Chapter 15

THE NOP-2 MODELING LANGUAGE

Hermann Schichl*
Institut für Mathematik der Universität Wien
Strudlhofgasse 4, A-1090 Wien, Austria
Hermann.Schichl@esi.ac.at

Arnold Neumaier
Institut für Mathematik der Universität Wien
Strudlhofgasse 4, A-1090 Wien, Austria
Arnold.Neumaier@univie.ac.at
http://www.mat.univie.ac.at/∼neum

Abstract We present a short overview over the modeling language `NOP-2` for specifying general optimization problems, including constrained local or global nonlinear programs and constrained single and multistage stochastic programs. The proposed language is specifically designed to represent the internal (separable and repetitive) structure of the problem.

Keywords: Modeling Language, Global Optimization, NOP-2

15.1 Introduction

Solving global optimization problems effectively in a rigorous way requires a lot of analytical knowledge about the functions involved. The speed of a branch-and-bound approach is, *e.g.*, directly correlated to the amount of overestimation produced by interval enclosures of linear, quadratic, or convex relaxations.

*supported by EU project COCONUT IST-2000-26063

The origin of the NOP-2 modeling language does not lie in the development of modeling languages but in global optimization. The NOP input format was constructed in 1997 by Arnold Neumaier mainly as an interface to the global optimization package GLOPT [165, 43], a global solver for nonlinear partial separable global NLPs. NOP provided a good means for entering specifications in a compact line oriented syntax; it had the possibility to mix element functions and mathematical notation but it did not have the strength of a true modeling language, it did not provide named variables, extensibility of element functions, or matrices. The first parser was written in FORTRAN 77, as was the GLOPT solver.

In the following year 1998, GLOPT turned out to be too inflexible, so the system was redesigned, and for the implementation it was decided to switch from FORTRAN to C. Since a new input interface was required anyway, NOP-2 was designed as input language for GLOPT-2, and it would be a convenient modeling language, at least for mathematicians. The new parser was written in Bos (an enhanced version of Ox), a kind of preprocessor language for Lex and Yacc.

At the time NOP-2 was developed, its element concept (see Section 15.3.1) seemed to be one of the most promising approaches to global optimization, but nowadays more flexible problem representations have been developed (*e.g.* directed acyclic graphs [188]).

The development of GLOPT-2 has been stopped in the meantime, and the authors work in the COCONUT [40] project towards a new solver platform for global optimization. This new platform uses AMPL [72] (see Chapter 7) as its primary input format, but a GAMS [28] (see Chapter 8) interface will be available as well.

These facts have greatly reduced the need for the NOP-2 language, and so its *development is stopped*, and it is *not maintained* any longer.

However, the language still contains a few unique concepts, and we hope that the providers of the commercial modeling languages can be persuaded to include some, if not all, of them into their systems.

15.2 Concepts

NOP-2 was primarily designed for modeling *global optimization* problems, and to be a general purpose modeling language which contains the possibility to split a problem specification into elementary functions, *explicitly display* its *block separable structure* and which has similar flexibility and extensibility as the other modeling languages presented in this volume (see Chapter 4).

Most solvers which are connected to modeling systems like AMPL or GAMS are local optimization codes who do not need analytical knowledge for a fast solving process. Extracting analytical information needed for global optimization (*e.g.*,

range enclosures with minimal overestimation, almost optimal linear enclosures of functions, convexity areas, etc.) from flat models which are typical for algebraic modeling languages cannot easily be done without human intervention or an enormous amount of computer algebra. Most rigorous global optimization packages, however, make use of this kind of information.

In NOP-2 a model has to be specified in a highly structured way; the emphasis is on the block-separability of the model structure. The element function approach, the extensibility of NOP-2, and a library of element functions make it possible to specify optimization problems in such a way that global optimization packages can be provided with as much analytical information as possible, see 15.3.1. However, the modeler needs a decent knowledge in mathematics and a will to twist the model, performing mathematical transformations, until it nicely fits with the provided elements.

Another important design parameter was the *minimality of modeling overhead*. The modeling language SIF [41] (the input language of Lancelot, and a nonlinear extension of MPS) was a negative example.

Furthermore, *stochastic programs and multistage models* should be representable, and the language should be easily expandable. Finally, as outlined in Chapter 4, Section 4.2, for global optimization and verified computing it is extremely important that all levels of rigor for data are representable.

Still, the main interest in developing NOP-2 was to provide a convenient input format for our solver. The design was an extension of the original NOP idea and was not developed as a new modeling language, and the wuthors were not specialists in modeling language design. Therefore, some features generally available in algebraic modeling languages are missing from NOP-2.

We did *not* care about *set indexing*, so variables can only be *indexed by integers* not by sets. This does not reduce the number of models which can be specified, but it is a major difference on how models are developed.

We did *not* think about *connecting* many *different solvers* to NOP-2, since it was primarily intended as an input language to GLOPT-2. There was no API written for solver connection. In addition, we did not care about passing data back from the solver to the modeling system; we could control the output of the solver anyway.

There was *no clear separation between models and data*. Data had to be specified in NOP-2 syntax, and the input statement was the only chance to separate data and models somewhat. No elaborate data reading was implemented, no database connections, no spreadsheet interface, etc.

Finally, there is (almost) no automatic generation of derived data, especially *no automatic differentiation*. Since the amount of analytical information in global optimization is huge, we kept the generation of this information, hence the automatic differentiation, in the solver.

By the way, there is absolutely no procedural part (not even a `solve` statement) in `NOP-2`. In some sense, it is a very pure declarative language.

15.3 Specialties of NOP-2

This section is devoted to presenting special features unique to `NOP-2`. Instead of giving details on how models in general are designed in our language, modeling the Rosenbrock [114] function

$$\begin{array}{ll} \min & 100(x_1^2 - x_2)^2 + (x_1 - 1)^2 \\ \text{s.t.} & x_1, x_2 \in [-2, 8]. \end{array} \tag{15.3.1}$$

shall serve as a illustrative example.

In (15.3.1), the objective function is a polynomial in two variables of degree 4. For multivariate higher degree polynomials, direct calculation of good interval range enclosures and quadratic relaxations are in general a non-trivial task. We can improve on that in this case, making the least squares structure apparent by introducing the variable

$$x_3 = 10x_1^2 - 10x_2, \tag{15.3.2}$$

thereby reducing the objective function to

$$x_4 = x_3^2 + (x_1 - 1)^2. \tag{15.3.3}$$

This shows that we can rewrite the optimization problem equivalently as

$$\begin{array}{ll} \min & x_4 \\ \text{s.t.} & x_4 = x_3^2 + (x_1 - 1)^2 \\ & x_3 = 10x_1^2 - 10x_2 \\ & x_1, x_2 \in [-2, 8]. \end{array} \tag{15.3.4}$$

In this form, we have increased the dimension by one (adding x_3) but we have reduced the objective function and constraints to quadratic functions. Quadratic relaxation and good interval enclosures have become easy; we can even use inverse functions for improving on the solution process.

If we now count the total number of variables and 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];
```

Here the line `min x[4]` tells the solver that the variable x_4 should be minimized, the line starting with `bnd` fixes the bounds on the variables x_1 and x_2, and the remaining lines give the constraints.

`qu2` and `qu4` are so called element functions (see 15.3.1). Many of them are predefined and specially selected such that most of the analytic information and estimates can be computed efficiently for them. Although it is possible to define new element functions, a modeler is advised to perform mathematical transformations until the model consists of predefined element functions only. This new structure will enable the global optimization code to use most of the built-in analytical knowledge.

15.3.1 Specifying Structure — The Element Concept

The most special element in `NOP-2` is the very explicit way of specifying the structure of the problem. Every model must be decomposed into elementary functions. These *element functions* are then entered line by line.

In general, `NOP-2` allows to describe an optimization problem of one of three basic forms

1 A classical global optimization problem

$$\begin{array}{ll} \min & \omega x_s \\ \text{s.t.} & E_\nu(x), \quad \nu = 1, \ldots, N \\ & x \in [x_0], \end{array} \tag{15.3.5}$$

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

2 A single stage stochastic optimization problem

$$\begin{array}{ll} \min & f(x, \xi) \\ \text{s.t.} & E_\nu(x, \xi), \quad \nu = 1, \ldots, N, \\ & x \in [x_0], \\ & \xi \sim g(b, x), \end{array} \tag{15.3.6}$$

as in 1. possibly with additional integrality constraints. The variables ξ are stochastic variables with (probably unknown or partially unknown) distribution functions $g(b, x)$.

3 Most general, a multistage optimization problems of the form

$$\begin{array}{ll} \min & f(x^{(k)},\xi)^{(k)} \\ \text{s.t.} & E_\nu^{(k)}(\xi^{(k)},\xi^{(<k)},x^{(<k)}), \quad \nu=1,\ldots,N^{(k)}, \\ & x^{(k)} \in [x_0^{(k)}], \\ & \xi^{(k)} \sim g^{(k)}(b^{(k)},x), \end{array} \tag{15.3.7}$$

as in 1. and 2. possibly with additional integrality constraints. The variables $\xi^{(k)}$ are stochastic variables with (probably unknown or partially unknown) distribution functions $g^{(k)}(b^{(k)},x)$, and are valid in stage k of this multistage problem.

The so-called *elements* $E_\nu(x)$ are expressions of one of the forms

$$\bigodot_k f(a_k, x_{J_k}) \in b\mathbf{q} + c, \tag{15.3.8}$$

$$\bigodot_k f(a_k, x_{J_k}) = bx_{K_j} + c, \tag{15.3.9}$$

$$\bigodot_k f(a_k, x_{J_k}) \in S, \tag{15.3.10}$$

and a few irregular variants, that consist 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, 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, $\mathbf{q}$ is a possibly unbounded box, possibly 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 useful in coding a large number of problems are listed in Tables 15.1 and Table 15.2, and are among others collected in a standard library, that is by default included into every `NOP-2` specification. 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. Other element functions can easily be defined using algebraic statements in a syntax similar to most algebraic modeling languages.

name	element function	element shape
`sum`	x	$\sum x_i$
`lin`	px	$\sum p_i x_i$
`abs`	$\|x\|$	$\sum \|x_i\|$
`abs2`	$p\|x\|$	$\sum p_i \|x_i\|$
`sqr`	x^2	$\sum x_i^2$
`qu1`	px^2	$\sum p_i x_i^2$
`qu2`	$(x-p)^2$	$\sum (x_i - p_i)^2$
`qu3`	$p_2(x-p_1)^2$	$\sum p_{i+m}(x_i - p_i)^2$
`qu4`	$p_1 x + p_2 x^2$	$\sum (p_i x_i + p_{i+m} x_i^2)$
`pow`	x^p	$\sum x_i^p$
`log`	$\log \|x-p\|$	$\sum \log \|x_i - p_i\|$
`xlog`	$x \log x$	$\sum x_i \log x_i$
`exp`	$p_1 e^{-p_2 x}$	$\sum p_i e^{-p_{i+m} x_i}$
`gss`	$p_3 e^{-\frac{p_2}{2}(x-p_1)^2}$	$p_{i+2m} e^{-\frac{p_{i+m}}{2}(x-p_i)^2}$
`atan`	$\arctan(x)$	$\sum \arctan(x_i)$
`pr0`	$x_1 x_2$	$\sum x_{2i-1} x_{2i}$
`pr1`	$x_1 x_2$	$\sum x_i x_{i+m}$
`pr2`	$(x_1 - p_1)(x_2 - p_2)$	$\sum (x_i - p_i)(x_{i+m} - p_{i+m})$
`div`	x_1 / x_2	$\sum x_{2i-1} / x_{2i}$
`bil`	$x_1 x_2 + p_1 x_1 + p_2 x_2$	$\sum (x_i x_{i+m} + p_i x_i + p_{i+m} x_{i+m})$
`qf1`	$p x_1 x_2$	$\sum p_i x_i x_{i+m}$
`qf2`	$p_1 x_1^2 + p_2 x_1 x_2 + p_3 x_2^2$	$\sum (p_i x_i^2 + p_{i+m} x_i x_{i+m} + p_{i+2m} x_{i+m}^2)$
`dsq`	$(x_1 - x_2)^2$	$\sum (x_i - x_{i+m})^2$
`atan2`	$\pm\arctan(x_1/x_2)$ (sign as in C, FORTRAN)	$\sum \pm\arctan(x_{2i-1}/x_{2i})$
`pol`		$\sum_{i=1}^{n} p_i {x_j}^{n+1-i}$
`sin`		$\sum_{i=1}^{n} a_i \sin(i\omega x_j - p_i)$
`cos`		$\sum_{i=1}^{n} a_i \cos(i\omega x_j - p_i)$
`if1`		if $x_i \le p$ then x_j else x_k
`if2`		if $x_i \ge p$ then x_j else x_k
`if3`		$\begin{cases} x_j & \text{if } x_i \le p, \\ x_k & \text{if } p < x_i \le q, \\ x_l & \text{if } x_i > q \end{cases}$
`prod`	x	$\prod_{i=1}^{n} x_{j_i}$
`min0`	x	$\min_{i=1}^{n} x_{j_i}$
`min1`	$\|x\|$	$\max_{i=1}^{n} \|x_{j_i}\|$
`min2`	$p\|x\|$	$\max_{i=1}^{n} p_i \|x_{j_i}\|$
`max0`	x	$\max_{i=1}^{n} x_{j_i}$
`max1`	$\|x\|$	$\max_{i=1}^{n} \|x_{j_i}\|$
`max2`	$p\|x\|$	$\max_{i=1}^{n} p_i \|x_{j_i}\|$

Table 15.1. Predefined elements with one dimensional result

name	element shape	short description
`mv`	Av	left multiplication of vector with matrix
`vm`	$v^T A$	right multiplication of vector with matrix
`vmv`	$v^T Av$	evaluation of the quadratic for specified by A
`lookup`		interpolate in a specified way a function given by a table
`trace`	tr(A)	the trace of the matrix A
`det`	det(A)	the determinant of the matrix A
`cond`	cond(A)	the condition number of the matrix A

Table 15.2. Elements having higher dimensional results or involving matrices and tensors

15.3.2 Data and Numbers

Data for global optimization is a tricky thing. We have seen in Chapter 1, Section 1.2.7 and in Chapter 4, Section 4.2 that various levels of rigor are interesting when data has to be specified. There are several possibilities, and the modeling language has to act according to the type of solver the model is sent to. Basically, for `GLOPT-2` one could distinguish between a rigorous mode and an approximating mode. In `NOP-2` data can be entered in various ways.

Data can contain an *approximate number, e.g.*,

```
const A = sin(4/3*pi);
```

in rigorous mode constructs a small interval enclosing the number, and in non-rigorous mode takes an approximate floating point value (rounded towards nearest).

Sometimes, data can be an approximate number, even in rigorous mode, but if it must be rounded, the *rounding direction* is important:

`const A = sin(4/3*pi)<+>;` round towards $+\infty$,
`const A = sin(4/3*pi)<->;` round towards $-\infty$,
`const A = sin(4/3*pi)<0>;` round towards 0 .

Especially important for mathematical proofs is that some numbers are not tampered with. *Exact data* is entered as follows:

```
const A = sin(4/3*pi)!;
```

If the data is not exactly representable as a floating point number, then do one of the following depending on the mode of operation: issue a warning, construct a small interval containing the true number, or pass the value as a literal string to the solver.

In some cases, a number is really approximate

```
const A = 3.56647?
```

It is represented it by a rounded floating point number in approximate mode, and converted it to an interval. The radius of that interval should be $\frac{1}{2}$ in the first unspecified digit or 0.5*ulp*, whichever is bigger.

For many practically relevant optimization problems, data cannot be specified exactly. So in NOP-2 it is possible to specify uncertain data in three widely used ways:

```
const B = [ 3.23245, 3.23268 ];
const C = 1.3345 +- 0.0013;
const D = 1.334562(34);
```

Here B is in interval notation, C is defined using center and radius, and the declaration of D is using the engineering notation, in this case $D = [1.334528, 1.334596]$.

Of course, data items can be infinite as well:

`const I = inf;` or `const MI = -inf;`

and numbers can be unknown

`const U = ?;`

Finally, NOP-2 allows several data types: integer, real, complex integer, complex, interval, and complex interval (disk representation).

15.3.3 Sets and Lists

Sets in NOP-2 are completely different from sets in most other modeling languages; some of the set-theoretic constructions can be performed in OPL, see Chapter 17, as well.

In NOP-2 sets represent possible domains of variables, constraints, and indices (integers). After specification, a range set is transformed to a normal form, a union of closed boxes, and an integer set is converted into a list of integer tuples.

In addition to sets, *lists* can be produced by analogous constructors. The only difference is that sets never contain duplicate entries, but lists do.

There are several ways for constructing sets in NOP-2. The easiest way is by giving explicit intervals

`[-3, 4.5],`

which is possible for reals and for integers.

Ranges follow their intuitive meaning. The value in the middle specifies the increment; if it is missing the increment is 1.

`1:2:12`, `0.5:0.2:3.4`, and `1:3`

will produce the sets $\{1, 3, 5, 7, 9, 11\}$, $\{0.5, 0.7, 0.9, \ldots, 3.3\}$, and $\{1, 2, 3\}$, respectively.

Everybody with experience in using numerical software (*e.g.* Matlab) will know that roundoff problems sometimes make increments unreliable, especially if a range has to be dissected into a given number of pieces of equal sizes. So in NOP-2 there is a constructor for that:

```
0.1~10~0.9
```

constructs the set $\{0.1, 0.18, 0.26, 0.34, \ldots, 0.82, 0.9\}$ of 11 points representing 10 subintervals of (approximately) equal size.

Repeaters just produce lists of specified length.

```
1**27
```

produces a list containing 27 entries of 1.

Of course, sets can be *enumerated explicitly*

```
{ 1, 2, 3, 7, 8.2, 9, 3*e^4 },
```

but it is also possible to use *simple descriptions* like

```
{ 3+4*i | i = 1:3:22 },
```

which are very close to mathematical notation. For complicated sets more *complex description* syntax is available:

```
{{ int k,int l,real r | 4*l*m^k | k=1:n, l=1:m,
                                  r=0.5:0.03:0.8 }}.
```

All the lists can be constructed sorted or unsorted, and there are the usual set operations *union* (|), *intersection* (&), *set difference* (\), *set power* (^), and *cartesian product* (*). Finally, there is the membership relation in:

```
A in ([3,4] | [1,2])*(0~4~1)^2.
```

15.3.4 Matrices and Tensors

Since many mathematical models contain matrices, we thought it natural to include matrix and tensor operations in NOP-2.

Vectors are constructed automatically from lists or from sparse definitions, *e.g.*,

```
v = ( 97 $ 1.4 $ 3.7@1,3,9 4.8@44 );
```

which defines a vector of dimension 97. Almost all components are 1.4, except there is a 3.7 in v_1, v_3, and v_9, and a 4.8 in v_{44}.

There are many ways to specify *matrices* and *tensors* (arrangements of numbers with more than two indices). You have the choice between several shapes: dense, sparse, banded, symmetric (or hermitian), antisymmetric, unsymmetric,

and triangular. The matrices can be entered using Matlab notation, row index notation, or as a stream of numbers.

Many operators for vectors, matrices, and tensors are defined, which can be used in the models:

- Sums for vectors (x, y), matrices (A, B), and tensors (S, T) of the same shape:

  ```
  A+B,  T-S,  x+y.
  ```

- Products of vectors (x, y), and matrices (A, B), if the shapes are compatible

  ```
  A.x,  A.B,  x.y.
  ```

- Tensor products:

  ```
  A(*)B
  ```

  if A and B are matrices, this definition gives the tensor $A \otimes B$ with 4 indices.

- Indexing is very similar to Matlab, the `:` stands for the whole range of the given index. If there are more than two indices, the notation `:(list)` can be used to take the whole range for all indices specified in the list, *e.g.*,

  ```
  A[1,3],  A[:,4],  T[:,3,:,5],  S[:(1:3),6].
  ```

- As in Matlab, a whole submatrix can be assigned:

  ```
  A[1:3,:] = B[6:8,:].
  ```

- Finally, additional useful matrix operators such as

  ```
  A', det(A), tr(A), cond(A,2),
  ```

  namely the transpose, determinant, trace, and condition numbers may be used; and there are matrix constraints like

  ```
  A is psd,
  ```

  which codes a positive semidefiniteness requirement on A.

15.3.5 Stochastic and Multistage Programming

In NOP-2 stochastic programs can be specified just like ordinary optimization problems. They just involve some stochastic variables and special constraints.

Stochastic variables and their distribution functions can be specified using the following syntax:

```
stoch real xi[5];    stoch int yi[3];
```

defines vectors of 5 continuous and 3 discrete stochastic variables. For these variables distribution information can be added:

```
distr xi[1] ~ N(0,1);
```

makes ξ_1 normally distributed with mean 0 and variance 1, while

```
distr xi[2] ~ N(3.44,?);
```

specifies that ξ_2 is normally distributed with mean 3.44 and unknown variance. It is even possible to specify covariance matrices, such as

```
distr xi[3:5] ~ covar(A);
```

where the matrix A would have to be defined somewhere else in the model.

Discrete variables work just like continuous ones:

```
distr yi[1] ~ binom(12,0.3);
```

defines a binomial distribution, while

```
distr yi[2] ~ discrete((1,0.1),(2,0.3),(3,0.6));
```

declares an arbitrary discrete distribution with values and associated probabilities, and some or all of the probabilities might even be unknown:

```
distr yi[2] ~ discrete((4,?),(5,?),(9,?));
```

In order to provide the capability to model stochastic programming NOP-2 allows the use of stochastic operators in constraints such as

`Expect(xi[3])`, `Prob(xi[2])`, or `Variance(yi[1]*xi[4])`.

Coding multistage problems is done by assigning a stage to each variable.

```
stage x = 3;
```

If no stage is assigned, the variable is implicitly supposed to be in stage 0. Stages are propagated through all expressions, and an expression belongs to the maximal stage of all variables occurring in it. Note that allowed stagenumbers are all integers, so variables could be declared to be in *e.g.* stage -4.

15.3.6 Recursive Modeling and Other Components

Every NOP-2 model containing free variables can be used as a function within another model. This is done via the `call` operator, as in

```
call "subprob.nop"(x, 3.0, 1.2);
```

supposing that the subproblem has 3 free variables. Here, the first variable is kept and x is passed there, and the two other variables are fixed at 3.0 and 1.2, respectively. The return values to the `call` operator are in the subproblem specified by the `solution` record:

```
solution x[1:30];
```

In principle, even recursive modeling is possible, if `call` is combined with *conditionals* and *loops* in the model.

In addition to the presented features NOP-2 provides an interface to *black-box functions*, and *integrals*.

Finally, bounds and constraints can be defined using arbitrary sets, *e.g.*,

```
bnd x[1 3 19] in {0} | [3.1,19.223];
bnd x[2 4] in {i^2 | i=0:6 } | [36,112];
```

could be used to specify semi-continuous and partially integer variables.

15.4 Conclusion

NOP-2 is a modeling language specifically designed for global optimization. It is between an advanced input language and a modern modeling language. There are many features not readily available in other modeling languages. On the other hand, many features contained in most algebraic modeling languages are missing. The element concept for capturing the structure is an interesting idea. However, nowadays newer representations for global optimization problems are available, making the element approach less appealing.

The language is unsupported now, and its development and use have ceased. In spite of that, it contains many useful features, and we hope that some, if not all, of them will in the near future be included in some commercial modeling systems. Especially important would be rigorous data handling and matrix support.