ooOPS

A C++ Callable Object-Orientated Library for the Definition and Solution of Large, Sparse Mixed Integer Nonlinear Programming (MINLP) Problems

Draft v. 1.25

L. Liberti, P. Tsiakis, B.R. Keeping, C.C. Pantelides
Centre for Process Systems Engineering
Imperial College of Science, Technology and Medicine
London SW7 2BY
United Kingdom

30^{th} January 2002

Contents

1 Introduction

ooOPS is a library of C++ callable procedures for the definition, manipulation and solution of large, sparse mixed integer linear and nonlinear programming (MINLP) problems. In particular, *ooOPS*:

- facilitates the definition of complex sets of constraints, reducing the required programming effort to a minimum;
- allows its client programs to create and manipulate simultaneously more than one MINLP;
- provides a common interface to diverse MINLP solvers to be used without any changes to client programs.

MINLPs are optimisation problems whose objective function and constraints can, in general, contain nonlinear terms. The variables appearing in the objective function and constraints are generally restricted to lie between specified lower and upper bounds. Furthermore, some of these variables may be restricted to integer values only. The aim of the optimisation is to determine values of the variables that minimise or maximise the objective function while satisfying the constraints and all other restrictions imposed on them.

A simple mathematical description of an MINLP can be written as:

[Flat MINLP]:

$$\min_x \phi(x)+c^T x \tag{1}$$

subject to:

$$a \leq f(x) + Ax \leq b \tag{2}$$

$$x^l \leq x \leq x^u \tag{3}$$

$$x_i \in \mathbb{Z} \qquad \forall i \in I \subseteq \{1,..,n\} \tag{4}$$

where $x, x^l, x^u, c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A : \mathbb{R}^n \to \mathbb{R}^m$, $\phi : \mathbb{R}^n \to \mathbb{R}$ and $f : \mathbb{R}^n \to \mathbb{R}^m$. Thus, the variables x are characterised by an index $i = 1,..,n$; all constraints are expressed as inequalities of the form ≤ 0 and are indexed over the discrete domain $1,..,m$.

The above general formulation also embeds three special cases:

- *Mixed Integer Linear Programming (MILP) Problems.*
  In this case, $\phi(x) = 0$ and $f(x) = 0$.

- *Nonlinear Programming (NLP) Problems.*
  In this case, $I = \emptyset$.

- *Linear Programming (LP) Problems.*
  In this case, $\phi(x) = 0$, $f(x) = 0$ and $I = \emptyset$.

The *ooOPS* software design aims to support the definition and solution of all these special cases, with minimal overhead being incurred because of the generality of the overall software.

Albeit quite general, the above MINLP form is not necessarily easy to construct and/or manipulate. A major reason for this is that the variables and constraints are maintained as unstructured "flat" lists or sets which may contain thousands of elements. On the other hand, most mathematical formulations of practical problems in terms of MINLPs are expressed in terms of a relatively small number of distinct sets of variables and constraints.

For example, in a typical network flow problem, a commonly occurring set of variables would be the flow of material from one node in the network to another, while a typical set of constraints would be the conservation of material arriving at, and leaving any node i in the network.

Of course, each such set of variables (or constraints) may have multiple elements, each corresponding to an individual variable (or constraint). An indexed set representation is usually employed for notational purposes. For instance, F_{ij} could represent the flow from node i to node j, while $\mathcal{C}_k$ could represent the set of conservation constraints:

$$\sum_{i \neq k} F_{ik} = \sum_{j \neq k} F_{kj}, \quad \forall \text{ node } k$$

In view of the above discussion, *ooOPS* allows types of variables and constraints to be defined in a structured fashion as sets of an arbitrary number of dimensions.

2 Fundamental Concepts

2.1 Object Classes

ooOPS is designed as object-orientated software recognising two major classes of objects, each with its own interface:

1. The `ops` object
   An `ops` object is a software representation of a MINLP problem. The corresponding interface, discussed in detail in section 4, provides the following functionality:
   - It allows MINLP objects to be constructed and modified in a structured manner.
   - It allows access to all information pertaining to the MINLP.
   - It provides the equivalent simple [**Flat MINLP**] form of the structured MINLP (see section 1).
2. The `opssystem` object
   This is formed by the combination of an MINLP object (see above) with a code ("solver") for the numerical solution of MINLP problems. The corresponding interface, discussed in detail in section 5, provides the following functionality:
   - It allows the behaviour of the solver to be configured via the specification of any algorithmic parameters that the solver may support.
   - It permits the solution of the MINLP.

2.2 Multidimensional Sets in ooOPS

As detailed later in the document, an MINLP is characterised by a number of distinct multidimensional sets of variables and constraints. We note that:

- A multidimensional set is an ordered set whose elements can be accessed through a list of indices.
- The *dimensionality* of a multidimensional set is given by the length of the index lists (i.e. the number of dimensions) and the range of each index in the list.

- The *dimensionality size* of a multidimensional set is the list of its dimension sizes. More precisely, a multidimensional set having n dimensions, with index i_j ranging from i_J^L to i_j^U for each j between 1 and n, has *dimensionality size*:

$$(i_1^U - i_1^L + 1, \ldots, i_n^U - i_n^L + 1).$$

- A scalar has dimensionality size (1).

For instance, consider a 3-dimensional variable set $X(i, j, k)$, with the first dimension varying from $i = 0$ to $i = 10$, the second dimension from $j = 1$ to $j = 5$, and the third dimension from $k = -1$ to $k = +1$. We can view this as a multidimensional set, the dimensionality of which is characterised by the number of dimensions (3, in this case) and the ranges of each dimension $(0 : 10, 1 : 5, -1 : 1)$. The dimensionality size is the list $(11, 5, 3)$ $(= 10 - 0 + 1, 5 - 1 + 1, 1 - (-1) + 1)$. The total number of elements of this set is 165 $(= 11 \times 5 \times 3)$.

ooOPS makes extensive use of the concept of *slices* as a convenient way of referring to subsets of multidimensional sets. In general:

- A slice $[s^L : s^U]$ of an N-dimensional set is *defined* by a pair of N-dimensional integer vectors $s_i^L, i = 1, .., N$ and $s_i^U, i = 1, .., N$ where $s_i^L \leq s_i^U$.

- The element at position $(k_1, k_2, .., k_N)$ of the set belongs to the slice $[s^L : s^U]$ if and only if:

$$s_i^L \leq k_i \leq s_i^U \ , \quad \forall \ i = 1, .., N \tag{5}$$

Here are some examples:

- The slice $[(2) : (5)]$ of a 1-dimensional set X denotes the elements $X_i, i = 2, 3, 4, 5$.

- The slice $[(2, 3) : (5, 4)]$ of a 2-dimensional set X denotes the elements $X_{ij}, i = 2, 3, 4, 5, j = 3, 4$.

- The slice $[(2, 3, 3) : (5, 3, 4)]$ of a 3-dimensional set X denotes the elements $X_{i3k}, i = 2, 3, 4, 5, k = 3, 4$.

- The slice $[(2, 3, 3, 4) : (2, 3, 3, 4)]$ of a 4-dimensional set X denotes the single element X_{2334}.

2.3 Constants, Variables, Constraints and Objective Function

Most optimization problems involve arithmetic constants. In *ooOPS*, these can be organized in one or more multidimensional sets. A *constant set* is characterised by the following information:

- its name;
- the dimensionality of the set;
- the current value of each element of the set;

The variables to be determined by the optimization are also organized in one or more multidimensional sets. A *variable set* is characterised by the following information:

- its name;
- the type of all variables in this set (continuous or integer);
- the dimensionality of the set ;
- the current value of each element of the set;
- the upper and lower bounds of the value of each element of the set.

Similarly, the constraints in the optimization problem are also organized in one or more multidimensional sets.

A *constraint set* is characterised by the following information:

- its name;
- the lower and upper bound of the constraint;
- the dimensionality of the set;
- the variables occurring in the linear parts of these constraints and the corresponding coefficients (see note 1 in section 2.5 below);
- the expression (see below) defining the nonlinear part of these constraints.

Most of the information characterising a set of constants, variables or constraints is common to *all* elements of the set. The *only* exceptions to this rule are:

- the values of elements of constant sets may differ from element to element;
- the values and bounds of elements of variable sets may differ from element to element;
- the constraint bounds of the constraint sets may differ from element to element.

Each MINLP object has a unique *objective function*. This is characterised by:

- the name of the objective function
- the type of the problem (minimisation or maximisation)
- the variables occurring in the linear part of the objective function and the corresponding coefficients (see note 1 in section 2.5 below);
- the expression defining the nonlinear part of the objective function.

2.4 Nonlinear Expressions and Constants

A nonlinear *expression* is, in general, built hierarchically from the algebraic combination of variables, constants (see below) and other expressions by using operators and functions. Expressions are characterised by the following information:

- their name;
- their dimensionality size;
- whether they represent variables, constants or operators.

An expression represents a valid algebraic expression. The operators and functions that can be used to combine variables, constants and other expressions are given in the following table:

Name	Meaning
Binary Arithmetic Operators	
`sum`	addition
`difference`	subtraction
`product`	multiplication
`ratio`	division
`power`	exponentiation
Unary Arithmetic Operators	
`minus`	unary minus
Unary Transcendental Functions	
`log`	natural logarithm
`exp`	exponential
`sin`	sine
`cos`	cosine
`tan`	tangent
`cot`	cotangent
`sinh`	hyperbolic sine
`cosh`	hyperbolic cosine
`tanh`	hyperbolic tangent
`coth`	hyperbolic cotangent
`sqrt`	square root

In general, expressions are multidimensional sets with the special property that the range of each dimension starts from 1. The dimensionality size of an expression can be determined from the dimensionality sizes of its variables, constants and subexpressions and the type of its constituent operators, according to the rules given in the following tables:

Binary Arithmetic Operators		
dimensionality size of 1st operand	dimensionality size of 2nd operand	dimensionality size of result
$(s_1, \ldots, s_n)$	$(s_1, \ldots, s_n)$	$(s_1, \ldots, s_n)$
(1)	$(s_1, \ldots, s_n)$	$(s_1, \ldots, s_n)$
$(s_1, \ldots, s_n)$	(1)	$(s_1, \ldots, s_n)$
otherwise		illegal

Unary Operators and Functions	
dimensionality size of operand	dimensionality size of result
$(s_1, \ldots, s_n)$	$(s_1, \ldots, s_n)$

2.5 Notes

1. *Variable occurrences in linear parts of constraints*
   An important part of the characterisation of the linear part of a constraint is the information on which variables actually occur in it, and the coefficients that multiply each such variable.

   In *ooOPS*, such occurrences are specified as a variable slice occurring in a constraint slice with a given coefficient. This simply means that:

   Every element of the variable slice occurs in *each* element of the constraint slice, always with the *same* coefficient.

   Of course, the client constructing an MINLP object using the facilities provided by *ooOPS* can add any number of such occurrences. Moreover, if any such specification involves an element of a variable vector that already appears[1] in one or more of the specified constraints, the later specification overrides the earlier one.

2. *Variable occurrences in linear part of objective function*
   An important part of the characterisation of the linear part of the objective function is the information on which variables actually occur in it, and the coefficients that multiply each such variable.

   In *ooOPS* , such occurrences are specified as a variable slice occurring in the objective function with a given coefficient. This simply means that:

   Every element of the variable slice occurs in the objective function always with the *same* coefficient.

   Of course, the client constructing an MINLP object using the facilities provided by *ooOPS* can add any number of such occurrences. Moreover, if any such specification involves an element of a variable vector that already appears[2] in the objective function, the later specification overrides all earlier ones.

3. *Nonlinear expression occurrences in constraints*
   An important part of the characterisation of a constraint is its nonlinear part. In *ooOPS*, this can be specified by assigning one or more expressions to different slices of the constraint. Each expression must either be a scalar or its dimensionality size must match that of the constraint slice to which it is assigned, in which case each element

[1] as a result of earlier specifications
[2] as a result of earlier specifications

of the slice is set the corresponding the corresponding element of the expression.

Note that, unlike linear variable occurrences, expression elements do not accumulate. If a new expression is assigned to a constraint element which has already been assigned an expression, the later assignment overrides the earlier one.

4. *Nonlinear expression occurrences in objective function*
   An important part of the characterisation of the objective function is its nonlinear part. In *ooOPS*, this is specified as a (scalar) expression occurring in the objective function.

   Note that, unlike linear variable occurrences, expression elements do not accumulate. if a new expression is assigned to an objective function which has already been assigned an expression, the later assignment overrides the earlier one.

3 General Software Issues

3.1 Software Constants

ooOPS defines two constants which should primarily be used for the specification of lower and upper bounds of variables, as well as constraint right hand side constants. These are:

- `ooOPSPlusInfinity`
  Setting the upper bound of a variable to `PlusInfinity` implies that this variable is effectively unbounded from above.
  Setting the upper bound of an inequality constraint to `PlusInfinity` is the standard way to represent an inequality of the form $c(x) \geq LB$.

- `ooOPSMinusInfinity`
  Setting the lower bound of a variable to `MinusInfinity` implies that this variable is effectively unbounded from below.
  Setting the lower bound of an inequality constraint to `MinusInfinity` is the standard way to represent an inequality of the form $c(x) \leq UB$.

3.2 The `si`, `sd` and `li` Argument Types and the `IntSeq` Auxiliary Function

In order to describe the arguments to the various methods of its object classes, *ooOPS* introduces the following C++ type definition:

- `si`: a sequence of integers
- `sd`: a sequence of doubles
- `li`: a list of integers

The precise implementation of this type is irrelevant as *ooOPS* also provides an auxiliary function for its construction. In particular, the `IntSeq` function takes a list of any number of integers and returns a vector of type `si`.

For example, the C++ code segment:

```
si* SetSize = IntSeq (3,7,4) ;
```

creates a sequence called `SetSize` of three integers (3, 7 and 4). This may then, for instance, be passed as an input argument to a method to create a new 3-dimensional variable set, with the lengths of the three dimensions being specified in `SetSize`.

4 The MINLP Object Class

4.1 MINLP Instantiation: The `NewMINLP` Function

Declaration: `ops* NewMINLP()`

Function: Creates a new empty MINLP.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`ops*`	the MINLP object

Notes: None

Example of usage:

The following creates a new MINLP called `NetOpt` :

```
ops* NetOpt = NewMINLP() ;
```

4.2 MINLP Construction Methods

4.2.1 Method `NewContinuousVariable`

Declaration: `void NewContinuousVariable(string vname, si* dimLB, si* dimUB, double LB, double UB, double value)`

Function: Creates a new set of continuous variables with given name, domain, bounds and value.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set to be created
`dimLB`	`si*`	lower bounds of the variable set dimensions
`dimUB`	`si*`	upper bounds of the variable set dimensions
`LB`	`double`	lower bound
`UB`	`double`	upper bound
`value`	`double`	value

Arguments returned to client: None

Notes:

- The specified name must be unique among all variables (continuous *or* integer) in the MINLP as it will be used to identify the variable in all future communications with the MINLP object.
- In general, a variable is represented by a multidimensional set (*cf.* section 2.2). The arguments `dimLB` and `dimUB` are sequences of integers, representing respectively the lower and upper bounds of the domains of the individual dimensions. In the case of the example of set X mentioned in section 2.2, these two sequences would be (0, 1, -1) and (10, 5, 1) respectively.
- The length of the above integer sequences must be equal to each other and implicitly determine the number of dimensions of the variable.
- Scalar variables should be specified as 1-dimensional sets of size 1.
- The index of the individual elements for each dimension ranges from the corresponding element of `dimLB` to the corresponding element of `dimUB`.
- The specified lower bound, upper bound and value must satisfy:

$$LB \leq value \leq UB$$

- All elements of the variable set being created are given the same lower and upper bounds as well as the same value. However, these quantities

may subsequently be changed for individual elements or slices of the set (see sections 4.3.1 and 4.3.2).

Example of usage:

The following creates a 2-dimensional continuous variable called `MaterialFlow`, with individual elements `MaterialFlow` $(i, j), i = 0, .., 4, j = 1, .., 6$ within an existing `ops` object called `NetOpt`. All elements of the new variable are initialised with a lower bound of 0.0, an upper bound of 100.0 and a current value of 1.0.

```
NetOpt->NewContinuousVariable("MaterialFlow", IntSeq(0,1),
                              IntSeq(4,6), 0.0, 100.0, 1.0);
```

4.2.2 Method `NewIntegerVariable`

Declaration: `void NewIntegerVariable(string vname, si* dimLB, si* dimUB, int LB, int UB, int value)`

Function: Creates a new set of integer variables with given name, size, bounds and value.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set to be created
`dimLB`	`si*`	lower bounds of the variable set dimensions
`dimUB`	`si*`	upper bounds of the variable set dimensions
`LB`	`int`	lower bound
`UB`	`int`	upper bound
`value`	`int`	value

Arguments returned to client: None

Notes:

- The specified name must be unique among all variables (continuous *or* integer) in the MINLP as it will be used to identify the variable in all future communications with the MINLP object.
- In general, a variable is represented by a multidimensional set (*cf.* section 2.2). The arguments `dimLB` and `dimUB` are sequences of integers, representing respectively the lower and upper bounds of the domains of the individual dimensions. In the case of the example of set X mentioned in section 2.2, these two sequences would be (0, 1, -1) and (10, 5, 1) respectively.
- The length of the above integer sequences must be equal to each other and implicitly determine the number of dimensions of the variable.
- Scalar variables should be specified as 1-dimensional sets of size 1.
- The index of the individual elements for each dimension ranges from the corresponding element of `dimLB` to the corresponding element of `dimUB`.
- The specified lower and upper bounds ,and value must satisfy:

$$LB \leq value \leq UB$$

- All elements of the variable set being created are given the same lower and upper bounds as well as the same value. However, these quantities may subsequently be changed for individual elements or slices of the set (see sections 4.3.1 and 4.3.2).

Example of usage:

The following creates a 1-dimensional integer (binary) variable called `PlantExists` of length 3 with individual elements `PlantExists`$(i), i = 1, .., 3$ within an existing `ops` object called `NetOpt`. All elements of the new variable are initialised with a lower bound of 0, an upper bound of 1 and a current value of 0.

```
NetOpt->NewIntegerVariable("PlantExists", IntSeq(1), IntSeq(3),
                           0, 1, 0);
```

4.2.3 Method `NewConstraint`

Declaration: `void NewConstraint(string cname, si* dimLB, si* dimUB, double LB, double UB)`

Function: Creates a new set of constraints with given name, size, type and bounds.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`cname`	`string`	name of constraint set to be created
`dimLB`	`si*`	lower bounds of the constraint set dimensions
`dimUB`	`si*`	upper bounds of the constraint set dimensions
`LB`	`double`	lower bound
`UB`	`double`	upper bound

Arguments returned to client: None

Notes:

- The specified name must be unique among all constraints in the MINLP as it will be used to identify the constraint in all future communications with the MINLP object.
- In general, a constraint is represented by a multidimensional set (*cf.* note 2.2 in section 2.5). The arguments `dimLB` and `dimUB` are sequences of integers, representing respectively the lower and upper bounds of the domains of the individual dimensions. Their length must be equal and determine the number of dimensions of the constraint set.
- Scalar constraints should be specified as 1-dimensional sets of size 1.
- The index of the individual elements for each dimension ranges from the corresponding element of `dimLB` to the corresponding element of `dimUB`.
- All elements of the constraint set being created are assigned lower and upper bounds. However, these may subsequently be changed (see section 4.3.3).
- Initially, the constraint does not contain any variables occurring linearly in it. It is also assigned a null nonlinear part.

Example of usage:

Creates a new constraint `UniqueAllocation` with bounds fixed at 1.

```
NetOpt->NewConstraint("UniqueAllocation", IntSeq(1), IntSeq(1),
                      1.0, 1.0);
```

4.2.4 Method `AddLinearVariableSliceToConstraintSlice`

Declaration: `void AddLinearVariableSliceToConstraintSlice(string vname, si* vdimLB. si* vdimUB, string cname, si* cdimLB, si* cdimUB, double coefficient)`

Function: Adds a linear occurrence of every element of a specified variable slice to each element of a specified constraint slice, always with the specified coefficient.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	variable set, elements of which are to be added
`vdimLB`	`si*`	lower bounds of the variable set slice to be added to constraints
`vdimUB`	`si*`	upper bounds of the variable set slice to be added to constraints
`cname`	`string`	name of constraint set, elements of which are to receive the new variable occurrences
`cdimLB`	`si*`	lower bounds of the constraint set slice to receive variable occurrences
`cdimUB`	`si*`	upper bounds of the constraint set slice to receive variable occurrences
`coefficient`	`double`	coefficient of variables in the constraints

Arguments returned to client: None

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections 4.2.1 and 4.2.2 respectively).
- The specified constraint set must have already been created using the `NewConstraint` method (see section 4.2.3).
- If an element of the variable slice has already been declared to occur in an element of constraint slice via an earlier invocation of this method, then the current specification supersedes the earlier one.
- A value of 0.0 for the specified `coefficient` has the effect of removing an existing occurrence of a variable in a constraint.

Examples of usage:

The following creates a new inequality constraint called `SourceFlowLimit` within an existing `ops` object called `NetOpt` for each of the 5 source nodes

in a network with a right hand side of 450. It then adds to the constraint for each node all the elements of the corresponding row of the variable `MaterialFlow` (*cf.* example in section 4.2.1) with a coefficient of 1.

```
NetOpt->NewConstraint ("SourceFlowLimit",
                       IntSeq(0), IntSeq(4), '<', 450.0);
for (int i=0; i<=4; i++)
  NetOpt->AddLinearVariableSliceToConstraintSlice(
    "MaterialFlow",     IntSeq(i,1), IntSeq(i,6),
    "SourceFlowLimit",  IntSeq(i),   IntSeq(i),   1.0);
```

The following adds an occurrence of all elements of the variable `PlantExists` (*cf.* example in section 4.2.2) to the scalar constraint `UniqueAllocation` (*cf.* example in section 4.2.3) within an existing `ops` object called `NetOpt`.

```
NetOpt->AddLinearVariableSliceToConstraintSlice
        ("PlantExists",      IntSeq(1), IntSeq(3),
         "UniqueAllocation", IntSeq(1), IntSeq(3), 1.0);
```

4.2.5 Method `NewConstant`

Declaration: `void NewConstant(string kname, si* kdimLB, si* kdimUB, double value)`

Function: Creates a new constant set of specified dimensionality, each element of which is initialized to the specified value.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`kname`	`string`	name of the constant set/slice being created
`kdimLB`	`si*`	lower bounds of constant set/slice
`kdimUB`	`si*`	upper bounds of constant set/slice
`value`	`value`	initial value

Arguments returned to client: None

Notes:

- None

Examples of usage:

The following code creates a constant called `Const1` within an existing `ops` object called `NetOpt`, consisting of a three-dimensional vector whose components are all 1.

```
NetOpt->NewConstant("Const1", IntSeq(1), IntSeq(3), 1);
```

4.2.6 Method `NewConstantExpression`

Declaration: `void NewConstantExpression(string ename, string kname, si* kdimLB, si* kdimUB)`

Function: Creates a new expression consisting of a constant set or constant slice.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	name of the expression being created
`kname`	`string`	name of the constant set/slice
`kdimLB`	`si*`	lower bounds of constant set/slice
`kdimUB`	`si*`	upper bounds of constant set/slice

Arguments returned to client: None

Notes:

- If an expression called `ename` already exists within the *ops* object, the new expression will overwrite it.

Examples of usage:

The following code creates an expression called `Expr1` (within an existing `ops` object called `NetOpt`) from the constant set `Const1`.

```
NetOpt->NewConstant("Const1", IntSeq(1), IntSeq(3), 1);
NetOpt->NewConstantExpression("Expr1", "Const1",
                              IntSeq(1), IntSeq(3));
```

4.2.7 Method `NewVariableExpression`

Declaration: `void NewVariableExpression(string ename, string vname, si* vdimLB, si* vdimUB*)`

Function: Creates a new expression consisting of a variable set or a variable slice.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	name of the expression
`vname`	`string`	name of the variable set/slice
`vdimLB`	`si*`	lower bounds of the variable set/slice
`vdimUB`	`si*`	upper bounds of the variable set/slice

Arguments returned to client: None

Notes:

- If an expression called `ename` already exists within the *ops* object, the new expression will overwrite it.

Examples of usage:

The following code creates a variable expression called `Expr2` (within an existing `ops` object called `NetOpt`) consisting of the vector (x_1, x_2, x_3).

```
NetOpt->NewContinuousVariable("x", IntSeq(1), IntSeq(3), -1, 1, 0);
NetOpt->NewVariableExpression("Expr2", "x", IntSeq(1), IntSeq(3));
```

4.2.8 Method `BinaryExpression`

Declaration: `void BinaryExpression(string ename, string ename1, si* edimLB1, si* edimUB1, string ename2, si* edimLB2, si* edimUB2, string binaryoperator)`

Function: Creates a new expression representing a binary operation between the expressions specified in the argument.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	name of the new expression
`ename1`	`string`	name of expression representing the first operand
`edimLB1`	`si*`	lower bound of slice to be used from first expression
`edimUB1`	`si*`	upper bound of slice to be used from first expression
`ename2`	`string`	name of expression representing the second operand
`edimLB2`	`si*`	lower bound of slice to be used from second expression
`edimUB2`	`si*`	upper bound of slice to be used from second expression
`binaryoperator`	`string`	label of the binary operator to be used.

Arguments returned to client: None

Notes:

- The dimensionality size of the new expression depends on those of its operands (see table in section 2.4).
- The argument `binaryoperator` describes the type of binary operator in the expression. It can be one of the following strings: `"sum"`, `"difference"`, `"product"`, `"ratio"`, `"power"` (cf. section 2.4).
- If an expression called `ename` already exists within the *ops* object, the new expression will overwrite it.

Examples of usage: The following code creates a nonlinear term x_1x_2 within an existing `ops` object called `NetOpt`.

```
NetOpt->NewContinuousVariable("x", IntSeq(1), IntSeq(2), -1, 1, 0);
NetOpt->BinaryExpression("NLT", "x", IntSeq(1), IntSeq(1),
                         "x", IntSeq(2), IntSeq(2), "product");
```

4.2.9 Method `UnaryExpression`

Declaration: `void UnaryExpression(string ename, string ename1, si* edimLB1, si* edimUB1, string unaryoperator)`

Function: Creates a new expression representing a unary operation on the expression specified in the argument.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	name of the new expression
`ename1`	`string`	name of expression representing the operand
`edimLB1`	`si*`	lower bound of slice to be used from the operand expression
`edimUB1`	`si*`	upper bound of slice to be used from the operand expression
`unaryoperator`	`string`	label of the unary arithmetic operator or unary transcendental function to be used

Arguments returned to client: None

Notes:

- The dimensionality and dimension sizes of the new expression is the same as that of its operands.
- The argument `unaryoperator` describes the type of unary operator in the expression. It can be one of the following strings (the meaning is self-explanatory): `"minus"`, `"log"`, `"exp"`, `"sin"`, `"cos"`, `"tan"`, `"cot"`, `"sinh"`, `"cosh"`, `"tanh"`, `"coth"` (cf. section 2.4).
- If an expression called `ename` already exists within the *ops* object, the new expression will overwrite it.

Examples of usage: The following code creates a nonlinear term $\log(x)$ within an existing `ops` object called `NetOpt`.

```
NetOpt->NewContinuousVariable("x", IntSeq(1), IntSeq(3), -1, 1, 0);
NetOpt->UnaryExpression("NLT", "x", IntSeq(1), IntSeq(3), "log");
```

4.2.10 Method `AssignExpressionSliceToConstraintSlice`

Declaration: `void AssignExpressionSliceToConstraintSlice (string ename, si* edimLB, si* edimUB, string cname, si* cdimLB, si* cdimUB)`

Function: Assigns the specified expression slice to the specified constraint slice.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	the name of the expression to be set
`edimLB`	`si*`	lower bound of expression slice
`edimUB`	`si*`	upper bound of expression slice
`cname`	`string`	name of constraint set to which the expression must be set
`cdimLB`	`si*`	lower bounds of the constraint set slice to be set with the expression elements
`cdimUB`	`si*`	upper bounds of the constraint set slice to be set with the expression elements

Arguments returned to client: None

Notes:

- The specified constraint set must have already been created using the `NewConstraint` method (see section 4.2.3).
- The specified expression either must be a scalar or its dimensionality size must match exactly that of the constraint slice. In the former case, the same scalar expression will be assigned to each of the constraints in the slice; in the latter, each element in the expression slice will be assigned to the corresponding element in the constraint slice.

Examples of usage:

The following code creates the term $\frac{x}{\log(x)}$ within an existing `ops` object called `NetOpt` and assigns it to the constraint `UniqueAllocation`.

```
NetOpt->NewContinuousVariable("x", IntSeq(1), IntSeq(3), -1, 1, 0);
NetOpt->UnaryExpression("NLT", "x", IntSeq(1), IntSeq(3), "log");
NetOpt->BinaryExpression("NLT", "x",
                         IntSeq(1), IntSeq(3), "NLT",
                         IntSeq(1), IntSeq(3), "ratio");
NetOpt->AssignExpressionSliceToConstraintSlice
        ("NLT",              IntSeq(1), IntSeq(3),
         "UniqueAllocation", IntSeq(1), IntSeq(3));
```

4.2.11 Method `NewObjectiveFunction`

Declaration: `void NewObjectiveFunction(string oname, string otype)`

Function: Specifies the name and the type for the objective function for the MINLP.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`oname`	`string`	objective function name
`otype`	`string`	objective function type

Arguments returned to client: None

Notes:

- The objective function created by this method does not contain any variables either linearly or nonlinearly.
- The type of the objective function must be a string of at least 3 characters; any characters beyond the third one are ignored. Valid type specifications are `"min"` and `"max"` denoting minimisation and maximisation respectively. The case of the characters in the type specification is irrelevant.
- If the method is invoked more than once for a given MINLP, then each invocation supersedes all earlier ones and any information associated with the previous objective function (*e.g.* on the variables occurring in it) is lost.

Examples of usage:

The following creates an objective function called `TotalProfit` within an existing `ops` object called `NetOpt`, that is to be maximised by the solution of the MINLP:

```
NetOpt->NewObjectiveFunction{"TotalProfit", "max") ;
```

4.2.12 Method `AddLinearVariableSliceToObjectiveFunction`

Declaration: `void AddLinearVariableSliceToObjectiveFunction (string vname, si* dimLB, si* dimUB, double coefficient, string oname)`

Function: Adds a linear occurrence of every element of a specified variable slice to the specified objective function using the specified coefficient.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set, elements of which are to be added
`dimLB`	`si*`	lower bounds of the variable set slice to be added to objective function
`dimUB`	`si*`	upper bounds of the variable set slice to be added to objective function
`coefficient`	`double`	coefficient of variables in the objective function
`oname`	`string`	name of current objective function

Arguments returned to client: None

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections 4.2.1 and 4.2.2 respectively).
- If an element of the variable slice has already been declared to occur in the objective function via an earlier invocation of this method, then the current specification supersedes the earlier one.
- A value of 0.0 for the specified `coefficient` has the effect of removing an existing occurrence of a variable in a constraint.
- The specified objective function must be the current objective function which must have already been created using the `NewObjectiveFunction` method (see section 4.2.11).

Examples of usage:

The following adds the elements of variables `MaterialFlow` (*cf.* example in section 4.2.1) relating to destination node 1 to the objective function `TotalProfit` (*cf.* example in section 4.2.11) within the existing `ops` object called `NetOpt`. A coefficient of 100 is used. It then subtracts from the same objective function the sum of the integer variables `PlantExists` (*cf.* example in section 4.2.2) multiplied by a coefficient of -0.1:

```
NetOpt->AddLinearVariableSliceToObjectiveFunction
        ("MaterialFlow", IntSeq(0, 1), IntSeq(4,1),
         100.0, "TotalProfit");
NetOpt->AddLinearVariableSliceToObjectiveFunction
        ("PlantExists",  IntSeq(1), IntSeq(3),
         -0.1,  "TotalProfit");
```

4.2.13 Method `AssignExpressionToObjectiveFunction`

Declaration: `void AssignExpressionToObjectiveFunction (string ename, string oname)`

Function: Sets the specified expression to the specified objective function.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	expression to be set
`oname`	`string`	name of current objective function

Arguments returned to client: None

Notes:

- The specified objective function must be the current objective function which must have already been created using the `NewObjective-Function` method (see section 4.2.11).
- The expression must be scalar.

Examples of usage:

The following code creates the scalar nonlinear term x_1x_2 and assigns it to the objective function `TotalProfit` within an existing `ops` object called `NetOpt`.

```
NewContinuousVariable("x", IntSeq(1), IntSeq(2), -1, 1, 0);
BinaryExpression("NLT", "x", IntSeq(1), IntSeq(1),
                 "x", IntSeq(2), IntSeq(2), "product");
AssignExpressionToObjectiveFunction("NLT", "TotalProfit");
```

4.3 MINLP Modification Methods

4.3.1 Method `SetVariableValue`

Declaration: `void SetVariableValue(string vname, si* dimLB, si* dimUB, double value)`

Function: Sets the current value of the elements of a specified slice of a specified variable set, overriding their previous values.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set, elements of which are to be modified
`dimLB`	`si*`	lower bounds of the variable set slice to be modified
`dimUB`	`si*`	upper bounds of the variable set slice to be modified
`value`	`double`	new value for elements to be modified

Arguments returned to client: None

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections 4.2.1 and 4.2.2 respectively).
- The specified value must lie between the lower and upper bounds for *every* element of the specified variable slice.

Examples of usage:

The following modifies all elements of the variable set `MaterialFlow` (*cf.* example in section 4.2.1) pertaining to source 3 to a new value of 150:

```
SetVariableValue ("MaterialFlow", IntSeq(3, 1), IntSeq(3,6),
                  150.0) ;
```

4.3.2 Method `SetVariableBounds`

Declaration: `void SetVariableBounds(string vame, si* dimLB, si* dimUB, double LB, double UB)`

Function: Sets the current lower and upper bounds of the elements of a specified slice of a specified variable set, overriding their previous values.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set, elements of which are to be modified
`dimLB`	`si*`	lower bounds of the variable set slice to be modified
`dimUB`	`si*`	upper bounds of the variable set slice to be modified
`LB`	`double`	new value of lower bound for elements to be modified
`UB`	`double`	new value of upper bound for elements to be modified

Arguments returned to client: None

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections 4.2.1 and 4.2.2 respectively).
- The specified bounds must satisfy $LB \leq UB$.

Examples of usage:

The following modifies the bounds of all elements of the variable set `MaterialFlow` (*cf.* example in section 4.2.1) pertaining to source 3 to new values of 10 and 20 respectively:

```
SetVariableBounds("MaterialFlow", IntSeq(3, 1), IntSeq(3,6),
                  10.0, 20.0) ;
```

The following sets both bounds of an element of the variable set `PlantExists` (*cf.* example in section 4.2.2) to 1, thereby effectively fixing the value of this variable in any MINLP solution also to 1:

```
SetVariableBounds("PlantExists",  IntSeq(2), IntSeq(2),
                  1.0, 1.0) ;
```

4.3.3 Method `SetConstraintBounds`

Declaration: `void SetConstraintBounds(string cname, si* dimLB, si* dimUB, double LB, double UB)`

Function: Sets the lower and upper bounds of a specified slice of a specified constraint set, overriding any previous bounds.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`cname`	`string`	name of constraint set, elements of which are to be modified
`dimLB`	`si*`	lower bounds of the constraint set slice to be modified
`dimUB`	`si*`	upper bounds of the constraint set slice to be modified
`LB`	`double`	new lower bound for elements to be modified
`UB`	`double`	new upper bound for elements to be modified

Arguments returned to client: None

Notes:

- The specified constraint set must have already been created using the `NewConstraint` method (see section 4.2.3).
- The specified bounds must satisfy $LB \leq UB$.

Examples of usage:

The following modifies the lower and upper bounds of constraint `Unique-Allocation` (*cf.* example in section 4.2.3) to infinity:

```
SetConstraintBounds("UniqueAllocation",  IntSeq(0), IntSeq(0),
                    MinusInfinity, PlusInfinity);
```

This modification effectively de-activates the `UniqueAllocation` constraint.

4.3.4 Method `SetConstantValue`

Declaration: `void SetConstantValue(string kname, si* indexlist, double value)`

Function: Sets the value of the constant element pointed to by the specified index list to the specified value.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`kname`	`string`	name of constant set containing the element to be modified
`indexlist`	`si*`	list of indices which point to the element to be modified
`value`	`double`	new value to be assigned to element

Arguments returned to client: None

Notes:

- The indices in the index list must lie between the respective index bounds defining the dimensionality of the constant set.

4.3.5 Method `SetConstantSliceValue`

Declaration: `void SetConstantSliceValue(string kname, si* kdimLB, si* kdimUB, double value)`

Function: Sets the values of the constant elements in the specified slice to the specified value.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`kname`	`string`	name of constant set containing the elements to be modified
`kdimLB`	`si*`	lower bounds of the constant set slice to be modified
`kdimUB`	`si*`	upper bounds of the constant set slice to be modified
`value`	`double`	new value to be assigned to elements

Arguments returned to client: None

Notes:

- None.

4.3.6 Method `SetKeyVariable`

Declaration: `void SetKeyVariable(string vame, si* dimLB, si* dimUB)`

Function: Sets the elements of a specified slice of a specified variable set to be used as key variables for the decomposition algorithm.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set, elements of which are to be modified
`dimLB`	`si*`	lower bounds of the variable set slice to be modified
`dimUB`	`si*`	upper bounds of the variable set slice to be modified

Arguments returned to client: None

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections 4.2.1 and 4.2.2 respectively).

Examples of usage:

The following sets all elements of the variable set `MaterialFlow` (*cf.* example in section 4.2.1) to be key variables:

```
SetKeyVariable ("MaterialFlow", IntSeq(3, 1), IntSeq(3,6)) ;
```

The following sets an element of the variable set `PlantExists` (*cf.* example in section 4.2.2) to be a key variable:

```
SetKeyVariable ("PlantExists",  IntSeq(2), IntSeq(2)) ;
```

4.4 Structured MINLP Information Access Methods

4.4.1 Method `GetVariableInfo`

Declaration: `void GetVariableInfo(string vname, si* dimLB, si* dimUB, double* value, double* LB, double* UB)`

Function: Returns the current values and lower and upper bounds of all elements of a specified slice of a specified set of variables.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set on which information is required
`dimLB`	`si*`	lower bounds of the variable set slice on which information is required
`dimUB`	`si*`	upper bounds of the variable set slice on which information is required

Arguments returned to client:

Argument	Type	Value on Exit
`value`	`double*`	pointer to set of real numbers containing current values of elements in specified slice
`LB`	`double*`	pointer to set of real numbers containing lower bounds of elements in specified slice
`UB`	`double*`	pointer to set of real numbers containing upper bounds of elements in specified slice

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections 4.2.1 and 4.2.2 respectively).
- In the case of multidimensional variable sets, the information in the sets `value`, `LB` and `UB` is ordered so that the last index varies fastest, followed by the penultimate index, and so on until the first index which varies the most slowly. For example, a 2-dimensional set is ordered by rows.

Examples of usage:

The following returns the information on a slice of the variable `MaterialFlow` (*cf.* example in section 4.2.1):

```
double* CurrentFlows ;
double* MinimumFlows ;
double* MaximumFlows ;

GetVariableInfo ("MaterialFlow",  IntSeq(1,1), IntSeq(4,4),
                 CurrentFlows, MinimumFlows, MaximumFlows) ;
```

4.4.2 Method `GetConstraintInfo`

Declaration: `bool GetConstraintInfo(string cname, si* dimLB, si* dimUB, double* LB, double* UB, double* LagMult)`

Function: Returns the current values, lower and upper bounds and Lagrange multipliers of all elements of a specified slice of a specified constraint set; the return value is `true` if every element of the constraint slice is linear, or `false` otherwise.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`cname`	`string`	name of constraint set on which information is required
`dimLB`	`si*`	lower bounds of the constraint set slice on which information is required
`dimUB`	`si*`	upper bounds of the constraint set slice on which information is required

Arguments returned to client:

Argument	Type	Value on Exit
`LB`	`double*`	pointer to set of real numbers containing the upper bounds of elements in specified slice
`UB`	`double*`	pointer to set of real numbers containing the upper bounds of elements in specified slice
`LagMult`	`double*`	pointer to set of real numbers containing the Lagrange multipliers of elements in specified slice
return value	`bool`	true if constraint is linear

Notes:

- The return value is `true` if every element of the constraint is linear and `false` otherwise.
- The specified constraint set must have already been created using the `NewConstraint` method (see section 4.2.3).
- In the case of multidimensional constraint sets, the information in the sets `LB`, `UB` and `LagMult` is ordered so that the last index varies fastest, followed by the penultimate index, and so on until the first index which varies the most slowly. For example, a 2-dimensional set is ordered by rows.
- If the Lagrange multiplier for a element of the specified slice is not

available (*e.g.* because the MINLP has not yet been solved or because the MINLP solver does not make this information available), the a value of `PlusInfinity` (*cf.* section 3.1) will be returned for the corresponding element of `LagMult`.

Examples of usage:

The following returns information on all elements of the constraint `SourceFlowLimit` (*cf.* example in section 4.2.3):

```
char*   CurrentType ;
double* CurrentUB  ;
double* CurrentLM   ;
bool isLinear    ;

isLinear = GetConstraintInfo("SourceFlowLimit",
                             IntSeq(0), IntSeq(4),
                             CurrentType, CurrentUB, CurrentLM);
```

4.4.3 Method `GetObjectiveFunctionInfo`

Declaration: `bool GetObjectiveFunctionInfo(string oname, char* otype, double& ovalue)`

Function: Returns the current value and type of the objective function specified by name; the "return value" is `true` if the constraint slice has an expression set to it, `false` otherwise.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`oname`	`string`	name of objective function on which information is required

Arguments returned to client:

Argument	Type	Value on Exit
`otype`	`char*`	pointer to set of characters containing the type of the objective function
`ovalue`	`double&`	value of the objective function
return value	`bool`	true if objective function is linear

Notes:

- The return value is `false` if the objective function has a nonlinear expression assigned to it and `true` otherwise.
- The specified objective function must have already been created using the `NewObjectiveFunction` method (see section 4.2.11).
- The returned type of the objective function could be either `"min"` or `"max"`, denoting minimisation and maximisation respectively.
- The value of the objective function returned is based on the current values of the variables.

Examples of usage:

The following returns information on the objective function named `Total-Profit` (*cf.* example in section 4.2.11):

```
char*   CurrentObjType ;
double  CurrentObjValue  ;
bool    isLinear;

isLinear = GetObjectiveFunctionInfo ("TotalProfit", CurrentObjType,
                                     CurrentObjValue) ;
```

4.4.4 Method `GetProblemInfo`

Declaration: `GetProblemInfo(string pname, bool& islinear, double& ovalue, bool& isfeasible)`

Function: Returns the problem name, whether the problem is a MILP or a MINLP, the current recorded value of the objective function (as set by the last solver module that tried to solve the problem), and whether the problem is feasible or not (with respect to the current values of problem variables).

Arguments to be specified by the client: None.

Arguments returned to client:

Argument	Type	Value on Exit
`pname`	`string`	name of the problem
`islinear`	`bool&`	`true` if problem is a MILP, `false` if it is a MINLP
`ovalue`	`double&`	value of the objective function
`isfeasible`	`bool&`	`true` if problem is feasible, `false` if it is not

Notes:

- The variable `ovalue` is *not* the value of the objective function calculated at the current variable values, but rather the value set by the solver module that last tried to solve this problem.

Examples of usage:

The following returns information on the problem.

```
string ProblemName;
bool MILP;
double ObjFunValue;
bool Feasible;

NetOpt->GetProblemInfo(ProblemName, MILP, ObjFunValue, Feasible);
```

4.5 Flat MINLP Information Access Methods

Although it is convenient for client programs to construct `ops` objects in a structured manner using the methods of section 4.2, most existing numerical solvers are designed to operate on the much simpler "flat" form [**Flat MINLP**] described in section 1.

In view of the above, the `ops` interface provides a set of methods that allows access to the information characterising this flat representation. The latter is constructed automatically and efficiently by *ooOPS* in a manner that is transparent to the client.

4.5.1 General Properties of the Flat MINLP

The flat MINLP generated by *ooOPS* has the following characteristics:

- The constraints in the flat MINLP comprise those elements of the constraint sets in the `ops` object that fulfil both of the following criteria:
  - they have at least one variable occurring linearly with a non-zero coefficient, or a nonlinear part occurrence;
  - at least one of the bounds is different from the respective infinity constant.

- The variables in the flat MINLP comprise those elements of the variable sets in the `ops` object that appear with a non-zero coefficient in at least one of the constraints and/or in the objective function in the flat MINLP (see above), or in the nonlinear parts of at least one constraint and/or the objective function.

4.5.2 Method `GetFlatMINLPSize`

Declaration: `void GetFlatMINLPSize(int& nv, int& niv, int& nlv, int& nliv, int& nc, int& nlc, int& nlz, int& nnz, int& nlzof, int& nnzof)`

Function: Returns information on the size of the flat MINLP.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`nv`	`int`	total number of variables in flat MINLP
`niv`	`int`	number of integer variables in flat MINLP
`nlv`	`int`	number of variables in flat MINLP which only appear linearly
`nliv`	`int`	number of integer variables in flat MINLP which only appear linearly
`nc`	`int`	total number of constraints in flat MINLP
`nlc`	`int`	number of linear constraints in flat MINLP
`nlz`	`int`	number of non-zero elements in the matrix of flat MINLP (A in eqn. (2))
`nnz`	`int`	number of non-zero elements in the Jacobian matrix of the nonlinear constraints (f in eqn. (2))
`nlzof`	`int`	number of variables having non-zero coefficients in the linear part of the objective function of flat MINLP
`nnzof`	`int`	number of non-zero first order derivatives in the objective function of flat MINLP

Notes:

- The numbers of variables and constraints in the flat MINLP are determined using the rules detailed in section 4.5.1.
- The number of variables `nv` includes both continuous and integer variables.

Examples of usage:

The following returns information on the size of a flat MINLP described by an existing `ops` object called `NetOpt`:

```
int NumberOfVariables                ;
```

```
int NumberOfIntegerVariables          ;
int NumberOfLinearVariables           ;
int NumberOfLinearIntegerVars         ;
int NumberOfConstraints               ;
int NumberOfLinearConstraints         ;
int NumberOfNZLinVarsInConstraints    ;
int NumberOfNZNonLinJacInConstraints  ;
int NumberOfNZLinVarsInObjFun         ;
int NumberOfNZNonLinJacInObjFun       ;

NetOpt->GetFlatMINLPSize(&NumberOfVariables,
                         &NumberOfIntegerVariables,
                         &NumberOfLinearVariables,
                         &NumberOfLinearIntegerVariables,
                         &NumberOfConstraints,
                         &NumberOfLinearConstraints,
                         &NumberOfNZLinVarsInConstraints,
                         &NumberOfNZNonLinJacInConstraints,
                         &NumberOfNZLinVarsInObjFun,
                         &NumberOfNZNonLinJacInObjFun);
```

4.5.3 Method `GetFlatMINLPStructure`

Declaration: `void GetFlatMINLPStructure(int* rowindex, int* columnindex, int* objindex, string structuretype)`

Function: Returns information on the sparsity structure of the objective function and the constraints. This corresponds to one of the following, depending on the request issued by the client:

1. the linear variable occurrences;
2. the jacobian elements occurrences;
3. the union of the preceding structures.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`structuretype`	`string`	specifies whether returned structure should be of type (1), (2) or (3) (see above)

Arguments returned to client:

Argument	Type	Value on Exit
`rowindex`	`int*`	pointer to set of integers containing the numbers of the constraints in the flat MINLP from which the nonzero elements originate
`columnindex`	`int*`	pointer to set of integers containing the numbers of the variables in the flat MINLP from which the nonzero elements in constraints originate
`objindex`	`int*`	pointer to set of integers containing the numbers of the variables in the flat MINLP from which the nonzero elements in the objective function originate

Notes:

- The input parameter `structuretype` must be one of the following strings: `"LINEAR", "NONLINEAR", "BOTH"` depending on whether the client needs the linear structure, the nonlinear structure or a union of both.
- The integer sets pointed at by `rowindex` and `columnindex` are both of length `nlz, nnz` or `nlz + nnz` (see section 4.5.2) depending on whether `structuretype` is `"LINEAR", "NONLINEAR"` or `"BOTH"`.

- The integer set pointed at by `objindex` is of length `nlzof, nnzof` or `nlzof + nnzof` (see section 4.5.2) according as to whether `structuretype` is `"LINEAR", "NONLINEAR"` or `"BOTH"`.
- Constraints and variables in the flat MINLP are numbered starting from 1.
- When calling with `"BOTH"` the linear and nonlinear vectors may be overlapping (i.e. there may be indices $i < j$ such that `rowindex[i]` = `rowindex[j]` and `columnindex[i]` = `columnindex[j]`, but the actual derivatives refer respectively to linear and nonlinear entries). For example, if the first flat problem constraint is $0 \leq x_1^2 + 3x_1 \leq 0$ then the linear derivatives matrix entry $(1, 1)$ is 3, and the nonlinear derivatives matrix entry $(1, 1)$ is $\frac{\partial x_1^2}{\partial x_1}$ evaluated at the current value of x_1. Thus the union of linear and nonlinear problem structure contains two entries for position $(1, 1)$, but the first refers to the linear derivative and the second refers to the nonlinear derivative.
- The same is true for the objective function structure.

Examples of usage:

The following returns the linear structure of a flat MINLP described by an existing `ops` object called `NetOpt`:

```
int* Rows          ;
int* Columns       ;
int* ObjFunColumns ;

NetOpt->GetFlatMINLPStructure(Rows, Columns,
                              ObjFunColumns, "LINEAR");
```

4.5.4 Method `GetFlatMINLPVariableInfo`

Declaration: `void GetFlatMINLPVariableInfo(int vid, string& vname, si* &index, bool& isinteger, bool& islinear, double& value, double& LB, double& UB)`

Function: Returns information pertaining to a variable in the flat MINLP.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vid`	`int`	the number of the variable in the flat MINLP structure

Arguments returned to client:

Argument	Type	Value on Exit
`&vname`	`string`	the name of the variable set from which this variable originates
`&index`	`si*`	the index of the variable in the variable set from which it originates
`&isinteger`	`bool`	true if the variable is of type integer false otherwise
`&islinear`	`bool`	true if the variable only appears linearly in the problem
`&value`	`double`	the current value of the variable
`&LB`	`double`	the lower bound of the variable
`&UB`	`double`	the upper bound of the variable

Notes:

- The variable number `vid` specified must be in the range 1,.., `nv` (see section 4.5.2).
- Index holds a pointer to a sequence which is part of the MINLP's internal data and must not be altered in any way.

Examples of usage:

The following returns information on variable 375 in a flat MINLP described by an existing `ops` object called `NetOpt`:

```
string vname      ;
si*    index      ;
bool   isinteger  ;
bool   islinear   ;
double value      ;
double LB         ;
double UB         ;
```

```
NetOpt->GetFlatMINLPVariableInfo(375, vname, index, isinteger,
                                 islinear, value, LB, UB);
```

On return from `GetFlatMINLPVariableInfo`, variable `vname` could have the value `MaterialFlow` (*cf.* example in section 4.2.1), `index` could be `(2, 4)`, *isinteger* could be `false`, and `value`, `LB` and `UB` could be 1, 0 and 100 respectively. Thus, we can deduce that variable 375 in the flat MINLP originated from the continuous variable `MaterialFlow(2,4)` in the original (structured) MINLP.

4.5.5 Method `SetFlatMINLPVariableBounds`

Declaration: `void SetFlatMINLPVariableBounds(double* LB, double* UB)`

Function: Changes the lower and upper problem variable bounds.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`LB`	`double*`	set of double precision numbers which will hold the new lower bounds
`UB`	`double*`	set of double precision numbers which will hold the new upper bounds

Arguments returned to client: None

Notes:

- The sets `LB` and `UB` have size `nv` (see section 4.5.2).

Examples of usage:

The following reads the values of the problem variables described by an existing `ops` object called `NetOpt`:

```
double* vl = new double [nv];
double* vu = new double [nv];
for(int i = 0; i < nv; i++) {
  vl[i] = 0;
  vl[i] = 1;
}
NetOpt->SetFlatMINLPVariableBounds(vl, vu);
```

4.5.6 Method `GetFlatMINLPVariableValues`

Declaration: `void GetFlatMINLPVariableValues(double* values)`

Function: Fills the set of double precision numbers passed to the function with the current values of the flat MINLP problem variables.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`values`	`double*`	set of double precision numbers which will hold the variable values

Arguments returned to client: None

Notes:

- The set `values` has size `nv` (see section 4.5.2).

Examples of usage:

The following sets the bounds of the problem variables described by an existing `ops` object called `NetOpt`:

```
double* v = new double [nv];
NetOpt->GetFlatMINLPVariableValues(v);
```

4.5.7 Method `SetFlatMINLPVariableValues`

Declaration: `void SetFlatMINLPVariableValues(double* values)`

Function: Sets the values of the flat MINLP variables.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`values`	`double*`	set of double precision numbers holding the variable values to be set

Arguments returned to client: None

Notes:

- The set `values` has size `nv` (see section 4.5.2).

Examples of usage:

The following sets the values of the problem variables described by an existing `ops` object called `NetOpt`:

```
double* vv = new double [nv];
for(int i = 0; i < nv; i++)
    vv[i] = i / 2;
NetOpt->SetFlatMINLPVariableValues(vv);
```

4.5.8 Method `GetFlatMINLPConstraintInfo`

Declaration: `void GetFlatMINLPConstraintInfo(int cid, string& cname, si* &index, double& LB, double& UB, si* &vlist, sd* &cflist, FlatExpression* &fe)`

Function: Returns information pertaining to a constraint in the flat MINLP.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`cid`	`int`	the number of the constraint in the flat MINLP structure

Arguments returned to client:

Argument	Type	Value on Exit
`&cname`	`string`	the name of the constraint set from which this constraint originates
`&index`	`si*`	the index of the constraint in the constraint set from which it originates
`&LB`	`double`	the current constraint lower bound
`&UB`	`double`	the current constraint upper bound
`&vlist`	`si*`	the vector of integers which contains the variable indices occuring in this constraint
`&cflist`	`sd*`	the vector of doubles which contains the coefficients of the variables occuring in this constraint
`&fe`	`FlatExpression*`	an object which contains the symbolic information which defines the nonlinear part of the constraint (see section 6.2)

Notes:

- The variable number `cid` specified must be in the range 1,..,`nc` (see section 4.5.2).
- Index holds a pointer to a sequence which is part of the MINLP's internal data and must not be altered in any way.
- The argument `FlatExpression* &fe` contains the symbolic information which defines the nonlinear part of the constraint. For its description and usage see section 6.2.

Examples of usage: The following returns information on constraint 532 in a flat MINLP described by an existing `ops` object called `NetOpt`:

```
string          cname   ;
```

```
si*              index   ;
double           LB      ;
double           UB      ;
si*              varList ;
sd*              coefList;
FlatExpression* fe;

NetOpt->GetFlatMINLPConstraintInfo(532, cname, index, LB, UB,
                                   varList, coefList, fe)
```

On return from `GetFlatMINLPConstraintInfo`, variable `cname` could have the value `"SourceFlowLimit"` (*cf.* example in section 4.2.4), `index` could be `(3)`, and `UB` could be 450. Thus, we can deduce that constraint 532 in the flat MINLP originated from the constraint `SourceFlowLimit(3)` in the original (structured) MINLP.

4.5.9 Method `EvalFlatMINLPNonlinearObjectiveFunction`

Declaration: `double EvalFlatMINLPNonlinearObjectiveFunction(void)`

Function: Returns the value of the nonlinear part of the objective function.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`double`	the value of the objective function at the current variable values

Notes: None

Examples of usage:

The following sets the variable values and evaluates the objective function of the problem described by an existing `ops` object called `NetOpt`:

```
double* vv = new double [nv];
for(int i = 0; i < nv; i++)
    vv[i] = i / 2;
NetOpt->SetFlagMINLPVariableValues(vv);
double vof = NetOpt->EvalFlatMINLPNonlinearObjectiveFunction();
```

4.5.10 Method `EvalFlatMINLPNonlinearConstraint`

Declaration: `void EvalFlatMINLPNonlinearConstraint(int lowercid, int uppercid, double* values)`

Function: Returns the values of a range of nonlinear parts of constraints, starting with constraint `lowercid` up to and including constraint `uppercid` (also see (4.5.8)).

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`lowercid`	`int`	the number of the first constraint of the range to be evaluated
`uppercid`	`int`	the number of the last constraint of the range to be evaluated

Arguments returned to client:

Argument	Type	Value on Exit
`values`	`double*`	vector containing the constraint values

Notes:

- The vector `values` containing the values of the evaluated constraints, has size `uppercid - lowercid + 1`.

Examples of usage:

The following evaluates constraints 2-5 in the flat MINLP described by an existing `ops` object called `NetOpt`:

```
double* cv = new double[4];
NetOpt->EvalFlatMINLPNonlinearConstraint(cv, 2, 5);
```

4.5.11 Method `GetFlatMINLPObjectiveFunctionDerivatives`

Declaration: `void GetFlatMINLPObjectiveFunctionDerivatives (double* A, string structuretype)`

Function: Returns a vector containing one of the following:

1. the values of the nonzero coefficients in the vector c defining the linear part of the objective function (*cf.* equation (1));
2. the values of the nonzero first order partial derivatives of the nonlinear part of the objective function evaluated at the current variable values;
3. the union of the preceding vectors.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`structuretype`	`string`	specifies whether returned structure should be of type (1), (2) or (3) (see above)

Arguments returned to client:

Argument	Type	Value on Exit
`values`	`double*`	pointer to set of doubles containing the nonzero elements of c, of the derivatives of the obj. fun., or both

Notes:

- The input parameter `structuretype` must be one of the following strings: `"LINEAR", "NONLINEAR", "BOTH"` depending on whether the client needs information about the linear part of the objective function, or the nonlinear part, or both.
- The length of this vector follows the rules given in note 3 to section 4.5.3.
- The indices of the variables to which the elements of this vector correspond can be obtained from method `GetFlatMINLPStructure` as the integer vector `objindex` (see section 4.5.3).
- When calling with `"BOTH"`, see notes on page 50.

Examples of usage:

The following returns the nonzero partial derivatives of the nonlinear part of the objective function of a flat MINLP described by an existing `ops` object called `NetOpt`, evaluated at the current variable values:

```
int d0, d1, d2, d3, d4, d5, d6, d7, d8;
int nnzof;
NetOpt->GetFlatMINLPSize(&d0, &d1, &2, &d3, &d4,
                         &d5, &d6, &d7, &8, &d9,  &nnzof);
double* A = new double [nnzof];
NetOpt->GetFlatMINLPObjectiveFunctionDerivatives(A, "NONLINEAR");
```

In this case, $c(k)$, $k = 1$,..,`nlzof` is the coefficient of variable `objindex(k)` in the objective function.

4.5.12 Method `GetFlatMINLPConstraintDerivatives`

Declaration: `void GetFlatMINLPConstraintDerivatives(string structuretype, int lowercid, int uppercid, sd& values)`

Function: Returns a sequence of doubles (see section 3.2) containing one of the following:

1. the values of the nonzero coefficient in the linear part of the specified constraints;
2. the values of the nonzero partial derivatives, evaluated at the current variable values, of the nonlinear parts of the specified constraints;
3. the union of the preceding vectors.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`structuretype`	`string`	specifies whether returned structure should be of type (1), (2) or (3) (see above)
`lowercid`	`int`	specifies the lower end of the constraint range
`uppercid`	`int`	specifies the upper end of the constraint range

Arguments returned to client:

Argument	Type	Value on Exit
`values`	`sd&`	sequence of doubles containing the nonzero elements of A, of the Jacobian of f, or both

Notes:

- The input parameter `structuretype` must be one of the following strings: `"LINEAR", "NONLINEAR", "BOTH"` according as to whether the client needs the linear part of the constraints, the derivatives, or a union of both.
- The length of this vector is given by `values.size()`.
- If `lowercid` and `uppercid` are both set to zero, then this function returns the nonzero coefficients of the entire matrix A (see equation (2)), or the nonzero coefficients of the entire Jacobian of f (see equation (2)) evaluated at the current variable values, or both, depending on `structuretype`. In this case this method effectively acts as though the whole range of problem constraints had been specified.

- The row and column indices of the elements of the vector can be obtained from method `GetFlatMINLPStructure` as integer vectors `rowindex` and `columnindex` respectively (see section 4.5.3).
- Nonlinear derivatives which are identically zero are not recorded. Thus, for example, if you have a linear constraint and you request its nonlinear derivatives, the vector `values` might be empty. Referring to elements of an empty vector results in runtime segmentation fault errors, so it is advisable to check `values.size()` before using the vector.
- When calling with `"BOTH"`, see notes on page 50.

Examples of usage:

The following returns the matrix A of a flat MINLP described by an existing `ops` object called `NetOpt`:

```
sd A;
NetOpt->GetFlatMINLPConstraintDerivatives(A, 0, 0, "LINEAR");
```

In this case, $A(k)$, k =1,..,`nlz` is the coefficient of variable `columnindex(k)` in constraint `rowindex(k)` in the left hand side matrix A (*cf.* equation (2)).

4.5.13 Method `GetFlatMINLPNoPartitions`

Declaration: `void GetFlatMINLPNoPartitions(int& np)`

Function: Returns the number of partitions of the flat MINLP.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`np`	`int`	total number of partitions of flat MINLP

Notes:

- The number of partitions are determimed based on the properties of key variables.

Examples of usage:

The following returns the number of partitions occuring in a flat MINLP described by an existing `ops` object called `NetOpt`:

```
int  np;

NetOpt->GetFlatMINLPNoPartitions(np) ;
```

4.5.14 Method `GetFlatMINLPPartition`

Declaration: `void GetFlatMINLPPartition(int& np, li* &varlist, li* &conlist)`

Function: Returns information on the partitions of the flat MINLP specified by its number.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`np`	`int`	the number of the partition of the flat MINLP

Arguments returned to client:

Argument	Type	Value on Exit
`varlist`	`li`	list of variable indices occuring in the partition
`conlist`	`li`	list of constraint indeces occuring in the partition

Notes:

- The number np must be less or equal to the maximum number of partitions.

Examples of usage:

The following returns the variable and constraint lists of the partition 2 occuring in a flat MINLP described by an existing `ops` object called `NetOpt`:

```
int  np;
li*  variableList;
li*  constraintList;

NetOpt->GetFlatMINLPPartition(np, variableList, constraintList) ;
```

4.6 Standard Form MINLP Information Access Methods

A MINLP is in *standard form* when its nonlinear parts are reduced to their basic building blocks and all its linear parts are gathered together in a matrix. This is explained in more details in [SP99]. Suffice it here to recall the basics with an example. The constraint

$$-1 \leq 4x_1 + 3x_2 - x_3 + \frac{x_1 x_2}{\log(x_3)} \leq 10$$

in standard form becomes

$$\begin{array}{rll} -1 \leq & w_1 & \leq 10 \\ & w_1 & = 4x_1 + 3x_2 - x_3 + w_2 \\ & w_2 & = \dfrac{w_3}{w_4} \\ & w_3 & = x_1 x_2 \\ & w_4 & = \log(x_3) \end{array}$$

The standard form of a MINLP is as follows:

$$\begin{array}{rcl} \min_x & x_i & \\ l \leq & Ax & \leq u \\ y & = & w \otimes z \\ x^L \leq & x & \leq x^U \end{array} \tag{6}$$

where y, w, z are subsets of $x \cup \mathbb{R}$ and $\otimes$ is any unary or binary operator.

In view of the above, the `ops` interface provides a set of methods that allows access to the information characterising this standard form representation. The latter is constructed automatically and efficiently by *ooOPS* in a manner that is transparent to the client.

4.6.1 Method `GetSFNumberOfVariables`

Declaration: `void GetSFNumberOfVariables(int& nv, int& nzlv)`

Function: It returns the total number of variables of the problem in standard form and the number of linearly appearing variables with nonzero coefficients in the linear part of the problem.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value On Exit
`nv`	`int&`	the number of total problem variables
`nzlv`	`int&`	number of nonzero coeff. linear variables

Notes:

- The total number of problem variables includes the original problem variables and the variables which have been added by the standard form process.

Examples of usage:

```
int  nv;
int  nzlv;
NetOpt->GetSFNumberOfVariables(nv, nzlv) ;
```

4.6.2 Method `GetSFVariableInfo`

Declaration: `void GetSFVariableInfo(int vid, double value, double& LB, double& UB)`

Function: It returns the current variable value and the lower and upper bounds of variable `vid` in the problem in standard form.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`vid`	`int`	variable id in the standard form problem

Arguments returned to client:

Argument	Type	Value On Exit
`value`	`double&`	current variable value
`LB`	`double&`	variable lower bound
`UB`	`double&`	variable upper bound

Notes: None

Examples of usage:

```
double value, LB, UB;
NetOpt->GetSFVariableInfo(1, value, LB, UB) ;
```

4.6.3 Method `GetSFObjFunVarIndex`

Declaration: `void GetSFObjFunVarIndex(int& vid)`

Function: It returns the variable index (variable id) corresponding to the objective function.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`vid`	`int&`	variable index corresponding to the objective function

Notes:

- As well as the constraints, the objective function of the MINLP is transformed by the standard form reduction, so that, for example, $\min x_1 x_2$ would become $\min w_1$ s.t. $w_1 = x_1 x_2$. In this case, the variable index of w_1 would be returned.

Examples of usage:

```
int  vid;
NetOpt->GetSFObjFunVarIndex(vid);
```

4.6.4 Method `GetSFNumberOfLinearConstraints`

Declaration: `void GetSFNumberOfLinearConstraints(int& nlc)`

Function: This returns the number of linear constraints in the standard form of the problem.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`nlc`	`int&`	number of linear constraints

Notes: None

- The number of linear constraints is equal to the number of rows in the matrix A in the general formulation of the standard form (see eqn. 6 above).

Examples of usage:

```
int  nlc;
NetOpt->GetSFNumberOfLinearConstraints(nlc);
```

4.6.5 Method `GetSFLinearBounds`

Declaration: `void GetSFLinearBounds(double* lb, double* ub)`

Function: It returns the vector of lower and upper bounds in the linear constraints of the standard form problem (l and u in the formulation 6).

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`lb`	`double*`	vector of lower bounds
`ub`	`double*`	vector of upper bounds

Notes:

- The arrays `lb` and `ub` must be created by the client with the correct length `nlc` (use method `GetSFNumberOfLinearConstraints` above).

Examples of usage:

```
int     nlc;
NetOpt->GetSFNumberOfLinearConstraints(nlc);
double* lb = new double[nlc];
double* ub = new double[nlc];
NetOpt->GetSFLinearBounds(lb, ub);
```

4.6.6 Method `GetSFLinearStructure`

Declaration: `void GetSFLinearStructure(int* rowindex, int* columnindex)`

Function: Returns the sparsity structure of the linear part of the problem in standard form.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`rowindex`	`int*`	vector of row indices
`columnindex`	`int*`	vector of column indices

Notes:

- The arrays `rowindex` and `columnindex` must be created by the client with the correct length `nzlv` (use method `GetSFNumberOfVariables` above).

Examples of usage:

```
int  nv;
int  nzlv;
NetOpt->GetSFNumberOfVariables(nv, nzlv) ;
int* rowindex = new int [nzlv];
int* columnindex = new int [nzlv];
NetOpt->GetSFLinearStructure(rowindex, columnindex);
```

4.6.7 Method `GetSFMatrix`

Declaration: `void GetSFMatrix(double* A)`

Function: Returns the linear part of the standard form problem in sparse format.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`A`	`double*`	matrix A in sparse form

Notes:

- The array `A` must be created by the client with the correct length `nlzv` (use method `GetSFNumberOfVariables` above).
- The sparsity structure can be found with the method `GetSFLinear-Structure` (4.6.6).

Examples of usage:

```
int  nv;
int  nzlv;
NetOpt->GetSFNumberOfVariables(nv, nzlv) ;
double* A = new double [nzlv];
NetOpt->GetSFMatrix(A);
```

4.6.8 Method `GetSFNumberOfNonlinearConstraints`

Declaration: `void GetSFNumberOfNonlinearConstraints(int& nnlc)`

Function: This returns the number of nonlinear constraints in the standard form of the problem.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`nnlc`	`int&`	number of nonlinear constraints

Notes: None

- The number of nonlinear constraints is equal to the number of "constraint definitions" of the form $y = w \otimes z$ in the general formulation of the standard form (see eqn. 6 above).

Examples of usage:

```
int  nnlc;
NetOpt->GetSFNumberOfNonlinearConstraints(nnlc);
```

4.6.9 Method `GetSFNonlinearConstraint`

Declaration: `void GetSFNonlinearConstraint(int cid, int& vid, int& vid1, int& vid2, string& operator, double& constant1, double& constant2`

Function: This function returns the elements of nonlinear standardized constraint number `cid` in the standard form of the problem.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`cid`	`int`	standardized nonlinear constraint id

Arguments returned to client:

Argument	Type	Value on Exit
`vid`	`int&`	left hand side variable id
`vid1`	`int&`	first right hand side variable id
`vid2`	`int&`	second right hand side variable id
`operator`	`string&`	operator type
`constant1`	`double&`	first left hand side constant
`constant2`	`double&`	second left hand side constant

Notes:

- The standard nonlinear constraint ID (`cid`) always starts from 1.
- In this discussion, we assume that each standard form nonlinear constraint has the form

  $$\textit{variable} = \textit{operand1} \otimes \textit{operand2}$$

  where *variable* is the "added variable" that is defined by the right hand side and the operator $\otimes$ is either unary or binary (if it is unary, then *operand2* is a dummy placeholder).
- `vid` is the variable id of *variable*.
- `vid1` is the variable id of *operand1* if the latter is a variable (e.g. $w_1 = x_1 x_2$). If *operand1* is a constant then `vid1` is set to -1 (e.g. $w_2 = \frac{2}{x_3}$).
- `vid2` is the the variable id of *operand2* if the latter is a variable and if $\otimes$ is a binary operator. If *operand2* is a constant (e.g. $w_3 = x_4^2$) or if $\otimes$ is unary then `vid2` is set to -1.
- `operator` represents the type of operator. It can be one of the following strings (the meaning is self-explanatory): `"sum"`, `"difference"`, `"product"`, `"ratio"`, `"power"`, `"minus"`, `"log"`, `"exp"`, `"sin"`, `"cos"`, `"tan"`, `"cot"`, `"sinh"`, `"cosh"`, `"tanh"`, `"coth"` (cf. section 2.4).

- `constant1` is meaningful only when `vid1` is set to -1.
- `constant2` is meaningful only when `vid2` is set to -1 and $\otimes$ is a binary operator.
- Note that in a problem in standard form the constraint id `cid` only applies to nonlinear standardized constraints and does not take into account linear constraints.

Examples of usage:

```
int     cid = 1;
int     vid;
int     vid1;
int     vid2;
string  operat;
double  constant1;
double  constant2;
NetOpt->GetSFNonlinearConstraint(cid, vid, vid1, vid2,
                                 operat, constant1, constant2);
```

4.6.10 Method `UpdateSolution`

Declaration: `void UpdateSolution(double* sol)`

Function: This method is specifically designed to make it easy to insert the solution of the MINLP into both the structured and the flat form. In short, this method updates the variable values in both the structured and flat MINLP forms.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`sol`	`double`	array of (flat) variable values

Arguments returned to client: None

Notes:

- The length of the array `sol` has to be at least `nv`, the number of variables in flat form (see section 4.5.2).
- This method has been specifically designed to make it easy to update the solution in the MINLP at the end of a MINLP solver module, and is therefore targeted towards those programmers who wish to write their own solver module.

Examples of usage:

```
double  sol[nv];
NetOpt->UpdateSolution(&(sol[0]));
```

5 MINLP Solvers and Systems

5.1 Introduction

Section 4 of this document described in detail how `ops` objects can be constructed and modified, and how information in them can be accessed in both a structured and a flat form. This section is concerned with the solution of the mathematical problem described by an already existing `ops` object.

5.1.1 MINLP Solver Managers and MINLP Systems

The solution of an MINLP is normally effected by a numerical solver. There are commercial solvers (*e.g.* SNOPT) as well as non-commercial ones (*e.g.* DONLP2). A major objective for the *ooOPS* software is to provide application programs with a *uniform* interface to all such solvers. This is achieved by embedding each solver within a `opssolvermanager` object.

The main function of the Manager for a given MINLP Solver is the creation of `opssystem` objects (*cf.* section 2.1) by combining an existing `ops` object with the numerical solver embedded within the Manager. It is this combination that ultimately permits the solution of the MINLP to take place.

5.1.2 Algorithmic Parameters for MINLP Solvers

MINLP solvers of the kind of interest to *ooOPS* are sophisticated pieces of software. Although the basic algorithms implemented by different solvers are often very similar, the specific implementations may be significantly different. Moreover, the users of these solvers are normally provided with substantial flexibility in configuring the details of the behaviour of the implementation. This is typically achieved by setting the values of one or more *algorithmic parameters*. These are usually quantities of logical, integer, real or string type. Different MINLP solvers may recognise different sets of parameters; some typical examples include:

- The branching strategy to be used by branch-and-bound algorithms (*e.g.* depth-first, breadth-first *etc.*).
- The maximum number of nodes to be examined during the branch-and-bound search.
- The maximum CPU time to be spent by the solution.
- The infeasibility tolerance within which constraints need to be satisfied.

Usually, MINLP solvers also incorporate a default value for each parameter. Although these values may lead to reasonably good performance for a wide range of applications, sophisticated users may wish to change them to suit the specific characteristics of particular applications.

ooOPS provides general mechanisms for handling algorithmic parameters that allows the client program (a) to determine the parameters a particular MINLP solver recognises, and their current values, and (b) to specify new values for one or more of these parameters. Parameter specification can operate at two levels:

- *At the Solver Manager level*:
  Specifying the value of a parameter in a `MINLPSolverManager` object ensures that any `opssystem` objects *subsequently* created from this `MINLPSolverManager` will, at least initially, have this value of the parameter.
- *At the MINLP System level*:
  Specifying the value of a parameter in a `opssystem` object affects this particular object only.

5.2 The `ssolpar` and `sstat` Argument Types

In order to handle the passing of these different kinds of parameters, *ooOPS* introduces the following C++ type definitions:

- `variant`: a union containing a value which may be one of several different types
- `ssolpar`: a sequence of solver parameters
- `sstat`: a sequence of solution statistics

The following C++ type definitions, which are included in the `ops.h` header file supplied to the user describe these types fully:

```
enum vtype {logical, integer, real, expression};

struct variant {
        vtype thetype;
        union val {
          int    ival;
          double  dval;
          string* sval;
```

```
          bool    bval;
        };
};

struct solparameter {
  variant theval;
  string  name;
  string  description;
  double  lowerbound;
  string  upperbound;
};

typedef vector<solparameter> ssolpar;

struct statistic {
  variant theval;
  string  description;
  string  unit;
};

typedef vector<statistic> sstat;
```

5.3 MINLP Solver Manager Instantiation: The `NewMINLPSolverManager` Function

Declaration: `opssolvermanager* (*NewMINLPSolverManager)(void)`

Function: Creates a new `MINLPSolverManager` object incorporating a specified numerical code.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`opssolvermanager*`	the ops solver manager incorporating the MINLP

Notes:

- This function is not available at compile time (hence it is only a pointer to a function). It is loaded from a shared object library at run time using the `dlopen()/dlsym()` mechanism. See example below for details.
- When using run time linking, keep in mind that search paths for shared object files vary from operating system to operating system and do not usually include the current working directory.
- Please be warned that shared object files produced from C++ source code have "mangled" symbol named which usually `dlopen` and `dlsym` cannot read properly. There are two solutions: first, write wrapper functions to `dlopen` and `dlsym` which take care of this problem; and second, use non-demangled symbol names (as in the example below).
- When writing a new solver manager for a particular solver code, the new solver manager has to expose the following function in the global namespace:

  ```
  opssolvermanager* NewMINLPSolverManager(void) {
    // ... code
    return new opssolvermanager_i();
  }
  ```

  Please use the provided template source files for the creation of new solver managers.

Examples of usage:

The following creates a new MINLP solver manager called `MySNOPTSolverManager` incorporating the SNOPT numerical MINLP solver:

```
#include <dcfcn.h>
opssolvermanager* (*NewMINLPSolverManager)(void);
opssolvermanager* MySNOPTSolverManager;
void* handle = dlopen("libopssnopt.so", RTLD_LAZY);
if (!handle) {
  cerr << "MAIN: shared object error: \n\t" << dlerror() << endl;
  exit(-1);
} else {
  // have to use "non-demangled" C++ symbol names
  void *tmp = dlsym(handle, "NewMINLPSolverManager__Fv");
  char* error;
  if ((error = dlerror()) != NULL) {
    cerr << "MAIN: shared object error: \n\t" << error << endl;
    exit(-1);
  }
  NewMINLPSolverManager = (opssolvermanager* (*)()) (tmp);
}
opssolvermanager* MySNOPTSolverManager = (*NewMINLPSolverManager)();
// ... code
dlclose(handle);
```

This manager can now be used to create one or more MINLP systems, each incorporating a separate `ops` object (see section 5.4.3 below).

5.4 MINLP Solver Managers

5.4.1 Method `GetParameterList`

Declaration: `ssolpar* GetParameterList()`

Function: Gets the list of parameters with which a MINLPSystem can be configured. It returns a sequence of structures holding the current values of the parameters, their (single word) names and short descriptions, and valid upper and lower bounds where applicable (the values `MinusInfinity` and `PlusInfinity` will be used to indicate unconstrained parameters).

The section 5.2 for the detailed description of this type.

Arguments to be specified by the client: None

Arguments returned to client: None

Examples of usage:

The following retrieves the list of parameters for a solver:

```
ssolpar* params = SnoptManager->GetParameterList();
```

5.4.2 Method `SetParameter`

Declaration: `void SetParameter(string ParamName, variant ParamValue)`

Function: Sets a specific parameter to configure a Solver Manager. Subsequent calls to `GetParameterList` will return the value supplied, and all `MINLPSystem`s subsequently created will use the value supplied.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
ParamName	string	parameter name
ParamValue	variant	assigns a value to the parameter

Arguments returned to client: None

Examples of usage:

The following sets a parameter named "`MaxRelaxations`" to 100 in the Solver Manager `SnoptManager`:

```
variant var100;
var100.vtype=integer;
var100.val.ival=100;
SnoptManager->SetParameter("MaxRelaxations",var100) ;
```

5.4.3 Method `NewMINLPSystem`

Declaration: `opssystem* NewMINLPSystem(const ops* theops)`

Function: Creates a new `opssystem` object from a given `ops` object.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`theops`	`const ops*`	the `ops` object to be incorporated in the new `opssystem`

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`opssystem*`	the ops system incorporating the MINLP and the numerical code

Notes:

- The specified `theops` object must already exist, having been created using the `NewMINLP` function (*cf.* section 4.1 and the methods described in sections 4.2 and 4.3.

Examples of usage:

The following uses the MINLP solver manager object `SnoptManager` created in the example of section 5.3 to create a new MINLP system, called `NetOptSnopt` incorporating the `ops` object `NetOpt` created in the example of section 4.1:

```
opssystem* NetOptSnopt = SnoptManager->NewMINLPSystem(NetOpt) ;
```

5.5 MINLP Systems

5.5.1 Method `GetParameterList`

Declaration: `ssolpar* GetParameterList()`

Function: Gets the list of parameters with which a MINLPSystem can be reconfigured after creation. It returns a sequence of structures holding the current values of the parameters, their (single word) names and short descriptions, and valid upper and lower bounds where applicable (the values `MinusInfinity` and `PlusInfinity` will be used to indicate unconstrained parameters).

See section 5.2 for the detailed description of this type.

Arguments to be specified by the client: None

Arguments returned to client: None

Examples of usage:

The following retrieves the list of parameters for a system:

```
ssolpar* params=NetOptSnopt->GetParameterList();
```

5.5.2 Method `SetParameter`

Declaration: `void SetParameter(string ParamName, variant ParamValue)`

Function: Sets a specific parameter to configure an MINLPSystem.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ParamName`	`string`	parameter name
`ParamValue`	`variant`	assigns a value to the parameter

Arguments returned to client: None

Examples of usage:

The following sets a parameter named "`DiagnosticsOutputFile`" to "`mydiag.out`" in the MINLPSystem `NetOptSnopt`:

```
variant filename;
filename.vtype=string;
filename.val.sval="mydiag.out";
NetOptSnopt->SetParameter("DiagnosticsOutputFile",filename) ;
```

5.5.3 Method `GetStatistics`

Declaration: `sstat* GetStatistics()`

Function: Gets the list of solution statistics which accumulate during the lifetime of the MINLPSystem. It returns a sequence of structures holding the current values of the statistics, their short descriptions, and units.

See section 5.2 for the detailed description of this type.

Arguments to be specified by the client: None

Arguments returned to client: None

Examples of usage:

The following retrieves the statistics for a system and writes them to standard output, assuming the C++ `<<` operator has been suitably configured for the `variant` type.

```
sstat* stats = NetOptSnopt->GetStatistics();
cout << "Solution statistics:" << endl;
for(sstat::const_iterator it = stats->begin();
    it != stats->end();
    it++)
  cout << "   " << it->description << ";"
       << it->theval << it->unit << endl;
```

The output produced might be:

```
Solution statistics:
   CPU time: 233.23 seconds
   Number of relaxations: 23
```

5.5.4 Method `Solve`

Declaration: `void Solve()`

Function: Attempts to solve the MINLP problem incorporated in the `opssystem` object using the numerical MINLP solver embedded within the `opssystem` object.

Arguments to be specified by the client: None

Arguments returned to client: None

Notes:

- The numerical solution algorithm is applied to the "flat" form of the MINLP. The solver obtains the latter from the `ops` object incorporated within the `opssystem` object using the methods of section 4.5. This operation is performed automatically and is completely transparent to the client.
- The solution procedure will leave the final values of variables and other information within the `ops` object. The client may subsequently retrieve them from there using the methods of section 4.4.

Examples of usage:

The following method invocation applied to the MINLP system `NetOptSnopt` created in the example of section 5.4.3 will trigger the solution of the MINLP described by the `ops` object `NetOpt` using the SNOPT solver:

```
NetOptSnopt->Solve() ;
```

5.5.5 Method `GetSolutionStatus`

Declaration: `void GetSolutionStatus(int& status)`

Function: Returns the exit status of the solver which attempted to solve the MINLP.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Specified On Entry
`status`	`int&`	exit status

Notes:

- This method only returns meaningful information after the method `Solve()` has been called.

Examples of usage:

The following returns the exit status of the numerical solver (SNOPT) which just tried to solve the `NetOpt` problem

```
int status;
NetOptSnopt->GetSolutionStatus(status);
if (status == 0)
  cout << "Solution OK!" << endl;
```

6 Auxiliary Interfaces

6.1 The Convexification Module

This interface provides functionality for producing a convex relaxation of the MINLP. The *ooOPS* software provides a `convexifiermanager` object which embeds an existing `ops` object and provides only one public method, `Convexify()`, which returns an `ops` containing a convex flat MILP.

The "convexification" algorithm gets its input data from the MINLP in standard form (see section 4.6) and produces a convex (linear) MILP. The output convex problem is embedded in a slightly modified version of the `ops` class (see section 4) whose interface only offers flat form data access (see section 6.1.4).

6.1.1 Convexifier Manager Instantiation: the Function `NewConvexifierManager`

Declaration: `convexifiermanager* (*NewConvexifierManager) (ops* theops)`

Function: Creates a new `convexifiermanager` object incorporating the MINLP `theops`.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`theops`	`ops*`	the MINLP to be convexified

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`convexifiermanager*`	the convexifiermanager incorporating the MINLP

Notes:

- This function is not available at compile time (hence it is only a pointer to a function). It is loaded from a shared object library at run time using the `dlopen()/dlsym()` mechanism. See example below for details.
- When using run time linking, keep in mind that search paths for shared object files vary from operating system to operating system and do not usually include the current working directory.
- Please be warned that shared object files produced from C++ source code have "mangled" symbol named which usually `dlopen` and `dlsym`

cannot read properly. There are two solutions: first, write wrapper functions to `dlopen` and `dlsym` which take care of this problem; and second, use non-demangled symbol names (as in the example below).

Examples of usage: The following creates a new `convexifiermanager` incorporating the MINLP `NetOpt`.

```
#include <dcfcn.h>
convexifiermanager* (*NewConvexifierManager)(ops*);
convexifiermanager* NetOptCM;
void* handle = dlopen("libopssnopt.so", RTLD_LAZY);
if (!handle) {
  cerr << "MAIN: shared object error: \n\t" << dlerror() << endl;
  exit(-1);
} else {
  // have to use "non-demangled" C++ symbol names
  void *tmp = dlsym(handle, "NewMINLPSolverManager__FP3ops");
  char* error;
  if ((error = dlerror()) != NULL) {
    cerr << "MAIN: shared object error: \n\t" << error << endl;
    exit(-1);
  }
  NetOptCM = (convexifiermanager*) (*)(ops*)) (tmp);
}
// ... code
dlclose(handle);
```

6.1.2 Method `GetConvexMINLP`

Declaration: `ops* GetConvexMINLP(void)`

Function: Returns a convex linear relaxation of the MINLP embedded in the `convexifiermanager`.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`ops*`	the convex linear relaxation of the MINLP

Notes:

- The memory allocated by the returned convex linear problem should not be deallocated before the convexifiermanager object is deleted.

Examples of usage: The following returns a new `ops` object containing a convex linear relaxation of the MINLP.

```
ops* myconvexops = NetOptCM->GetConvexMINLP() ;
```

6.1.3 Method `UpdateConvexVarBounds`

Declaration: `void UpdateConvexVarBounds(double* lb, double* ub)`

Function: Updates the convex problem created with `GetConvexMINLP` (see 6.1.2) with new variable bounds. Because of the way the convexification is done, this has the effect of changing some of the linear structure of the convex problem.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
lb	`double*`	new lower bounds of variables in the convex problem
ub	`double*`	new upper bounds of variables in the convex problem

Arguments returned to client: None

Notes:

- The convex problem created with `GetConvexMINLP` must not be deallocated prior to the call to this function.

Examples of usage: The following updates bounds to the first convex problem variable.

```
// get convex problem
ops* myconvexops = NetOptCM->GetConvexMINLP() ;
// get convex problem size
int NumberOfVariables                ;
int NumberOfIntegerVariables         ;
int NumberOfLinearVariables          ;
int NumberOfLinearIntegerVars        ;
int NumberOfConstraints              ;
int NumberOfLinearConstraints        ;
int NumberOfNZLinVarsInConstraints   ;
int NumberOfNZNonLinJacInConstraints ;
int NumberOfNZLinVarsInObjFun        ;
int NumberOfNZNonLinJacInObjFun      ;
myconvexops->GetFlatMINLPSize(&NumberOfVariables,
                              &NumberOfIntegerVariables,
                              &NumberOfLinearVariables,
                              &NumberOfLinearIntegerVariables,
                              &NumberOfConstraints,
                              &NumberOfLinearConstraints,
                              &NumberOfNZLinVarsInConstraints,
                              &NumberOfNZNonLinJacInConstraints,
                              &NumberOfNZLinVarsInObjFun,
                              &NumberOfNZNonLinJacInObjFun);
```

```
// get convex problem variable bounds
double vlb = new double [NumberOfVariables];
double vub = new double [NumberOfVariables];
string strdummy;
si* sidummy;
bool bdummy1, bdummy2;
double ddummy;
for(int i = 1; i <= NumberOfVariables; i++) {
  myconvexops->GetFlatMINLPVariableInfo(i, strdummy, sidummy,
                                        bdummy1, bdummy2, ddummy,
                                        vlb[i - 1], vub[i - 1]);
}
// change first variable bounds
vlb[0] = vlb[0] - 1;
vub[0] = vub[0] + 1;
// update the convex problem
NetOptCM->UpdateConvexVarBounds(vlb, vub);
```

6.1.4 Methods of the Convex MILP

This is a cut-down version of the MINLP object interface (the `ops` class, see section 4) which only offers functionality for reading/writing (linear) data in the flat form problem. This modified `ops` class offers no methods for dealing with (nonlinear) information, no multidimensional construction methods and no standard form methods.

The methods provided by this interface are:

- `GetFlatMINLPSize` (see 4.5.2);

  Notes: Call is the same as in section 4.5.2.

- `GetFlatMINLPStructure` (see 4.5.3);

  Notes: Call is the same as in section 4.5.3.

- `GetFlatMINLPVariableInfo` (see 4.5.4);

  Notes: The arguments `string vname`, `si* index`, `bool isinteger`, `bool islinear` are meaningless in this context and are only kept for compatibility[3].

- `GetFlatMINLPConstraintInfo` (see 4.5.8);

  Notes: The arguments `string cname`, `si* index`, `si* cilist`, `sd* cflist`, `FlatExpression* fe` are meaningless in this context and are only kept for compatibility[4].

- `GetFlatMINLPVariableValues` (see 4.5.6);

  Notes: Call is the same as in section 4.5.6.

- `SetFlatMINLPVariableValues` (see 4.5.7);

  Notes: Call is the same as in section 4.5.7.

- `SetFlatMINLPVariableBounds` (see 4.5.5);

  Notes: Call is the same as in section 4.5.5.

- `EvalFlatMINLPNonlinearObjectiveFunction` (see 4.5.9);

  Notes: This is for compatibility only; it returns 0.

- `EvalFlatMINLPNonlinearConstraint` (see 4.5.10);

[3] They can simply be skipped in the call.
[4] They can simply be skipped in the call.

Notes: This is for compatibility only; it does not do anything.

- `GetFlatMINLPObjectiveFunctionDerivatives` — linear objective function only (see 4.5.11);

  Notes: Call is the same as in section 4.5.3.

- `GetFlatMINLPConstraintDerivatives` — linear constraints only (see 4.5.12).

  Notes: Call is the same as in section 4.5.3.

6.2 The `FlatExpression` Interface

This object class has the purpose of conveying symbolic flat expression information to the client. A `FlatExpression` object is returned by the `GetFlatMINLPConstraintInfo` method (see section 4.5.8) and is then transformed, according to its type, to any of the derived objects `FlatVariableExpression`, `FlatConstantExpression`, `FlatOperatorExpression`. Each of these objects provides a special interface used to access the encapsulated data.

The `FlatExpression` interface has only one method.

Method `GetKind`

Declaration: `int GetKind()`

Function: Returns the kind of this `FlatExpression`.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value On Exit
return value	`int`	kind of `FlatExpression`

Notes:

- The value returned by this method can be one of `FlatConstantType`, `FlatVariableType`, `FlatOperatorType` according as to whether the current `FlatExpression` is respectively one of `FlatConstantExpression`, `FlatVariableExpression`, `FlatOperatorExpression`.

Examples of usage:

See below for a comprehensive example of all the methods relative to `FlatExpression`.

6.2.1 The `FlatConstantExpression` Interface

This object class is derived from the `FlatExpression` class and provides flat symbolic information about constant expression objects created with the `NewConstantExpression` method (see section 4.2.6).

This interface has only one method.

Method `GetValue`

Declaration: `double GetValue()`

Function: Returns the value of the flat symbolic constant expression.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value On Exit
return value	`double`	value of flat symbolic constant

Notes: None

Examples of usage:

See below for a comprehensive example of all the methods relative to `FlatExpression`.

6.2.2 The `FlatVariableExpression` Interface

This object class is derived from the `FlatExpression` class and provides flat symbolic information about variable expression objects created with the `NewVariableExpression` method (see section 4.2.7).

This interface has only one method.

Method `GetVarIndex`

Declaration: `double GetVarIndex()`

Function: Returns the variable index (or variable id, a.k.a. `vid`) of the flat symbolic variable expression.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value On Exit
return value	`double`	value of flat symbolic constant

Notes: None

Examples of usage:

See below for a comprehensive example of all the methods relative to `FlatExpression`.

6.2.3 The `FlatOperatorExpression` Interface

This object class is derived from the `FlatExpression` class and provides flat symbolic information about expression objects created with the `UnaryExpression` and `BinaryExpression` methods (see sections 4.2.9 and 4.2.8).

This interface has two methods.

Method `GetOperator`

Declaration: `int GetOperator()`

Function: Returns the operator type of the flat symbolic unary or binary expression.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value On Exit
return value	`int`	operator type

Notes:

- The return function value describes the type of operator in the expression. It can be one of the following strings (the meaning is self-explanatory): `"sum"`, `"difference"`, `"product"`, `"ratio"`, `"power"`, `"minus"`, `"log"`, `"exp"`, `"sin"`, `"cos"`, `"tan"`, `"cot"`, `"sinh"`, `"cosh"`, `"tanh"`, `"coth"` (cf. section 2.4).

Examples of usage:

See below for a comprehensive example of all the methods relative to `FlatExpression`.

Method `GetOperand`

Declaration: `FlatExpression* GetOperand(int eid)`

Function: Returns one of the operands of the flat symbolic unary or binary expression.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
eid	`int`	the operand expression id

Arguments returned to client:

Argument	Type	Value On Exit
return value	`FlatExpression*`	flat symbolic operand expression

Notes:

- The operand expression id, `eid`, can only be 1 for unary expressions; it can be 1 or 2 for binary expressions.
- The return value is placed into an object of type `FlatExpression`. This can be analysed with the methods described in section 6.2.

Examples of usage:

See below for a comprehensive example of all the methods relative to `FlatExpression`.

6.2.4 Usage of `FlatExpression` Interface

The procedure to gather symbolic expression information from a `FlatExpression` is as follows.

1. Find out the `FlatExpression` type; there are three possible types: `Constant, Variable` and `Operator`.

   ```
   int fetype = fe->GetKind();
   ```

2. Depending on the type, cast the `FlatExpression` object dynamically to one of the following objects: `FlatConstantExpression, FlatVariableExpression` and `FlatOperatorExpression`.

   ```
   FlatConstantExpression* fke;
   FlatVariableExpression* fve;
   FlatOperatorExpression* foe;
   switch(fetype) {
   case Constant:
     fke = dynamic_cast<FlatConstantExpression*>(fe);
     break;
   case Variable:
     fve = dynamic_cast<FlatVariableExpression*>(fe);
     break;
   case Operator:
     foe = dynamic_cast<FlatOperatorExpression*>(fe);
     break;
   }
   ```

3. Now find out the actual information.

   - The `FlatConstantExpression` interface has only one method:

     ```
     double GetValue(void);
     ```

     which returns the actual value of the constant.

   - The `FlatVariableExpression` interface has only one method:

     ```
     long int GetVarIndex(void);
     ```

     which returns the flat variable index of the variable.

   - The `FlatOperatorExpression` interface has two methods: the first,

     ```
     int GetOpType(void);
     ```

     returns the type of operator of the expression (possible values are as on page 12); the second method,

     ```
     FlatExpression* GetFlatExpression(int index);
     ```

returns an operand of the operator given by `GetOpType()`. In the case of binary operators `index` can be 0 (for the left operand) or 1 (for the right operand). In the case of unary operators `index` can only be zero.

4. The symbolic analysis of the expression can go on in a recursive fashion until no more `FlatOperatorExpression`s are found.

5. It is important to notice that the deallocation of all the `FlatExpression` objects is left to the client.

The following is the actual coded example:

```
string          cname   ;
si*             index   ;
double          LB     ;
double          UB     ;
si*             varList ;
sd*             coefList;
FlatExpression* fe;

NetOpt->GetFlatMINLPConstraintInfo(532, cname, index, LB, UB,
                                   varList, coefList, fe)

FlatConstantExpression* fke;
FlatVariableExpression* fve;
FlatOperatorExpression* foe;
switch(fe->GetKind()) {
case Constant:
  fke = dynamic_cast<FlatConstantExpression*>(fe);
  cout << "Constraint Nonlinear Part is a Constant" << endl;
  cout << "Value = " << fke->GetValue() << endl;
  break;
case Variable:
  fve = dynamic_cast<FlatVariableExpression*>(fe);

  cout << "Constraint Nonlinear Part is a Variable" << endl;
  cout << "VarIndex = " << fve->GetVarIndex() << endl;
  break;
case Operator:
  foe = dynamic_cast<FlatOperatorExpression*>(fe);
  cout << "Constraint Nonlinear Part is an Operator" << endl;
  cout << "Operator Type = " << foe->GetOpType() << endl;
  break;
}
delete fe;
```

7 Implementation Restrictions

The following restrictions of the current implementation in comparison to the functionality described in this document are known:

1. *Limited number of arguments to* `IntSeq` *function*
   The `IntSeq` auxiliary function can have a maximum of 8 integer arguments.

2. *Currently available MINLP Managers*
   The following MINLP solvers have been interfaced to date to *ooOPS* and can be used in conjunction with the function `NewMINLPSolverManager` (see section 5.3):

   - SNOPT v. 5.3 (Systems Optimization Laboratory, Stanford University.)
     Accessed by specifying `sname = "snopt"`

8 An Example of the Use of *ooOPS*

The example is based on a slightly simplified form of the Resource Task Network (RTN) formulation proposed by C. Pantelides for process scheduling, plus a few spurious nonlinear terms which mess up the model hopelessly and completely, but help show how to use the methods which deal with nonlinearity. The formulation seeks to optimise a process involving NR resources $r = 1, .., NR$ and NK tasks $k = 1, .., NK$ over a time horizon discretised into NT time intervals $t = 1, .., NT$. It involves the objective function:

$$\max \sum_r (C_r^F (R_{r,NT} - R_{r0}) + \sum_t R_{rt}^2) \tag{7}$$

subject to the constraints:

$$R_{rt} = R_{r,t-1} + \sum_k \sum_{\theta=0}^{\tau_k} (\mu_{kr\theta} N_{k,t-\theta} + \nu_{kr\theta} \xi_{k,t-\theta}) + \Pi_{rt} + \frac{R_{rt} R_{r,t-1}}{\Pi_{rt}} \quad \forall r, t \tag{8}$$

$$0 \leq R_{rt} \leq R_{rt}^{\max} \quad \forall r, t \tag{9}$$

$$V_k^{\min} N_{kt} \leq \xi_{kt} \leq V_k^{\max} N_{kt} \quad \forall k, t \tag{10}$$

The variables in the above formulation are the following:

Variable	Type	Range	Description
R_{rt}	Continuous	$r = 1, .., NR$ $t = 0, .., NT$	Amount of resource r at time t
N_{kt}	Integer	$k = 1, .., NK$ $t = 1, .., NR$	Number of units used for task k at time t
ξ_{kt}	Continuous	$k = 1, .., NK$ $t = 1, .., NT$	Size of task k at time t

Table 1: Variable sets

A number of parameters and coefficients also appear in the formulation. These are listed in Table 2. The initial values of the resource levels, R_{r0}, are also fixed at given values R_r^*.

Parameter	Type	Description
C_r^f	Constant	Unit cost of resource r
μ_{rt}	Constant	Production or consumption coefficient referring to integer variable N_{kt}
ν_{kt}	Constant	Production or consumption coefficient referring to continuous variable ξ_{kt}
Π_{rt}	Constant	Amount of resource r made available from/to external sources at time t
τ_k	Constant	Duration of task k
$R_{rt}^{\max}$	Constant	Maximum amount of resource r that can be stored at time t
$V_k^{\min}$	Constant	Minimum useful capacity of unit suitable for task k
$V_k^{\max}$	Constant	Maximum useful capacity of unit suitable for task k

Table 2: Parameters appearing in the RTN formulation

8.1 Creating the MINLP

The first step is to create (an empty) `ops` object:

```
ops* RTNops=NewMINLP();
```

Now we have to add to this object (`RTNops`) the information that defines it. This is done below using the methods described in section 4.

8.1.1 Creating Variables

Variables are added to the `ops` object using the variable construction methods described in section 4.2.

Continuous variables are created using the method `NewContinuousVariable` in the way specified in section 4.2.1:

```
RTNops->NewContinuousVariable("Xi", IntSeq(1,1), IntSeq(NK,NT),
                              0.0, ooOPSPlusInfinity, 0.0);
RTNops->NewContinuousVariable("R", IntSeq(1,0), IntSeq(NR,NT),
                              0.0, ooOPSPlusInfinity, 0.0);
```

while integer variables are created using the method `NewIntegerVariable` described in section 4.2.2:

```
RTNops->NewIntegerVariable("N", IntSeq(1,1), IntSeq(NK,NT),
                           0, ooOPSPlusInfinity, 0);
```

All variables are two-dimensional, their lower bounds are initialised to zero and so are their default values. No upper bounds are imposed at this stage.

8.1.2 Creating Constraints

We now use the construction method presented in section 4.2.3 to create constraints (8) and (10):

```
RTNops->NewConstraint("ResourceBalance",IntSeq(1,1),
                      IntSeq(NR,NT), 0.0, 0.0);
RTNops->NewConstraint("EquipmentCapacityLB",IntSeq(1,1),
                      IntSeq(NK,NT), 0.0, ooOPSPlusInfinity);
RTNops->NewConstraint("EquipmentCapacityUB",IntSeq(1,1),
                      IntSeq(NK,NT), ooOPSMinusInfinity, 0.0);
```

We note that constraint 8 has been rearranged to the form:

$$\begin{array}{c} -R_{rt} + R_{r,t-1} + \sum_k \sum_{\theta=0}^{\tau_k} (\mu_{kr\theta} N_{k,t-\theta} + \nu_{kr\theta} \xi_{k,t-\theta}) + \\ + \Pi_{rt} + \frac{R_{rt} R_{r,t-1}}{\Pi_{rt}} = 0 \quad \forall r, t \end{array} \tag{8'}$$

while 10 has been split into two separate constraints of the form:

$$\xi_{kt} - V_k^{\min} N_{kt} \geq 0 \quad \forall k, t \tag{10'}$$

and

$$\xi_{kt} - V_k^{\max} N_{kt} \leq 0 \quad \forall k, t \tag{10''}$$

On the other hand, no constraint corresponding to (9) is introduced since this can be dealt with via upper bounds imposed on the R_{rt} variables (see section 8.1.6).

8.1.3 Adding Variables to Constraints

The constraints created in section 8.1.2 do not yet contain any variables. We now have to create appropriate variable occurrences in them by applying

the method `AddVariableSliceToConstraintSlice` as described in section 4.2.4 [5]:

- Constraint $(8')$

```
for(int r=1 ; r<=NR ; r++){
  for(int t=1 ; t<=NT ; t++){
    RTNops->AddVariableSliceToConstraintSlice("R",
             IntSeq(r,t), IntSeq(r,t), "ResourceBalance",
             IntSeq(r,t),IntSeq(r,t), -1.0);
    RTNops->AddVariableSliceToConstraintSlice("R",
             IntSeq(r,t-1),IntSeq(r,t-1),"ResourceBalance",
             IntSeq(r,t),IntSeq(r,t), 1.0);
    for(int k=1 ; k<=NK ; k++){
      for(int theta = 0;
          theta <= min(tau(k), (double) t - 1);
          theta++){
        RTNops->AddVariableSliceToConstraintSlice("N",
                 IntSeq(k,t-theta), IntSeq(k,t-theta),
                 "ResourceBalance",IntSeq(r,t),IntSeq(r,t),
                 mu(k,r,theta));
        RTNops->AddVariableSliceToConstraintSlice("Xi",
                 IntSeq(r,t-theta), IntSeq(r,t-theta),
                 "ResourceBalance",IntSeq(r,t),IntSeq(r,t),
                 nu(k,r,theta));
      }
    }
  }
}
```

- Constraint $(10')$

```
  for(int k=1 ; k<=NK ; k++){
    for(int t=1 ; t<=NT ; t++){
RTNops->AddVariableSliceToConstraintSlice("Xi",
                IntSeq(k,t), IntSeq(k,t),
                "EquipmentCapacityLB",
                IntSeq(k,t),IntSeq(k,t), 1.0);
RTNops->AddVariableSliceToConstraintSlice("N",
                IntSeq(k,t), IntSeq(k,t),
                "EquipmentCapacityLB",
                IntSeq(k,t),IntSeq(k,t),
                -Vmin(k,t));
  }
  }
```

[5] Here we assume that appropriate sets holding data components to the parameters of table 2 are already available.

- Constraint $(10')$

```
    for(int k=1 ; k<=NK ; k++){
      for(int t=1 ; t<=NT ; t++){
  RTNops->AddVariableSliceToConstraintSlice("Xi",
                  IntSeq(k,t), IntSeq(k,t),
                  "EquipmentCapacityUB",
                  IntSeq(k,t),IntSeq(k,t), 1.0);
  RTNops->AddVariableSliceToConstraintSlice("N",
                  IntSeq(k,t), IntSeq(k,t),
                  "EquipmentCapacityUB",
                  IntSeq(k,t),IntSeq(k,t),
                  -Vmax(k,t));
      }
    }
```

8.1.4 Objective Function

The creation of the objective function is effected using the method `NewObjectiveFunction` (*cf.*section 4.2.11):

```
RTNops->NewObjectiveFunction("TotalProfit","max");
```

For this example, the name of the objective function to be maximised is `"TotalProfit"` .

8.1.5 Objective Function Coefficients

The creation of the objective function is followed by the declaration of the coefficients of the variable instances appearing in it. The appropriate method is `AddVariableSliceToObjectiveFunction` described in section 4.2.12:

```
for(int r=1 ; r<=NR ; r++){
  RTNops->AddVariableSliceToObjectiveFunction("R",IntSeq(r,NT),
           IntSeq(r,NT), CFR(r),"TotalResources");
  RTNops->AddVariableSliceToObjectiveFunction("R",IntSeq(r,0),
           IntSeq(r,0), -CFR(r),"TotalResources");
}
```

Again, we assume the coefficients C_r^F that appear in the objective function are stored in set `CFR`.

8.1.6 Modifying the Variable Bounds

Variable R_{rt} is bounded as shown in eqn. (9). Method `SetVariableBounds` can be used to impose these bounds which may be different for different elements of the R_{rt} set:

```
for(int r=1 ; r<=NR ; r++){
  for(int t=1 ; t<=NT ; t++){
    RTNops->SetVariableBounds("R",IntSeq(r,t),IntSeq(r,t),
                              0.0, R_max[r][t]);
  }
}
```

Moreover, the initial resource levels R_{r0} are fixed at given values R_r^*. This is achieved by setting both lower and upper bounds to R_r^*:

```
for(int r=1 ; r<=NR ; r++){
  RTNops->SetVariableBounds("R",IntSeq(r,0),IntSeq(r,0),
                            R_star[r], R_star[r]);
}
```

8.1.7 Modifying the Constraint Bounds

Constraint (8) has a right hand coefficient which is not constant so we use the function `SetConstraintBounds` to obligate this restriction.

```
for(int r=1 ; r<=NR ; r++){
  for(int t=1 ; t<=NT ; t++){
    RTNops->SetConstraintBounds("ResourceBalance",
                                IntSeq(r,t), IntSeq(r,t),
                                -Pi[r][t], -Pi[r][t]);
  }
}
```

8.1.8 Creating the Nonlinear Parts

We create nonlinear parts of constraints and objective function by employing the methods described in section 4.2.6.

```
// make nonlinear part of constraints
RTNops->NewVariableExpression("Leaf1", "R",
                              IntSeq(1,1), IntSeq(NR, NT));
RTNops->NewVariableExpression("Leaf2", "R",
                              IntSeq(1,1), IntSeq(NR, NT));
```

```
RTNops->BinaryExpression("Expr1",
                         "Leaf1", IntSeq(1,1), IntSeq(NR, NT),
                         "Leaf2", IntSeq(1,1), IntSeq(NR, NT),
                         "product");
RTNops->NewConstant("Const1", IntSeq(1,1), IntSeq(NR, NT), 1.0);
for(int r=1 ; r<=NR ; r++){
  for(int t=1 ; t<=NT ; t++){
    RTNops->SetConstantValue("Const1", IntSeq(r,t), 1/Pi[r][t]);
  }
}
RTNops->NewConstantExpression("Leaf3", "Const1",
                              IntSeq(1,1), IntSeq(NR, NT));
RTNops->BinaryExpression("Expr1",
                         "Expr1", IntSeq(1,1), IntSeq(NR,NT),
                         "Leaf3", IntSeq(1,1), IntSeq(NR,NT),
                         "ratio");

// make nonlinear part of objective function
RTNops->NewConstant("Const2", IntSeq(1,1), IntSeq(1,1), 2.0);
RTNops->NewConstantExpression("Power2", "Const2",
                              IntSeq(1,1), IntSeq(1,1));
RTNops->BinaryExpression("Expr2",
                         "Leaf1", IntSeq(1,1), IntSeq(NR,NT),
                         "Const2", IntSeq(1,1), IntSeq(1,1),
                         "power");
RTNops->NewConstant("Zero", IntSeq(1), IntSeq(1), 0);
RTNops->NewConstantExpression("Expr3", "Zero",
                              IntSeq(1), IntSeq(1));
for(int r=1; r<=NR; r++) {
  for(int t=1; t<= NT; t++) {
     RTNops->BinaryExpression("Expr3",
                              "Expr3", IntSeq(1), IntSeq(1),
                              "Expr2", IntSeq(r,t), IntSeq(r,t),
                              "sum");
  }
}
```

8.1.9 Assigning Expressions to Constraints and Objective Function

After having created the expressions representing the nonlinear parts, we assign them to the existing constraints and objective function by using the methods described in section 4.2.7.

```
// assign expressions to constraints
RTNops->AssignExpressionSliceToConstraintSlice
        ("Expr1", IntSeq(1,1), IntSeq(NR, NT),
```

```
         "ResourceBalance", IntSeq(1,1), IntSeq(NR, NT));

// assign expression to objective function
RTNops->AssignExpressionToObjectiveFunction
        ("Expr3", "TotalProfit");
```

8.2 MINLP Solution

Having created the `ops` object `RTNops`, we now have to combine it with an appropriate MINLP solver to create an `opssystem` that can be solved.

8.2.1 Creating an MINLP Solver Manager Object

We start by creating an appropriate `opssolvermanager` object. The usage of this method is described in section 5.3. Here, we create a MINLP solver manager based on the SNOPT solver:

```
opssolvermanager* SnoptManager = NewMINLPSolverManager("snopt");
```

8.2.2 Creating an MINLP System

Using the MINLP solver manager object created above, a `opssystem` is created from the `ops` object:

```
opssystem* RTNsystem = SnoptManager->NewMINLPSystem(RTNops);
```

8.2.3 Solving the MINLP

At last, the above `MINLPsystem` can be solved by invoking its `Solve` method as described in section 5.5.4:

```
RTNsystem->Solve();
```

8.3 Accessing the Solution of the MINLP

Various aspects of the MINLP solution can be accessed using methods described in section 4.4.

8.3.1 Obtaining Information on the Variables

The optimal values of the variables can be obtained using method 4.4.1. In this example we have three types of variables:

- R_{rt}

  ```
  double* R_value;
  double* R_LB;
  double* R_UB;
  ```

```
RTNops->GetVariableInfo("R",IntSeq(1,1),IntSeq(NR,NT),
                        R_value, R_LB, R_UB);
```

- N_{kt}

```
int* N_value;
int* N_LB;
int* N_UB;

RTNops->GetVariableInfo("N",IntSeq(1,1),IntSeq(NK,NT),
                        N_value, N_LB, N_UB);
```

- ξ_{kt}

```
double* Xi_value;
double* Xi_LB;
double* Xi_UB;

RTNops->GetVariableInfo("Xi",IntSeq(1,1),IntSeq(NK,NT),
                        Xi_value, Xi_LB, Xi_UB);
```

8.3.2 Obtaining Information on the Objective Function

The value of the objective function can be accessed using the method of section 4.4.3:

```
char*   obj_type;
double* obj_value;

RTNops->GetObjectiveFunctionInfo("TotalProfit",
                                 objt_type, obj_value);
```

The value of the objective function returned is based on the current (hopefully optimal) values of the variables.

References

[SP99] E.M.B. Smith and C.C. Pantelides. A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex minlps. *Computers and Chemical Engineering*, 23:457–478, 1999.