	gcc, icc
FLLUX21DGL	Linux32	gcc
FLLUX22DCL	Linux32	gcc

In case your NAG Library/system is not covered, there are two generic makefiles to help you write your own. Please check your User's Note supplied together with the NAG Library for implementation details if necessary.

To build the solver invoke `make` (or `nmake`) as it is described in the comments in the makefile, for example:

```
C:\honza\math\nag\ampl\src>nmake -f Makefile.fldll224ml.cl all
```

```
Microsoft (R) Program Maintenance Utility Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.

   cl /O2 /nologo /I"C:\honza\math\ampl\amplsolver\sys.cl" /DUSE_STDCALL
   e04uff_ampl.c "C:\honza\math\ampl\amplsolver\sys.cl\amplsolv.lib"
   "C:\math\NAG\FL22\fldll224ml\lib\FLDLL224M_nag.lib"  /Fee04uf.exe
e04uff_ampl.c
   Creating library e04uf.lib and object e04uf.exp

   cl /O2 /nologo /I"C:\honza\math\ampl\amplsolver\sys.cl" /DUSE_STDCALL
   e04ugf_ampl.c "C:\honza\math\ampl\amplsolver\sys.cl\amplsolv.lib"
   "C:\math\NAG\FL22\fldll224ml\lib\FLDLL224M_nag.lib"  /Fee04ug.exe
e04ugf_ampl.c
   Creating library e04ug.lib and object e04ug.exp
```

Once you build your own solver with an AMPL interface, you should test it as suggested in Section 3 or with an embedded test in the makefile:

```
C:\honza\math\nag\ampl\src>nmake -f Makefile.fldll224ml.cl test

Microsoft (R) Program Maintenance Utility Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.

##############################################################
# Mini test of e04uf.exe
# Test1: Routine identification & licence check
#   the result should start with:
#     E04UF, NAG Fortran Library
#     Licence: valid
# Test2: Simple test problem, Hock-Schittkowski #72
#   an approximate results are:
#     x = [193.4, 179.5, 185.0, 168.7]
#     objective function = 727.679
##############################################################

#########################  TEST 1  #########################
        e04uf.exe
E04UF, NAG Fortran Library
Licence: valid
...
```

In case of any difficulties, please verify that the build process was completed without any errors and that the NAG library is present in your `PATH` variable (for Windows) or `LD_LIBRARY_PATH` (for Linux). The build should be similar to any other build of a NAG Library application.

Note that we unfortunately cannot provide you with any support regarding the ASL library, however, we may be able to help build an interface specific to your library version. Please do not hesitate to contact us.

10 Conclusion

We hope you enjoyed the taster session of AMPL and NAG optimization routines. We covered mostly the basics of the AMPL language but it should be enough to try it by yourself and see how useful AMPL (or other modelling languages) would be for you.

The variability of the models is probably AMPL's main strength together with the unified interface across various solvers and problems. This could save you a substantial amount of time during the first phase of development. It can also be beneficial for demonstrating and teaching as you don't need to code much to receive the results. It is apparent that AMPL is not a solution for everything, however, even if you want to incorporate NAG solvers into a bigger piece of software, AMPL can be handy with initial prototyping and testing of the mathematical model.

Let us know what you think. We would love to hear from you, your comments and experience. Do not forget that you can download all the files from the NAG website. Have you encountered any problems? Do you use AMPL or other modelling languages? Would you like to see AMPL interfaces for some routines accompanying the library? Thanks in advance for your kind replies.

References

[1] AMPL executables (Student Edition)
`http://netlib.sandia.gov/ampl/student/index.html`

[2] AMPL executable for MS Windows (Student Edition)
`http://netlib.sandia.gov/ampl/student/mswin/ampl.exe.gz`

[3] AMPL main website
`http://www.ampl.com/`

[4] ASL – AMPL Solver Library, source codes
`http://netlib.sandia.gov/ampl/solvers/index.html`

[5] COPS, a large-scale Constrained Optimization Problem Set
`http://www.mcs.anl.gov/~more/cops/`

[6] *E04UFF*, NAG Fortran Library Manual, Mark 22
`http://www.nag.co.uk/numeric/FL/nagdoc_fl22/pdf/E04/e04uff.pdf`

[7] *E04UGF*, NAG Fortran Library Manual, Mark 22
`http://www.nag.co.uk/numeric/FL/nagdoc_fl22/pdf/E04/e04ugf.pdf`

[8] Fourer R., Gay D. M., Kernighan B. W.: *AMPL: A Modeling Language for Mathematical Programming*, 2nd. ed, 2002. Brooks/Cole Publishing Company.

[9] GAMS main website
`http://www.gams.com/`

[10] Gay D. M.: *Hooking Your Solver to AMPL*, Technical report, 1997. Bell Laboratories, Murray Hill, NJ.
`http://www.ampl.com/REFS/hooking2.pdf`

[11] gzip website and Windows binary
`http://www.gzip.org/`
`ftp://tug.ctan.org/tex-archive/tools/zip/info-zip/MSDOS/gzip124.exe`

[12] Hock W., Schittkowski K.: *Test Examples for Nonlinear Programming Codes*, Lecture Notes in Economics and Mathematical Systems, No 187, 1981. Springer. Schittkowski K.: *More Test Examples for Nonlinear Programming Codes*, Lecture Notes in Economics and Mathematical Systems, No 282, 1987. Springer.
`http://www.math.uni-bayreuth.de/~kschittkowski/tpnp08.htm`

[13] Hock-Schittkowski test collection as AMPL models
`http://www.princeton.edu/~rvdb/ampl/nlmodels/hs/`

[14] NAG Technical Reports website
`http://www.nag.co.uk/doc/techrep/index.asp`

[15] *Rosenbrock function*, Wikipedia.
`http://en.wikipedia.org/wiki/Rosenbrock_function`

[16] Simple examples for ASL, source codes
`http://netlib.sandia.gov/ampl/solvers/examples/index.html`

[17] 7-zip website
`http://www.7-zip.org/`