The NOP-2 modeling language

H. Schichl A. Neumaier S. Dallwig

Institut für Mathematik, Universität Wien, Strudlhofg. 4, A-1090 Wien, Austria
E-mail: Hermann.Schichl@esi.ac.at, neum@cma.univie.ac.at, sdal@cma.univie.ac.at

An enhanced version NOP-2 of the NOP language for specifying global optimization problems, defined in [12], is described. Because of its additional features NOP-2 is comparable to other modeling languages like AMPL [6] and GAMS [1], and allows the user to define a wide range of problems arising in real life applications such as global constrained (and even stochastic) optimization programs.

NOP-2 permits named variables, parameters, indexing, loops, relational operators, extensive set operations, matrices and tensors, and parameter arithmetic.

The main advantage that makes NOP-2 look and feel considerably different from other modeling languages is the display of the internal analytic structure of the problem. It is fully flexible for interfacing with solvers requiring special features such as automatic differentiation or interval arithmetic.

Keywords: large-scale optimization, global optimization, separable structure, nonlinear programming, modeling language

AMS Subject classification: 90C30

1. Introduction

This paper describes the modeling language NOP-2 for specifying general optimization problems, including constrained local or global nonlinear programs and constrained stochastic single and multistage programs. The proposed language is specifically designed to represent the internal (partially separable, repetitive, or sparse) structure of the problem. Thus it enables the model builder to somewhat step back from the pure meaning of the problem to focus on the mathematical content and to rewrite it in a form that is more adapted to global and large scale optimization algorithms that can exploit such structure.

Models described in NOP-2 do not depend on the computer architecture. Therefore, models developed on personal computers can later be solved on high speed workstations or supercomputers.

The language also has features for checking correct specification of NOP-2 files.

In contrast to the SIF input format (cf Section 4.1 below) proposed by Conn, Gould, and Toint [3] for their LANCELOT package, the amount of overhead in the formulation of smaller problems is very small: for example, Rosenbrock's function (in SIF the description takes almost a page) can be represented in a few lines in such a way that the least squares structure is visible in the representation.

Together with planned and partly finished interfaces to the modeling languages AMPL (Fourer, Gay & Kernighan [6]) and GAMS (Brooke, Kendrick, Meeraus [1]), to our new global optimization code GLOPT [4], and to the optimization package MINOS (Murtagh & Saunders [11]), this is a promising tool for the formulation and solution of various types of optimization problems.

Each *NOP-2 file* consists of a sequence of *records* describing an optimization problem of one of the following forms.

(i) *Nonlinear programs.*

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } E_\nu(x), \quad \nu = 1, \dots, N; \quad x \in [x_0]. \end{aligned}$$

The bound constraints $x \in [x_0]$ define componentwise restrictions $\underline{x}_0 \le x \le \overline{x}_0$, and may contain $\pm\infty$ as bounds to allow for one-sided bounds and free variables.

(ii) *Stochastic programs.*

$$\begin{aligned} &\min f(x, \xi) \\ &\text{s.t. } E_\nu(x, \xi), \quad \nu = 1, \dots, N; \quad x \in [x_0], \\ &\quad \xi \sim g(b, x). \end{aligned}$$

The variables ξ are stochastic variables with (fully or partially specified) distribution functions $g(b, x)$.
(iii) *Stochastic multistage programs.*

$$\begin{aligned}&\min f(x^k, \xi^k)\\ &\text{s.t. } E^k_\nu(\xi^k, \xi^{<k}, x^{<k}), \quad \nu = 1, \ldots, N^k; \quad x^k \in [x^k_0],\\ &\qquad \xi^k \sim g^k(b, x^{<k}),\end{aligned}$$

The variables ξ^k are stochastic variables with (fully or partially specified) distribution functions $g(b, x)$ and are valid in stage k of this multistage problem.

In each case, additional integer constraints can be specified (see **variable declarations**). The so-called *elements* $E_\nu(x)$ are expressions of one of the forms

$$\bigodot_k f(a_k, x_{J_k}) \in b[q] + c, \tag{1.1}$$

$$\bigodot_k f(a_k, x_{J_k}) = b x_{K_j} + c, \tag{1.2}$$

$$\bigodot_k f(a_k, x_{J_k}) \in S, \tag{1.3}$$

and a few irregular variants consisting of only one operand, allowing simple coding of Boolean expressions, polynomials, trigonometric polynomials, and a limited form of branching.

Here f is a so-called *element function*, a_k, b, c are parameters, parameter vectors, parameter matrices or higher order tensors, or lists of such. x_{J_k}, and x_{K_j} are subvectors of x indexed by the index lists J_k and K_j, $[q]$ is a possibly unbounded box, maybe restricted to integers, and S is a union of finite sets and possibly unbounded boxes. $\odot$ specifies one of the following operators: $\sum$, $\prod$, max, min, $\sum_k (-1)^k$. The contributions $f(a, b_k, x_{J_k})$ are referred to as the *pieces* of the element. (Elements containing a single piece only are, of course, permitted.)

The element functions that we found most handy in coding a large number of problems are collected in a standard library `nop_stdlib.nop` that is visible to every NOP-2 specification by default. For all these functions it is possible to get a complete analytic overview over ranges and inverse ranges, which makes these elements suitable for applications in a branch and bound framework such as that proposed in [4]. Other element functions can easily be defined using algebraic statements in a syntax similar to some higher level programming languages like FORTRAN, C or MATLAB.

Work on a package that reads NOP-2 files and generates subroutines useful for feeding into nonlinear programming routines is almost finished. In particular, for all standard and user defined functions the final package will contain a tool that automatically generates C code for function and gradient evaluation, bounds for ranges and inverse ranges, linear enclosures, and underestimating quadratic functions.

2. Basic philosophy of language construction

The NOP-2 modeling language was constructed with three major goals in mind, which differ from the philosophy followed by other languages like AMPL and GAMS.

First, the language should enable and encourage the modeler to explicitly specify the model's inherent structure. Here the *modeler* should have as much flexibility as possible. He should be able to extract common subexpressions and to make partial separability obvious. At the same time, the language should provide the *solver* with as much information about the optimization problem as possible, giving near optimal interval enclosures, quadratic underestimators, convexity checks, inverse enclosures,...

These goals lead us to the introduction of element functions as basic building blocks. The *standard library* was developed to contain basic elements that appear in many common optimization problems and test sets. At the same time, it became clear that a fixed number of element functions might not be sufficient to specify all problems. That was the reason for making the language extensible by the user.

The third goal was to design a language that introduces as little overhead as possible, provided that the goals are reached. This lead us to the decision to choose a *record structure* to specify the model. A summary of the format is given in Section 5.

3. Examples

Before describing the language in a short informal way we give some examples for later reference. (This section is adapted from [12], where the same examples were treated with the precursor language NOP.) The first three examples are taken from the global optimization test sets of [7,15,5].

We have coded many of the test problems in these collections to ensure that our input format is easy to use, and does not require too much repetitive or error-prone adaptation, given the original mathematical description. The corresponding NOP-2 files will be made publicly available via the second author's global optimization pages on the World Wide Web. The URL of the relevant section will be

```
http://solon.cma.univie.ac.at/
    ~/neum/glopt/nop-2.html
```

3.1. The Rosenbrock function

Our first problem [7] is the minimization of the well-known Rosenbrock function in a box,

$$\begin{aligned} &\min\ 100(x_1^2 - x_2)^2 + (x_1 - 1)^2 \\ &\text{s.t.}\ \ x_1, x_2 \in [-2, 8]. \end{aligned}$$

The least squares structure becomes apparent by introducing the variable

$$x_3 = 10x_1^2 - 10x_2, \tag{3.1}$$

reducing the objective function to

$$x_4 = x_3^2 + (x_1 - 1)^2. \tag{3.2}$$

Since quadratic programs are very important in global optimization, both equations can be expressed easily by standard elements: (3.1) corresponds to the predefined element function `qu4` (with two pieces) and (3.2) corresponds to the predefined element function `qu2` (again with two pieces). If we now remember the bounds, we end up with the following NOP-2 file. (The lines starting with `//` are comments added only to make the file more readable.)

```
// Rosenbrock function
min x[4];
bnd x[1 2] in [-2,8];
// element list
qu4 x[1 2]; 0 -10 10 0 = x[3];
qu2 x[3 1]; 0 1 = x[4];
```

3.2. Another least squares problem

Our second problem [15] is a least-squares problem of Kowalik with additional bound constraints. It has been slightly changed from the original version published in [9, 6.2]. There instead of the b_i^{-1} the b_i (called y_i) were specified up to four significant digits. The data

i	a_i	b_i^{-1}
1	0.1957	0.25
2	0.1947	0.50
3	0.1735	1.00
4	0.1600	2.00
5	0.0844	4.00
6	0.0627	6.00
7	0.0456	8.00
8	0.0342	10.00
9	0.0323	12.00
10	0.0235	14.00
11	0.0246	16.00

define the following problem

$$\begin{aligned} &\min \sum_{i=1}^{11} \left(a_i - x_1 \frac{b_i^2 + b_i x_2}{b_i^2 + b_i x_3 + x_4} \right)^2 \\ &\text{s.t.}\ \ x_i \in [0, 0.42] \quad (i = 1, 2, 3, 4), \end{aligned}$$

To model the least squares terms we introduce a new nonstandard element function

$$\mathrm{quot}(p, x_1, \ldots, x_4) = \frac{x_1(1 + px_2)}{1 + px_3 + p^2 x_4} \tag{3.3}$$

reducing the objective function to

$$\sum_{i=1}^{11} (a_i - \mathrm{quot}(b_i^{-1}, x_1, \ldots, x_4))^2. \tag{3.4}$$

After introducing new variables for the results of `quot` (that define elements with a single piece only), (3.4) reduces further to an element with predefined element function `qu2` and 11 pieces. The result is the following NOP file.

```
// Kowalik problem
// variable definitions
var obj;
// constant definitions
const a = (0.1957 0.1947 0.1735 0.1600
           0.0844 0.0627 0.0456 0.0342
           0.0323 0.0235 0.0246);
const binv = (0.25 0.5 1 2 4 6 8 10 12 14 16);

//basic problem specification
min obj;
bnd x[1:4] in [0,.42];
func quot(real y[4]; real p)
  y[1]*(1+p*y[2])/(1+p*(y[3]+p*y[4]));
```

```
endfunc
// element list
qu2 x[5:15]; a = obj;
loop(i=1:11)
  quot x[1:4]; 1/binv[i] = x[4+i];
endloop
```

With a little more work, we can eliminate the user-defined element function by introducing the constants

$$c_i = b_i^{-1}$$

and writing

$$x_{i+4} = x_1 \frac{b_i^2 + b_i x_2}{b_i^2 + b_i x_3 + x_4} = \frac{x_1(1 + c_i x_2)}{1 + c_i x_3 + c_i^2 x_4}.$$

After multiplication with the denominator, we may write this as

$$x_{i+4} + c_i x_{i+15} = x_1,$$

where

$$x_{i+15} = x_{i+4} x_3 + c_i x_{i+4} x_4 - x_1 x_2.$$

These constraints can be handled with linear and bilinear elements, resulting in the following NOP file.

```
// Kowalik problem
// variable definitions
var obj;
// constant definitions
const a = (0.1957 0.1947 0.1735 0.1600
           0.0844 0.0627 0.0456 0.0342
           0.0323 0.0235 0.0246);
      c = (0.25 0.5 1 2 4 6 8 10 12 14 16);

//basic problem specification
min obj;
bnd x[1:4] in [0,.42];

// element list
qu2 x[5:15]; a = obj;
loop(i=1:11)
 lin x[4+i 15+i]; 1 c[i] = x[1];
 qf1 x[4+i 3 4+i 4 1 2]; 1 c[i] -1 = x[15+i];
endloop
```

In the second version of the problem the non-standard element function has been removed by introducing new intermediate variables. In NOP-2 formulations this can always be done. Increasing the dimension in this way sometimes significantly improves performance of a subsequent optimization, but the performance may deteriorate, too. Therefore, such transformations should be used with care.

3.3. A quadratically constrained nonlinear program

Our third problem [5] is a nonlinear program with bilinear constraints,

$$\begin{aligned}
&\min\ x_1 + x_2 + x_3 \\
&\text{s.t.}\ \ -1 + 0.0025(x_4 + x_6) \le 0, \\
&\qquad -1 + 0.0025(-x_4 + x_5 + x_7) \le 0, \\
&\qquad -1 + 0.01(-x_5 + x_8) \le 0, \\
&\qquad 100x_1 - x_1 x_6 + 833.33252 x_4 - 8333.333 \le 0, \\
&\qquad x_2 x_4 - x_2 x_7 - 1250 x_4 + 1250 x_5 \le 0, \\
&\qquad x_3 x_5 - x_3 x_8 - 2500 x_5 + 1250000 \le 0, \\
&\qquad 100 \le x_1 \le 10000, \\
&\qquad 1000 \le x_2 \le 10000, \\
&\qquad 1000 \le x_3 \le 10000, \\
&\qquad 10 \le x_4 \le 1000, \\
&\qquad 10 \le x_5 \le 1000, \\
&\qquad 10 \le x_6 \le 1000, \\
&\qquad 10 \le x_7 \le 1000, \\
&\qquad 10 \le x_8 \le 1000.
\end{aligned}$$

We introduce a variable for the objective function,

$$x_9 = x_1 + x_2 + x_3,$$

using an element of the form sum with 3 pieces. The first three constraints (after adding 1 and dividing by the factor, though these cosmetic operations could be dispensed with) become elements of the form `lin` with 2, 3, and 2 pieces, respectively. The three bilinear constraints could be handled directly using the predefined element `bil`, but seeing that there are common factors, and the factor $x_5 - x_8$ is common to the third and the 6th constraint, we chose to introduce extra variables

$$x_{10} = x_1(x_6 - 100), \quad x_{11} = x_4 - x_7, \tag{3.5}$$

$$x_{12} = x_2 x_{11}, \quad x_{13} = x_8 - x_5, \quad x_{14} = x_3 x_{13} \tag{3.6}$$

using the elements `pr2`, `lin`, and `pr0`. After the corresponding substitution this leaves only linear constraints coded by the element function `lin`. The result is the following NOP file. (The third constraint reduces to a bound constraint, and the definition of x_{13} appears after the bound declaration.)

```
// Hock - Schittkowski Problem 106
// = Floudas - Pardalos Chapter 3,
```

```
//              Test Problem 1
real obj;

min obj;
bnd x[1] in [100,10000];
bnd x[2 3] in [1000,10000];
bnd x[4:8] in [10,1000];
bnd x[13] <= 100;
// element list
sum x[1:3] = obj; // objective function
sum x[4 6] <= 40;
lin x[4 5 7]; -1 1 1 <= 40;
lin x[5 8]; -1 1 = x[13];
pr2 x[1 6]; 0 -100 = x[10];
lin x[10 4]; -1 833.33252 <= 83333.333;
lin x[4 7]; 1 -1 = x[11];
pr0 x[2 11] = x[12];
lin x[12 4 5]; 1 -1250 1250 <= 0;
pr0 x[3 13] = x[14];
lin x[14 5]; -1 -2500 <= -1250000;
```

3.4. A multiphase chemical mixture problem

For maximal efficiency, we strongly advise formulating problems in such a way that the element functions depend on only a few variables. All significant summations should appear either explicitly in the element structure or in matrix-vector expressions; the latter should be used only when the matrices involved are either essentially dense or defined in one of the sparse matrix formats provided. The formal increase of dimension resulting from introducing extra variables and equality constraints to achieve this is usually much less relevant than the gain in structural information, even for problems that are originally unconstrained.

We exemplify this by considering a global optimization problem originally investigated by Mfayokurera [10], describing the mixture of n chemical species at fixed temperature, pressure, and mole number y_i^0, $i = 1, \ldots, n$, separating in up to n phases. (The objective function is based on the NRTL model of Renon & Prausnitz [13] and the Gibbs free energy formulation of Castillo & Grossmann [2].)

$$\begin{aligned} \min\ & \sum_{i,k=1}^{n} y_i^k \log \frac{y_i^k}{\sum_{j=1}^n y_j^k} + \sum_{i,k=1}^{n} y_i^k \frac{\sum_{j=1}^n A_{ji} y_j^k}{\sum_{j=1}^n B_{ji} y_j^k} \\ \text{s.t. } & \textstyle\sum_{k=1}^n y_i^k = y_i^0 \quad (i = 1, \ldots, n), \\ & 0 \le y_i^k \le y_i^0 \quad (i, k = 1, \ldots, n), \\ & y_1^k \le y_1^{k+1} \quad (k = 1, \ldots, n-1). \end{aligned} \tag{3.7}$$

The B_{jk} are nonnegative numbers, and $A_{ii} = 0$, $B_{ii} = 1$, making the problem well-defined. However, extra care is needed to cope with the limit expressions $0 \log 0$ and $0 \frac{v}{0}$ that may occur.

To obtain an element formulation, we introduce extra variables for the inner sums by adding the constraints

$$u^k := \sum_{j=1}^{n} y_j^k, \tag{3.8}$$

$$v_i^k := \sum_{j \ne i}^{n} A_{ji} y_j^k, \tag{3.9}$$

$$w_i^k := \sum_{j \ne i}^{n} B_{ji} y_j^k, \tag{3.10}$$

and expand the expressions

$$\log \frac{y_i^k}{u^k} = \log y_i^k - \log u_k.$$

If we now notice that

$$\operatorname{xlog}(y) := \begin{cases} y \log y \text{ if } y > 0, \\ 0 \text{ otherwise} \end{cases} \tag{3.11}$$

is one of the predefined element functions and introduce the new non-standard function

$$\operatorname{frac}(x, y, z) := \begin{cases} xy/(y+z) \text{ if } y > 0, \\ 0 \text{ otherwise,} \end{cases} \tag{3.12}$$

we can write the objective function as a sum of three elements

$$f = \sum_{i,k=1}^{n} \operatorname{xlog}(y_i^k) - \sum_{k=1}^{n} \operatorname{xlog}(u^k) + \sum_{i,k=1}^{n} \operatorname{frac}(v_i^k, y_i^k, w_i^k). \tag{3.13}$$

Since the first and the last sum in (3.13) have the same structure, one can improve the formulation further by introducing another element function

$$\operatorname{term}(x, y, z) := \begin{cases} y \log y + xy/(y+z) \text{ if } y > 0, \\ 0 \text{ otherwise} \end{cases} \tag{3.14}$$

and simplify (3.13) to a sum of two elements

$$f = -\sum_{k=1}^{n} \operatorname{xlog}(u^k) + \sum_{i,k=1}^{n} \operatorname{term}(v_i^k, y_i^k, w_i^k). \tag{3.15}$$

It is now easy to code this as a NOP-2 file. Note that since the coefficients A_{ji} and B_{ji} appear in several of the new constraints, they should be predefined as constants. Then one can specify (3.9) and (3.10) as elements of type `lin`.

```
/*
** NRTL model of RENON and PRAUSNITZ
** and Gibbs free energy formulation
**     of CASTILLO and GROSSMANN
*/
real f;
/*
** size of the problem and other parameters
** (these should be specified later)
** (the later specifications will be checked
** against these template definitions)
*/
param int n;
param real A[n,n];
param real B[n,n];
param real y0[n];
/*
** include the definitions of the matrices
** A, B and the vector y0 from matrices.nop.
** This file should also contain an assignment
** to n!
*/
incl "matrices.nop"

real y[n,n] u[n] v[n,n] w[n,n]
     xlogs terms[n];

func term(real x[#] y[#] z[#])
  // an element function with a
  // varying number of arguments
  apply(+,i=1:#)
    if(y[i]>0)
      y[i]*log(y[i])+x[i]*y[i]/(y[i]+z[i]);
    else
      0;
    endif
  endapply
endfunc

// parameter checks
// forces the parser to check the
// basic problem requirements
bnd B[:,:] > 0;
loop(i=1:n)
  bnd A[i,i] = 0;
  bnd B[i,i] = 1;
endloop

// basic problem specification
min f
loop(i=1:n)
  bnd y[:,i] in [0,y0[i]];
endloop
// elements
sum xlogs terms = f;
xlog u = xlogs;
loop(i=1:n)
  sum y[:,i] = y0[i];
  sum y[i,:] = u[i];
  mv y[i,:] A = v[i,:];
  mv y[i,:] B = w[i,:];
  term v[i,:] y[i,:] w[i,:] = terms[i];
endloop
loop(i=1:n-1)
  lin y[i,1] y[i+1,1]; 1 -1 <= 0;
endloop
```

This problem demonstrates that parametrized models can easily be specified in NOP-2. The `param real...` statements specify templates for the variables that should be specified in the include file `matrices.nop`. Such an include file could look like follows:

```
// matrices.nop
// include file for matrix specification
const n = 4;
// A is a dense matrix entered row-wise
const A = dense[row](n,n)
 0 -1.3 0.02 1.03
 -1  0  3.2   -4
 2.1 -.03  0 -1.24
 7.3  4.9 -.01  0;
/*
** B is a band matrix with one band above
** and one band below the main diagonal. The
** bands are specified starting with the main
** diagonal followed by alternating upper
** and lower subdiagonals.
*/
```

```
const B = band[inout](1,1)
 1  1  1  1
 0.1 0.59 1.4
 5.6 0.47 8.3;

const y0 = (1.9 3.8 4.7 2.43);
```

4. Connections to other modeling languages

As mentioned in Section 2, the basic philosophy of NOP-2 is different from that of other modeling languages.

Both AMPL and GAMS put maximal emphasis on the convenience of the modeler.

SIF, in contrast, the input language for the local optimizer LANCELOT, provides lots of information for the solver, but has an exceptionally complicated format.

In designing NOP-2, we tried to find an intermediate way. The solver should be provided with as much structural information as possible. On the other hand, specifying the problem should not be a too formidable task. In NOP-2, apart from knowing the syntax, the modeler only needs to look up the names of the basic element functions in a big (and user-extensible) table.

A conversion routine from AMPL to NOP-2 called `ampl2nop` is under development, and it will be described in a subsequent report. A main problem in automatic conversion is that it is not well-defined where to introduce intermediate variables and what common subexpressions to use for new function definitions. This must be decided by some heuristics; but the intelligence of the modeler in detecting structure can hardly be matched by a conversion algorithm.

The basic steps in converting specifications from SIF, AMPL and GAMS modeling languages mentioned are described below.

4.1. Conversion from SIF

The SIF format used as input for the LANCELOT package [3] handles nonlinear programs of the form

$$\begin{aligned} &\min \sum_{i\in I_0} g_i\left(\sum_{j\in J_i} w_{ij} f_j(x_{K_j}) + a_i^T x - b_i\right) \\ &\text{s.t. } l_i \le x_i \le u_i \qquad (1 \le i \le n), \\ &\qquad g_i\left(\sum_{j\in J_i} w_{ij} f_j(x_{K_j}) + a_i^T x - b_i\right) \in [v_i, w_i] \\ &\qquad\qquad (i \notin I_0). \end{aligned} \tag{4.1}$$

The g_i are called group functions, and the arguments of the g_i are the groups. The w_{ij} are weights, the f_j nonlinear element functions depending on subvectors x_{K_j} of x, and the $a_i^T x - b_i$ are referred to as linear element functions.

Converting to NOP-2 format can be done as follows. We introduce extra variables by adding the constraints

$$y_j = f_j(x_{K_j}), \tag{4.2}$$

$$z_i = \sum_{j\in J_i} w_{ij} y_j + a_i^T x - b_i \tag{4.3}$$

if $w_{ij} \neq 1$ or $a_i \neq 0$, and

$$z_i = \sum_{j\in J_i} f_j(x_{K_j}) - b_i \tag{4.4}$$

otherwise. In this way the objective function simplifies to a simple element

$$f = \sum_{i\in I_0} g_i(z_i) \tag{4.5}$$

(or to a sum of several elements if the g_i are different), and the constraints become elements with a single piece only,

$$g_i(z_i) \in [v_i, w_i]. \tag{4.6}$$

No example is included here, since the specification of Rosenbrock's function already needs more than one page in SIF.

4.2. Conversion from AMPL

Here we want to demonstrate that our NOP-2 modeling language has capabilities comparable to AMPL. Consider the following optimization problem which is taken from the AMPL book [6]. Models like this frequently arise in the analysis of economic processes.

Given:	P,	a set of products
	a_j	= tons per hour of product j, for each $j \in P$
	b	= hours available at the mill
	c_j	= profit per ton of product j for each $j \in P$
	u_j	= maximum tons of product j for each $j \in P$
Define variables:	X_j	= tons of product j to be made, for each $j \in P$
Maximize:	$\sum_{j \in P} c_j X_j$	
Subject to:	$\sum_{j \in P} (1/a_j) X_j \leq b$	
	$0 \leq X_j \leq u_j$, for each $j \in P$	

The AMPL version of these algebraic expressions is:

```
set P;
param a {j in P};
param b:
param c {j in P};
param u {j in P};
var X {j in P};

maximize profit:
    sum {j in P} c[j] * X[j];
subject to Time:
    sum {j in } (1/a[j] * X[j] <= b;
subject to Limit {j in P}:
    0 <= X[j] <= u[j];
```

with the AMPL data file `prod.dat`.

```
set P := bands coils;
param: b := 40;
param:    a    c    u    :=
  bands  200  25  6000
  coils  140  30  4000 ;
```

The comparison with the problem coded in NOP-2 shown below and the above listing clearly shows usefulness of our new modeling language.

```
// data templates
param int n;
param real a[n] b c[n] u[n];
// get the 'real' data
incl "data.nop"

// vector of variables
real profit x[n];

// what to do
max profit;
// problem bounds
bnd x in [0,u];
// constraints
lin x; c = profit;
lin x; 1/a <= b;
```

and the equivalent data file

```
//          bands    coils
const a = ( 200       140)
      c = (  25        30)
      u = (6000      4000);
const b = 40;
```

The conversion is not straightforward since it involves the detection of structure and common element functions. However, a tool is being prepared that does the conversion in many cases automatically and otherwise gives help during the conversion process.

4.3. Conversion from GAMS

The NOP-2 modeling language also has capabilities comparable to GAMS. The following optimization problem is taken from the GAMS home page. As in the case of AMPL, the model could arise in the analysis of an economic process.

```
SETS
   I canning plants
       / SEATTLE, SAN-DIEGO /
   J markets
       / NEW-YORK, CHICAGO, TOPEKA / ;
PARAMETERS
   A(I)  capacity of plant i in cases
     /    SEATTLE     350
          SAN-DIEGO   600  /
   B(J)  demand at market j in cases
     /    NEW-YORK    325
          CHICAGO     300
          TOPEKA      275  / ;
TABLE D(I,J)  distance in thousands of miles
              NEW-YORK   CHICAGO   TOPEKA
   SEATTLE     2.5        1.7       1.8
   SAN-DIEGO   2.5        1.8       1.4  ;
SCALAR F  freight in dollars per case
```

```
          per thousand miles  /90/ ;
PARAMETER C(I,J)  transport cost in thousands
                  of dollars per case ;
          C(I,J) = F * D(I,J) / 1000 ;
VARIABLES
    X(I,J)  shipment quantities in cases
    Z       total transportation costs
            in thousands of dollars ;
POSITIVE VARIABLE X ;
EQUATIONS
    COST      define objective function
    SUPPLY(I) observe supply limit at plant i
    DEMAND(J) satisfy demand at market j ;
COST ..      Z =E= SUM((I,J), C(I,J)*X(I,J)) ;
SUPPLY(I) .. SUM(J, X(I,J)) =L= A(I) ;
DEMAND(J) .. SUM(I, X(I,J)) =G= B(J) ;
MODEL TRANSPORT /ALL/ ;
SOLVE TRANSPORT USING LP MINIMIZING Z ;
```

The GAMS model can be translated to NOP-2 resulting in a model of about the same size:

```
// data templates
param int n m;
param real A[n] B[m] C[m,n];
// get the 'real' data
incl "data.nop"

// vector of variables
real X[n,m] // shipment quantities
     Z      // total transportation costs
            // in thousands of dollars;

// what to do
min Z;
// problem bounds
bnd X[:,:] >= 0;
// constraints
lin X[:,:]; C[:,:] = Z;
loop(i=1:n)
  sum X[i,:] <= A[i]
endloop
loop(j=1:m)
  sum X[:,j] >= B[j]
endloop
```

and the equivalent data file

```
const n=2 m=3;
//         seattle   san diego
// capacity of the plants
const A = ( 350       600)
//     new york  chicago  topeka
// capacity of the markets
  B = ( 325       300       275);
//   table of distances in thousand miles
const D = dense(3,2)
//new york  chicago  topeka
     2.5     1.7      1.8   // seattle
     2.5     1.8      1.4;  // san diego
// freight in dollars per thousand miles
const f = 90;
// transport cost in thousands of dollars
const C = f*D/1000;
```

5. Condensed language specification

This section only contains a basic description of the language and some examples. A more detailed description is, e.g., contained in [14].

All records start with a keyword that defines the interpretation of records. Records may be continued to the following lines without continuation character (in contrast to NOP), and end after a semicolon (`;`) before the next keyword. There is no limit on the maximum length of a record.

Moreover, text after `//` up to the end of the line or between `/*` and `*/` is ignored (C++ style comments). Blanks, tabs and newlines are ignored except that they separate tokens. They can be left out whenever the expression is unambiguous. Multiple blanks are treated as a single blank.

All keywords and names are case sensitive.

In the following description of records and examples, keywords are printed in bold face.

There are the following kinds of records:

Problem declaration. It must appear exactly once in every NOP-2 file and defines the type of the optimization problem (find local/global minimizers, local/global maximizers, or some/all feasible points). The related keywords are **min**, **lmin**, **max**, **lmax**, **some**, and **all**.
Examples:

valid: `min objective;`
valid: `lmax y[25+3*5];`
valid: `some;`

valid: `all;`
invalid: `min;`
This is invalid since for minimization (maximization) an objective variable must be specified.
invalid: `all x[29];`
For finding all feasible points an objective variable must not be specified.

Iterated declarations. They are used to iterate certain declarations to allow for scalable problem specifications. The specified iterator must evaluate to a finite set of constants. For every constant in this list all declarations in the list of declarations are evaluated, hereby replacing all occurrences of the iterator variable identifier with the value of the constant. The related keywords are **loop** and **endloop**. Iterators may, of course, be nested.
Examples:

valid:

```
real a[n];
loop(i=1:n)
  const a[i] = i*(i+1)/2;
endloop
```

valid:

```
real Hilbert[n,m];
loop(i=1:n)
  loop(j=1:m)
    const Hilbert[i,j] = 1/(i+j-1);
  endloop
endloop
```

invalid (because of zero increment):

```
const a=0;
loop(i=1:0:2)
  const a+=i;
endloop
```

Selected declarations. An if–statement is used to select different declarations depending on others. They consist of one **if** part, at most one **else** part, and in between an arbitrary number of **elseif** parts. The declaration is valid only if all boolean expressions in the selectors can be evaluated at compile-time. The selected declarations should not be confused with the `if1`, `if2`, and `if3` standard element functions which enable branching in mathematical expressions.
Examples:

valid:

```
param real n;
...
if(n<=0)
  error "n must be greater than 0";
else
  a[n] = 1**n;
endif
```

valid:

```
real A[n,n];
loop(i=1:n)
  loop(j=1:m)
    if(i==j)
      const A[i,j] = 1;
    else
      const A[i,j] = 1/abs(i-j);
    endif
  endloop
endloop
```

invalid:

```
if(i<=5)
  const a=i;
else
  const a=5;
else
  error "i must be less or equal to 5";
endif
```

Constant declarations. They are used to assign constants that can be used in every expression or record following them. They start with the keyword **const** and consist of a list of constant assignments. Assignment is possible for numbers, vectors, matrices, higher order tensors, and strings. Numbers are entered in a way similar to FORTRAN, C, or C++. There are additional qualifiers **!** for numbers specifying constants that have to be represented exactly (and if this is not possible, are represented by the smallest enclosing intervals, with a warning issued), and **?** for inaccurate

numbers known to be precise up to half a unit in the last place. (This allows correct input of problems that need to be solved with mathematical guarantee, by global optimization routines such as INTOPT_90 [8] that use directed rounding to take roundoff errors into account.) In addition, the keyword **infty** is used to specify ∞ as special "number".
Examples:

valid: `const a=7.347e-12? l=3.985543 b=-2*l!;`
valid: `const msg="Stop Mad John!";`
valid: `const A[-1:2:7] = (-2 5.27 1.348 e^(-2) 0);`
valid: `const A[:,5:8] = B[:,1 2 4 7];`
invalid: `const a b=3;`
There has to be an assignment for every constant defined.
invalid: `const a(1:4) = (-2 9 4 3);`
Vectors and tensors are specified using square brackets as in C and in contrast to FORTRAN.
invalid: `const a[1:4] +-*/.= (-2 9 4 3);`
`+-*/.=` is not a valid operator.

The standard library contains some predefined constants, such as `pi`, `e`, and `i`.

Specification of vectors and matrices Vectors are usually specified as a list of constants enclosed in parenthesis. There is an alternative way for specifying vectors where almost all entries are equal to the same constant.
Examples:

valid: `const v = (0.2 4*3.7 2*pi e^(-1/2));`
defines a vector of length 4 with the given components,
valid: `const w = (29$1 2.7@3 9 11; 4.9@12 23);`
defines a vector w of dimension 29; $w_i = 2.7$ if $i = 3, 9, 11$, $w_i = 4.9$ if $i = 12, 23$, and $w_i = 1$ otherwise.

Matrix definitions are more complicated, as there are various ways for specifying them, including special formats for expressing sparsity. Special matrix types include **dense**, **band**, **diag** (diagonal), **tria** (triangular), and **sparse**.They can be entered **row**–wise or **col**umn–wise. All matrices can be specified to be **sym**metric or **antisym**metric.
Examples:

valid `const A = dense[sym,up](3) 1.1 1.2 2.2 1.3 2.3 3.3;`
This specifies a symmetric dense matrix by listing the elements of the upper triangular part (column-wise, which is the default).
valid `const B = sparse[sym,up](4) 1.2@(1 2), 3.4@(3 4);`
This specifies a sparse symmetric matrix, whose entries are $a_{12} = a_{21} = 1.2$, $a_{34} = a_{43} = 3.4$, and $a_{ij} = 0$ otherwise. There are three other methods for specifying sparse matrices, which closely resemble MATLAB output, and schemes using index vectors, respectively.

Higher order tensors can be specified in a similar way using the keywords **tdense** and **tsparse**.

Variable declarations. Variables must be declared prior to their first use. (However, the standard library contains a set of predefined variables: a real vector `x`, a complex vector `z`, and an integer vector `j`.)

A variable declaration is a record starting with an optional qualifier keyword: one of **free** for free variables (such appear only in NOP-2 files that specify optimization problems, that appear as subproblems of other optimization problems, called with a **call** statement), or **param**, for the declaration of parameters, that should be specified later.

After the initial qualifier there may appear an optional **stoch** for stochastic variables, and thereafter a *type* is specified, which is one of **complex**, **real**, **int**, having their obvious meaning, **bin**, for binary variables that assume only the values 0 or 1, and **string** for character strings.

Immediately following the type is a list of variables (possibly with multidimensional array ranges for vectors, matrices or higher order tensors).
Examples:

valid: `param int i;`
valid: `stoch bin b[i];`
The array size may depend on previously defined constants or parameters.
valid: `free stoch real xi[7] eta;`
valid: `real A[7,9:12];`
valid: `free string name;`
invalid: `free param real x;`
There must not be two qualifiers
invalid: `param x[9];`
There are no implicit types. Every type must be explicitly listed.

invalid: `real v[1:2:9];`
Unlike constants, arrays have to be contiguous, the range including all numbers between the start index and the end index. If the start index is omitted, it is assumed to be one.

Bound declarations. These declarations are used to reduce the possible range of a set of variables. They start with the keyword **bnd**.
Examples:

valid: `bnd x[1:9] in [-1,3];`
valid: `bnd y x[4] A[:,:]  <= 0;`
valid: `bnd j in {1,4,9,16,25};`
Finite sets are possible; unions of sets can be specified using the operator `|`, and intersections using `&`.
valid: `bnd x < 4;`
`<` and `>` are valid range restrictors. However, they retain their meaning only for integer variables. For the others, `<` and `>` are interpreted as `<=`, and `>=` respectively.
valid: `bnd A is(psd)`
For Matrices special matrix properties may be specified, like being positive semi-definite.
invalid: `bnd v[1:9:12];`
no bound specified
invalid: `bnd 89 <= x[4];`
variables must appear on the left side

Hints. If modelers wants to give hints about possible good starting points for local optimization, etc., they can do so by using **hint** records. They start with the keyword **hint** followed by a list of variables, an equality sign, and a list of constants.

The number of hints is not limited, and it is possible to specify an incomplete list of variables. This information is passed to the solver unchanged and not-specified variables are specially marked.
Examples:

valid: `hint x[1:9] = 1**9;`
tells the parser that $x_{1\ldots9} = (1, \ldots, 1)$ should be a good starting point.
valid: `hint x[1 4 6] y[1:19] obj = 1**3 2**19 0.765;`
some variables may remain unspecified (e.g. `x[2]`)
invalid: `hint 4 3 8;`
no variables specified.
invalid: `hint 42 = x[4];`
variables must appear on the left side.

Distribution declarations. In order to specify the distribution of stochastic variables, a distribution declaration is required. It starts with **distr** followed by the variables, the name of the distribution function, and a comma-separated list of constant expressions, defining the parameters.
Examples:

valid: `distr xi ~ N(0,1);` This specifies a Gaussian random variable with zero mean and variance 1.
valid: `distr eta[1:9] ~ covar(A);` This specifies a random vector with zero mean and covariance matrix A (nothing else known).
invalid: `distr foo ~ 0.24;`
There is no name for a distribution function present.

Stage declarations. Such a declaration is used to specify in which stage of a multistage stochastic model the variable exists. Its syntactic form is **stage** followed by a list of variables, the `=` sign and an integer.

Variables with undeclared stages are taken to belong to stage 1.
Examples:

valid: `stage xi[1:9] = 2;`
valid: `stage xi eta[4] A[:,:]  = 1;`
invalid: `stage xi < 4;`
No equal sign.
invalid: `stage eta = 3.57*pi;`
The number must be an integer.

Function declarations. A number of elements that occur frequently in models, listed in [12], are predefined in the standard library. These need not (and must not) be declared. (Note that for binary variables, the `min0` and `max0` elements code for `and` and `or`.)

Other functions can be defined by the user, using a notation similar to higher level programming languages, or via a black box interface. The function definition is enclosed in **func**–**endfunc** blocks. Some examples are given below and in the list of examples of Section 3.

A function's expression can consist of the single keyword **blackbox** that tells the parser – and later the automatic evaluation function generator – that all necessary work will be done by external subroutines supplied by the model builder. This feature is provided to allow for widest flexibility. The information to be provided

by the user in such an external subroutine will usually include function values and gradients; for branch and bound applications also reasonable outer enclosures of the range, Lipschitz constants, quadratic underestimations, and similar global information may be needed. For the user-defined element functions, this kind of information is produced automatically with a quality depending on the amount of dependence and nonlinearity present in the definition; for predefined element functions, an attempt was made to produce global information of highest quality.

Examples for function definitions:

valid:

```
func stability(real al be ga M[4,4];
               real p q)
  blackbox;
endfunc
```

valid:

```
func xlog(real y[#])
  apply(+,i=1:#)
    y[i]*log(y[i]);
  endapply
endfunc
```

The character `#` tells the parser that `y` is a vector, whose dimension is to be compiled at parse time.

valid:

```
func min1(real y[#])
  apply(min,i=1:#)
    abs(y[i]);
  endapply
endfunc
```

valid:

```
func rmrowcol(real M[#[1],#[2]]; int i j;
              real RM[#[1]-1,#[2]-1])
  if(i<=0 || i>#[1])
    error("Row index i (".i.") is ".
          "greater than number of rows".
          #[1]);
  endif
  if(j<=0 || j>#[2])
    error("Column index j (".j.") is ".
          "greater than number of columns".
          #[2]);
  endif
  RM = M[1:i-1 i+1:#[1], 1:j-1 j+1:#[2]];
endfunc
```

invalid:

```
func(real a[#[1]] b[#[2]]; int i j k)
  if(i-j+k>27)
    a[9] = 7+b[i]-j*k;
  endapply
endfunc
```

invalid for two reasons: **if** is ended by **endapply**, and `a[#[1]]` and `b[#[2]]` specify two arrays of variable size. It is impossible for the parser, however to detect where the first array is supposed to end, and where the second array is supposed to start.

invalid:

```
func(real y[10])
  apply(sin,i=1:10)
    pi*y[i];
  endapply
endfunc
```

Only `+`, `*`, `+-`, `min`, and `max` are valid within `apply`.

Element declarations. Element declarations are used to define the constraints of the model. They start with an element name, followed by a list of variables and an optional list of parameters, separated from the variables by a semicolon `;`. Then comes a bound-specifier, which is similar to the bound definition for variables, or an `=` sign and a simple expression involving other variables.

Examples:

valid: `sum x[1 2 4] y[2 7] = x[22];`
$x_1 + x_2 + x_4 + y_2 + y_7 = x_{22}$.

valid: `lin x[1:4]; -1 -3.8 3+4*pi e^(-2) <= 27;`

valid: `mv y A = w;`

valid: `uhu ; -2 3 -27 = 7.9*al[2] + 4;`
An element not containing any variables at all will be checked at compile-time, and then discarded.

invalid: `dont_try[-4,9] <= x[42];`
This record does not contain an element name.

invalid: `old_format 1 3 4 6; -1.2 1.e17 9 x15;`

The old format for entering NOP expressions is no longer valid.

Call declaration. For stochastic multistage problems, minimax problems, or multilevel programs, it is possible to specify optimization subproblems in a separate NOP-2 file. To make the subproblem accessible by the main specification, the call declaration is provided. It looks as follows

call *file-name* (constant expressions)**;**

where the constant expressions are substituted at call time for the `free` variables in the sub-specification.
Examples:

valid: `call "subprogram.nop"(42,x[42],y[42], "don't panic");`
valid: `call "try_this.nop"(A,v[0], "segmentation fault");`
invalid: `call missing_quotes(3.247*pi*y[1:39]);`
invalid: `call no_parenthesis x y;`

Expressions. Expressions closely resemble mathematical notation used in programming languages. The following operators are provided. Their semantic meaning is defined below depending of the type of their operands. Infix operators are:

`||`	Boolean or
`^^`	Boolean exclusive or
`&&`	Boolean and
in	whether the left hand side is included in the set on the right hand side.
==	Boolean valued equality
!=	Boolean valued inequality
>	greater than relation for real and integer matrices, vectors, sets, and scalars. Matrices and vectors are compared componentwise, and the relation is true, if all components of the lhs are greater than the corresponding ones of the rhs. The set on the lhs is greater than the one on the rhs, if it is a true superset.
<	less than relation analogous to >.
>=	greater or equal relation analogous to >.
<=	less or equal relation analogous to >.
<=>	three-way comparison. Evaluates to 1 if the left hand side is greater than the right hand side, to 0 if the left hand side equals the right hand side, and to -1 otherwise.
+	addition of scalars, vectors, matrices, and tensors
–	subtraction of scalars, vectors, matrices, and tensors
|	union of two sets
&	intersection of two sets
\	set theoretic difference of two sets
*	multiplication of scalars, componentwise multiplication of vectors, matrices, and tensors, multiplication of scalars with vectors, matrices, and tensors, and Cartesian product of sets.
.	matrix multiplication, considering vectors as matrices with either 1 row or 1 column, as appropriate.
(*)	tensor product of matrices, tensors, and vectors, yielding tensors of higher order
/	division (of scalars, vectors, matrices, and tensors) by scalars.
%	modulo division of integer scalars, vectors, matrices, and tensors by integer scalars.
^	taking powers of scalars, sets, and matrices. The right hand side must be an integer in the case of sets and matrices.

Prefix operators:

- − unary minus of scalars, vectors, matrices, and tensors
- + has no effect as prefix operator (unary plus).
- **!** Boolean not

Postfix operators:

- ´ transposition of a matirx.

6. Application

We have written an ANSI-C library that provides functions for parsing a NOP-2 file and translating it into a binary format suitable for interpretation within other programs. All variables are incorporated into a single vector, and all index ranges, constants, repetitions are expanded. All constant expressions that can be evaluated at parse time are evaluated. The complete problem information is stored in a hierarchical tree of C structures.

This library is used in GLOPT-2, a global optimization package written in ANSI-C that is currently under development. GLOPT-2 uses a branch and bound technique to split the problem recursively into subproblems that are either eliminated or reduced in their size. This is done by an extensive use of the block separable structure of the optimization problem, as displayed in the NOP-2 format. Apart from the techniques already implemented in the precursor GLOPT [4], it includes many enhancements in the reduction techniques.

The latest news can always be found on the global optimization web pages mentioned at the beginning of Section 3.

Acknowledgment

The major part of this work was supported by the European External Research Program (Contract No. AU-043) by DIGITAL Equipment Corporation, Austria, and by the Austrian Fond zur Förderung der wissenschaftlichen Forschung (FWF) under grant P11516-MAT. Some part was done during visits of the authors at Bell Laboratories (Lucent Technologies, formerly AT&T), Murray Hill, NJ, USA.

In addition, we want to express our special thanks to Prof. David Gay (Lucent Technologies) for various fruitful comments.

References

[1] A. Brooke, D. Kendrick, A. Meeraus, *GAMS – A User Guide*, The Scientific Press, Redwood City, CA, 1988

[2] J. Castillo and I.E. Grossmann, *Computation of phase and chemical equilibria*, Computers Chem. Engin. **5** (1981), 99–108.

[3] A.R. Conn, N.I.M. Gould, and Ph. L. Toint, *LANCELOT. A Fortran Package for large-scale nonlinear optimization*, Springer, Berlin 1992.

[4] S. Dallwig, A. Neumaier, H. Schichl, *GLOPT - a program for constrained global optimization*, in: I. Bomze et al., eds., Developments in Global Optimization (Proc. 3rd Workshop Global Optimization, Szeged 1995).

[5] C. A. Floudas and P.M. Pardalos, *A collection of test problems for constrained global optimization algorithms*, Lecture Notes Comp. Sci. 455, Springer, Berlin 1990.

[6] R. Fourer, D.M. Gay, B.W. Kernighan, *AMPL. A modeling language for mathematical programming*, Scientific Press, San Francisco 1993.

[7] C. Jansson and O. Knüppel, *A global minimization method: The multi-dimensional case*, Technical Report 92.1, Hamburg-Harburg 1992. (electronic copy at http://www.ti3.tu-harburg.de/Software.html)

[8] R.B. Kearfott, *Rigorous Global Search: Continuous Problems*, Kluwer, Dordrecht 1996

[9] J.R. Kowalik, M.R. Osborne, *Methods for Unconstrained Minimization*, American Elsevier, New York 1968

[10] A. Mfayokurera, *Nonconvex phase equilibria computations by global minimization*, M. Sc. Thesis, Dept. of Chem. Engineering, Univ. of Wisconsin, Madison, Wisconsin, 1989.

[11] B. A. Murtagh and M.A. Saunders, *MINOS 5.1 user's guide*, Tech. Report SOL 83-20R, Stanford Univ., Stanford, Calif. 1983, revised 1987.

[12] A. Neumaier, *NOP - a compact input format for nonlinear optimization problems*, in: I. Bomze et al., eds., Developments in Global Optimization (Proc. 3rd Workshop Global Optimization, Szeged 1995).

[13] H. Renon and J.M. Prausnitz, *Local compositions in thermodynamic excess functions for liquid mixtures*, AIChE **14** (1968), 135–144.

[14] H. Schichl, S. Dallwig, A. Neumaier, *NOP-2 modeling language*, Technical Report to Digital Equipment corporation for the European External Research Program (Contract No. AU-043) (1997)

[15] G. Walster, E. Hansen and S. Sengupta, *Test results for a global optimization algorithm*, pp.272–287 in: Numerical Optimization 1984 (P.T. Boggs et al., eds), SIAM, Philadelphia 1985.