A Modeling Interface to Non-Linear Programming Solvers

An instance: xMPS, the extended MPS format

Bjarni V. Halldórsson[*] bjarni@cmu.edu

Erlendur S. Thorsteinsson[†] esth@cmu.edu

Bjarni Kristjánsson[‡] bjarni@maximal-usa.com

9th February 2000

Abstract

We present a Modeler-Optimizer Interface (MOI) for general closed form Non-Linear Programs (NLP), which can be used to to transfer NLPs in a clear and simple manner between optimization components in a distributed environment. We demonstrate how this interface allows first order derivative information to be easily calculated on the optimizer's side, using automatic differentiation, hence removing the bottleneck of communicating derivative information between the modeler and the optimizer.

We also show how this interface directly corresponds to a file format for NLPs, the extended MPS format (xMPS). This format directly extends the standard MPS file format for linear and mixed integer programs to include NLPs and permits a standardized way of transferring benchmark problems. The format spares the modeler the tedious task of calculating derivative information with minimal extra work required by the optimizer and thus increases efficiency. This work was originally done at Maximal Software in order to connect the MPL modeling language [15] with non-linear solvers.

1 Introduction

In this paper we present a Modeler-Optimizer Interface (MOI) for general closed form Non-Linear Programs (NLP), which can be used to to transfer NLPs in a clear and simple manner between optimization components in a distributed environment.

Let us start by looking at how most general purpose NLP optimizers work and what they demand from their environment. Most (if not all) follow this iterative framework:

```
Initialize a set of current solutions $X$.
While the optimal solution has not been found:
    Obtain function/derivative values at points stemming from $X$.
    Update $X$.
```

Furthermore, many NLP optimizers use higher order derivatives when updating the current solution, in particular second order derivatives.

Hence, general purpose NLP optimizers assume that function values, derivative values and possibly higher order derivative values can be provided by an external source at any given point in the domain. As an alternative, only function values can be provided to the optimizer, in which case it has to approximate the derivatives numerically.

[*]Department of Mathematical Sciences; Carnegie Mellon University; Pittsburgh, PA 15213; U.S.A.

[†]Graduate School of Industrial Administration; Carnegie Mellon University; Pittsburgh, PA 15213; U.S.A.

[‡]Maximal Software, Inc.; 2111 Wilson Blvd., Ste. 700; Arlington, VA 22201; U.S.A.

Neither of these two options are appealing. The first one requires that functions that evaluate the partial derivatives be explicitely written. This can be a complex task, prone to errors, time consuming and is usually intractable for large problems. The second option has built-in numerical errors which can lead to numerical instability in the optimization process.

More prohibitive, however, is the fact that both options are computationally expensive. The first option requires that the function value and the n partial derivative values (n being the number of variables) be calculated and passed separately from the modeler to the optimizer. The second option requires a constant number of function evaluations for each partial derivative. This means that $O(n)$ function evaluations would be required to approximate all the partial derivatives. It is, however, well known that using the techniques of automatic differentiation only $O(1)$ evaluations are needed to calculate all the necessary values [9]. The interface we propose takes advantage of that.

Both of the approaches described above also cause much traffic between the modeler and the optimizer. They essentially require all optimization components to reside on the same computer, or on computers connected by a high speed network and even then the amount of traffic may cause congestion and delays. In many applications, however, we would like the modeler to reside on a different computer than the optimizer, where the modeler would typically reside on a user friendly machine and the optimizer on a computationally powerful machine. Also, we would like to be able to distribute the optimizer between different computers. Hence, each component of the optimizer must be able to have all the necessary calculations done locally. Our interface minimizes traffic by sending only the necessary information from the modeler to the optimizer at the start of the optimization process and then the function evaluation and differentiation is performed on the optimizer's side.

Notice that none of this is a problem if we are only considering Linear Programs (LP) or Mixed Integer Programs (MIP). Then it is quite simple to transfer the necessary information around, as the coefficient tableau (matrix) provides all the details. Using the tableau it is quite simple to calculate the function value and all derivative values at a given point. This has become the *de facto* standard way of exchanging information on LPs/MIPs, both between different optimization components in software and as a file format [5, 10, 11, 12]. We argue that it is also necessary to have a standardized interface for NLPs between different optimization components. The Modeler-Optimizer Interface (MOI) we propose has four main advantages:

- It provides a clear separation between the modeler and the optimizer (or different optimization components) with minimal communication.
- It is very simple, both from users' and software developers' perspectives, enabling a short learning curve.
- The interface allows function and derivative information to be easily calculated (using automatic or implicit differentiation), thus sparing the modeler the tedious task of calculating derivative information with minimal extra work required by the optimizer.
- As a special instance, it has a direct correspondence to a file format for NLPs, the extended MPS format (xMPS) [19]. This format directly extends the standard MPS file format for linear and mixed integer programs to include NLPs and permits a standardized way of transferring benchmark problems and maintaining testbeds, thus facilitating communication between researchers in this field and comparison of different NLP optimizers.

2 Background

2.1 Expression trees and stack machines

The underlying data structure of MOI relies heavily on the representation of closed form functions as expression trees, or equivalently, as stack machines. There+fore we include a brief introduction below, but detailed discussion on these topics can be found, e.g., in [1, 14, 18].

For illustration let us consider the following optimization problem:

$$\begin{array}{llcrcll}
\min & \sin(x_1) + x_1x_2 & + & 2x_2 & & & (obj) \\
\text{s.t.} & & & x_1 + x_2 & \leq & 4, & (g_1) \\
& 4\ln(x_1x_2) & + & x_1 & \geq & 1. & (g_2)
\end{array} \tag{1}$$

We are only going to be concerned with the non-linear part of each expression, as the linear part can be easily communicated between the modeler and optimizer in the form of a matrix, as in the case of LP/MIP. This allows for quicker derivative evaluation of the linear part as the derivatives of linear functions are trivial to evaluate. The linear part can, however, also be transported along with the non-linear part.

Looking at the linear part of (1) we notice that x_1 appears linearly in (g_1) and (g_2), and x_2 appears linearly in (obj) and (g_1). In addition, (obj) and (g_2) have some non-linear elements to them. Rewriting and regrouping we can thus state (1) in the following manner, noting the matrices that can be sent directly to the optimizer (in square brackets):

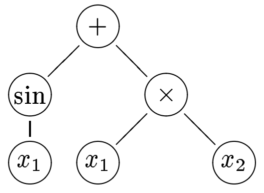
$$\begin{array}{lccccc}
\min & \left\{ \sin(x_1) + x_1x_2 \right\} & + & \begin{bmatrix} 0 & 2 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} & & \\
\text{s.t.} & \left\{ \begin{array}{c} 0 \\ 4\ln(x_1x_2) \end{array} \right\} & + & \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} & \begin{array}{c} \leq \\ \geq \end{array} & \begin{bmatrix} 4 \\ 1 \end{bmatrix}
\end{array}$$

Using *expression trees* we can describe the non-linear part of (obj) and (g_2) in a simple manner. We first convert each (non-linear) expression into *post-fix* notation (Polish notation) [1, 14].

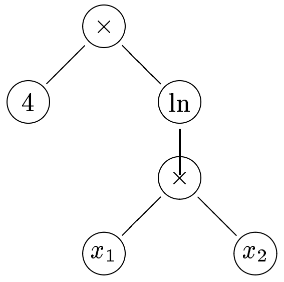
In-fix notation	Post-fix notation
$\sin(x_1) + x_1 \times x_2$	x_1 sin x_1 x_2 $\times$ $+$
$4\ln(x_1 \times x_2)$	x_1 x_2 $\times$ ln 4 $\times$

We then construct an expression tree by scanning the post-fix expression from left to right and

- for every coefficient/variable, create a leaf node containing the coefficient/variable,
- for every n-ary operator (e.g., elementary arithmetic operators and logarithmic and trigonometric functions, see App. A for full details), join the n most previous nodes that do not have a parent yet by an inner node containing the operator.



Non-linear part of (obj)



Non-linear part of (g_2)

An expression tree is evaluated *bottom-up*, i.e., in order to evaluate a node we first evaluate its children recursively and then use their values to evaluate the node itself. This can be accomplished by traversing the tree in a *depth-first* manner.

The post-fix form also directly corresponds to a *stack machine*. We can evaluate a post-fix expression directly as a stack machine by scanning it from left to right and

- for every coefficient/variable, push the coefficient/variable onto the stack.

- for every n-ary operator, pop n elements off the stack, apply the operator to them and push the result onto the stack.

We get the following for the non-linear part of (*obj*):

Input:	x_1	sin	x_1	x_2	$\times$	$+$	
	$\uparrow\searrow$	$\uparrow\searrow$	$\uparrow\searrow$	$\uparrow\searrow$	$\uparrow\searrow$	$\uparrow\searrow$	
					x_2		
				x_1	x_1	$x_1 \times x_2$	
Stack:	$\varnothing$	x_1	$\sin(x_1)$	$\sin(x_1)$	$\sin(x_1)$	$\sin(x_1)$	$\sin(x_1) + x_1 \times x_2$

The input line shows the input element being processed. Below each element is the stack as it is before processing the element, and below-right is the stack after processing the element.

As can be seen from this example, evaluating the post-fix expression as a stack machine is very simple. Also, parsing expressions written in mathematical notation, such as (1), separating the linear and non-linear parts and transforming the non-linear parts into a post-fix expression is quite simple and well known [1, 14].

The Modeler-Optimizer Interface (MOI) we propose will be based on an extended stack machine, where elements are never popped off the stack, thus allowing expressions to refer to all intermediate values previously calculated.

2.2 Automatic differentiation

One of the greatest benefit of MOI is that it allows for direct application of automatic differentiation to the underlying data structure. By using automatic differentiation it is shown in [9] that by taking care in storing quantities that are common to the function and the partial derivatives, the cost of evaluating all the partial derivatives is no more than five times the cost of evaluating the function.

Automatic differentiation is an application of the chain rule of calculus. Each node n in the expression tree, which contains an operator, expresses the function $f_n = \text{operator}(f_{c_1}, \dots, f_{c_k})$, where $c_1, \dots, c_k$ are the children of the node n. The chain rule then implies that following holds for the partial derivative of f_n with respect to x_i:

$$\frac{\partial f_n}{\partial x_i} = \frac{\partial f_n}{\partial f_{c_1}} \frac{\partial f_{c_1}}{\partial x_i} + \cdots + \frac{\partial f_n}{\partial f_{c_k}} \frac{\partial f_{c_k}}{\partial x_i}. \tag{2}$$

Note that the evaluation of $\partial f_n / \partial f_{c_i}$ is only the evaluation of a derivative of an operator with respect to its its operands and is a simple expression dependant only on the value of its operands. For example, if the operator is addition then $f_n = +(f_{c_1}, f_{c_2}) = f_{c_1} + f_{c_2}$ and

$$\frac{\partial f_n}{\partial x_i} = 1 \frac{\partial f_{c_1}}{\partial x_i} + 1 \frac{\partial f_{c_2}}{\partial x_i}. \tag{3}$$

If the operator is multiplication then $f_n = \times(f_{c_1}, f_{c_2}) = f_{c_1} \times f_{c_2}$ and

$$\frac{\partial f_n}{\partial x_i} = f_{c_2} \frac{\partial f_{c_1}}{\partial x_i} + f_{c_1} \frac{\partial f_{c_2}}{\partial x_i}. \tag{4}$$

Let N be the set of all leaf nodes and let P_{l_j} be the set of all edges (n_p, n_c), from a parent node to a child node, on the the path from the root node to the leaf node $l_j \in N$. Let f_r denote the function the expression tree represents, i.e., the root node. If we expand the partial derivatives using (2) recursively and then multiply out, we get

$$\frac{\partial f_r}{\partial x_i} = \sum_{l_j \in N} \left(\prod_{(n_p, n_c) \in P_{l_j}} \frac{\partial f_{n_p}}{\partial f_{n_c}} \right) \frac{\partial f_{l_j}}{\partial x_i}, \tag{5}$$

where the sum is taken over the leaf nodes. For every leaf l in the expression tree, which contains a variable x_j,

$$\frac{\partial f_l}{\partial x_i} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise,} \end{cases} \tag{6}$$

and for every leaf l in the expression tree, which contains a constant,

$$\frac{\partial f_l}{\partial x_i} = 0. \tag{7}$$

By (6)–(7) the term $\partial f_{l_j}/\partial x_i$ in (5) will always be zero unless the node l_j contains the variable x_i and will then be one. One consequence of this is that the derivative with respect to x_i of a subtree that does not contain x_i will be identically zero.

Notice that the parenthesized expressions in each term of the sum (5) can be written $\partial f_r/\partial f_{l_j}$ if we define

$$\frac{\partial f_r}{\partial f_n} = \prod_{(n_p,n_c)\in P_n} \frac{\partial f_{n_p}}{\partial f_{n_c}},$$

where P_n is the set of edges from the root node to node n. Thus $\partial f_r/\partial f_{l_j}$ is independent of the variable x_i, with respect to which we want to differentiate. The variables only occur in the leaf nodes so if we know all the subtree function values we can evaluate all partial derivatives in an incremental fashion by descending the tree from the root using these updating rules:

- At an operator node, n, we store the partial derivative of that node with respect to the function, i.e., $\partial f_r/\partial f_n$.
- At a leaf node, l, containing a variable, x_i, we increment the partial derivative $\partial f_r/\partial x_i$ by $\partial f_r/\partial f_l$, the derivative of the leaf node with respect to the function.

Notice that in both of these cases, if p is the parent of the current node n then

$$\frac{\partial f_r}{\partial f_n} = \frac{\partial f_r}{\partial f_p}\frac{\partial f_p}{\partial f_n},$$

where $\partial f_r/\partial f_p$ has been previously evaluated and $\partial f_p/\partial f_n$ is a simple expression that relies only on the values of the children of p, as in the examples (3)–(4). See App. A for full details on what operators are supported and what their derivatives are with respect to their operands.

3 Modeler-Optimizer Interface (MOI)

As mentioned in Sec. 1, non-linear optimizers generally assume that the modeler stores the model. The optimizer will then ask the modeler for function values and possibly derivative values at any point

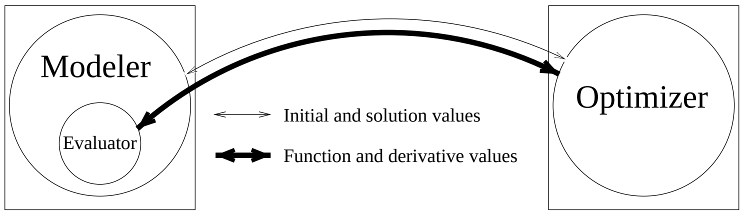


Figure 1: Current communication setup.

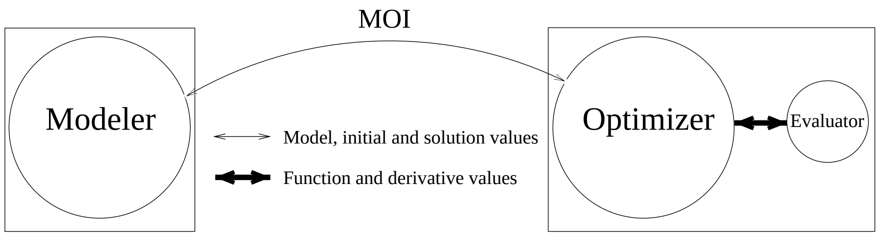

MOI
Modeler
Optimizer
Evaluator
Model, initial and solution values
Function and derivative values

Figure 2: Proposed communication setup using MOI.

in the domain. This is prohibitive to a more widespread use of non-linear solvers since it does not provide for a clear separation and a communication interface between the modeler and the optimizer.

The Modeler-Optimizer Interface (MOI), see Fig. 2, transports the model from the modeler to the optimizer, which can then communicate parts of the model to a function and derivative evaluator as needed. Note that the evaluator need not be a part of either the modeler nor the optimizer. As there generally is a high level of communication between the evaluator and the optimizer, it is important that a copy of the evaluator reside close to the optimizer. In particular if the optimizer has distributed components that each require function and derivative evaluations, it is important a copy of the evaluator reside close to each of them. In contrast notice that the modeler generally does not require function and derivative evaluations and hence the evaluator need not reside close to the modeler.

3.1 The MOI callable library format

The Modeler-Optimizer Interface is a specific implementation of a stack machine as a format for transporting the non-linear part of an optimization problem between different optimization components.

In each line of the stack machine we store an operator and its left and right operand. The operands can be of four types: A variable, in which case we store the variable number; a constant, in which case we store the value of the constant; a previous line, in which case we store its index; or non-existent if the operand is monomial. A single stack machine (with l lines) for a non-linear expression can thus be transported using the following seven arrays; Oper, ArgL, ArgR, IndexL, IndexR, ValueL, ValueR:

Oper A vector of l integers, $(\mathrm{Oper}_1, \mathrm{Oper}_2, \ldots, \mathrm{Oper}_k)$. Oper_i contains the unique code for the operator in line i (see App. A).

ArgL, ArgR Vectors of l characters, $(\mathrm{Arg[LR]}_1, \mathrm{Arg[LR]}_2, \ldots, \mathrm{Arg[LR]}_l)$. Contain the argument type: `X` indicates a variable, `C` indicates a constant and `V` indicates a previous line.

IndexL, IndexR Vectors of l integers, $(\mathrm{Index[LR]}_1, \mathrm{Index[LR]}_2, \ldots, \mathrm{Index[LR]}_l)$. If $\mathrm{Arg[LR]}_i$ indicates a variable then $\mathrm{Index[LR]}_i$ contains the variable number and if $\mathrm{Arg[LR]}_i$ indicates a previous line then $\mathrm{Index[LR]}_i$ contains the line number.

ValueL, ValueR Vectors of l floating point numbers, $(\mathrm{Value[LR]}_1, \mathrm{Value[LR]}_2, \ldots, \mathrm{Value[LR]}_l)$. If $\mathrm{Arg[LR]}_i$ indicates a constant then $\mathrm{Value[LR]}_i$ is that constant.

We can then describe the objective function in example (1) in the following way:

Line	Oper	ArgL	ArgR	IndexL	IndexR	ValueL	ValueR		Expression
1	OpSIN	`X`		1					$v_1 : \sin x_1$
2	OpMUL	`X`	`X`	1	2				$v_2 : x_1 x_2$
3	OpADD	`V`	`V`	1	2				$v_3 : v_1 + v_2$

3.1.1 Transporting multiple constraints

To transport all the non-linear constraints as a whole we line up all the stack machines one after the other. To do this we add the eighth array:

NlBegin A vector of integers of length $m+2$, where m is the total number of constraints (not including the objective function). All lines from the stack machine corresponding to the objective function are stored in lines NlBegin_0 to $\text{NlBegin}_1 - 1$. All lines from the stack machine corresponding to the i^{th} constraint are stored in NlBegin_i to $\text{NlBegin}_{i+1} - 1$. If constraint i does not contain any non-linear elements then NlBegin_i and NlBegin_{i+1} should be equal. The last entry in NlBegin points to the position after the last stack machine.

Example (1) is then stored in the MOI data structure in the following way:

NlBegin		Line	Operator	ArgL	ArgR	IndexL	IndexR	ValueL	ValueR		Expression
0		1	OpSIN	X		1					$v_1 : \sin x_1$
3		2	OpMUL	X	X	1	2				$v_2 : x_1 x_2$
3		3	OpADD	V	V	1	2				$v_3 : v_1 + v_2$
6		1	OpMUL	X	X	1	2				$v_1 : x_1 x_2$
		2	OpLOG	V		1					$v_2 : \ln v_1$
		3	OpMUL	V	C	2			4.0		$v_3 : 4.0 v_2$

3.1.2 Interface to non-linear solvers

Non-linear solvers can now be interfaced in a similar fashion to LP/MIP solvers, with the addition of an extra function call where the non-linear part is transported from the modeler to the optimizer.

```
int LoadNonLinear(void* EnvironPtr, int NumVars, int NumCons
            int NumStacks, int* NlBegin, int* Oper, char* ArgL, char* ArgR,
            int* IndexL, int* IndexR, double* ValueL, double* ValueR)
```

Figure 3: C interface to a non-linear solver.

3.2 Automatic derivative algorithm for MOI

In the previous section we saw that we can evaluate the partial derivatives of a given node in our expression tree if we know the partial derivative of the node's parent, the value of the node and the value of the node's siblings.

We will now demonstrate how we can evaluate all partial derivatives. The algorithm described is known as the reverse mode of automatic differentiation [16] and calculates all the partial derivatives in two sweeps over the MOI data structure. First we sweep forward over the MOI data structure and calculate and store the value of all the nodes (the value of the partial expression in the expression tree rooted at the given node). This corresponds to ascending the corresponding expression tree. We then sweep backward over the MOI data structure to calculate all the partial derivatives, which corresponds to descending the corresponding expression tree.

We store the value of node n_i in NodeValue_i and the partial derivative of node n_i with respect to the root, $\partial f_r / \partial f_{n_i}$, in $(\partial V)_i$. The partial derivative of f_r with respect to x_i is stored in $(\partial x)_i$. The partial derivative of the operator Oper with respect to its child C, evaluated at V_L, V_R, is denoted $\partial\text{Oper}/\partial C(V_L, V_R)$. Evaluate(Oper,$V_L$, V_R) is the evaluation of Oper with arguments V_L and V_R and StackLength is the total number of lines in the stack machine.

```
Algorithm CalculateAllFirstOrderPartialDerivatives
    // First calculate the value of all nodes
    For $k \leftarrow 1$ to StackLength
        $[V_L, V_R] \leftarrow$ RetrieveChildrenValues($k$)
        $\text{NodeValue}_k \leftarrow \text{Evaluate}(\text{Oper}_k, V_L, V_R)$
    // Then calculate the derivative values
    Initialize all $(\partial x)_i$ to 0, and $(\partial V)_{\text{StackLength}}$ to 1
    For $k \leftarrow$ StackLength to 1
        $[V_L, V_R] \leftarrow$ RetrieveChildrenValues($k$)
        For $C \leftarrow \{\text{L, R}\}$
            If $\text{Arg}C_k$ is X
                $(\partial x)_{\text{Index}C_k} \leftarrow (\partial x)_{\text{Index}C_k} + (\partial V)_k \times \partial\text{Oper}_k/\partial C(V_L, V_R)$
            Else If $\text{Arg}C_k$ is V
                $(\partial V)_{\text{Index}C_k} \leftarrow (\partial V)_k \times \partial\text{Oper}_k/\partial C(V_L, V_R)$

Function $[V_L, V_R] \leftarrow$ RetrieveChildrenValues($k$)
    For $C \leftarrow \{\text{L,R}\}$
        If $\text{Arg}C_k$ is C Then $V_C \leftarrow \text{Value}C_k$
        Else If $\text{Arg}C_k$ is X Then $V_C \leftarrow x_{\text{Index}C_k}$
        Else $V_C \leftarrow \text{NodeValue}_{\text{Index}C_k}$
```

3.2.1 An Example

Looking back at example (1), let us evaluate the partial derivatives of the non-linear part of the objective function at the point $(\pi/2, 2\text{e}/\pi)$:

Initialize $(\partial x)_i$ to 0.0 and initialize the derivative value of the root node with respect to the function, $(\partial V)_3$ to 1.0.

Forward sweep to evaluate NodeValue. We find that $\text{NodeValue}_1 = 1.0$, $\text{NodeValue}_2 = 1.0$ and $\text{NodeValue}_3 = 2.0$. We note that NodeValue_3 is the function value f_r.

Backward sweep to evaluate the partial derivatives:

Line 3

Oper	ArgL	ArgR	IndexL	IndexR	ValueL	ValueR	NodeValue	(∂V)
OpADD	V	V	1	2			2.0	1.0

Evaluate the derivative values of lines 1 and 2, $(\partial V)_1 \leftarrow 1.0 \times 1.0$, $(\partial V)_2 \leftarrow 1.0 \times 1.0$.

Line 2

Oper	ArgL	ArgR	IndexL	IndexR	ValueL	ValueR	NodeValue	(∂V)
OpMUL	X	X	1	2			1.0	1.0

Increment $(\partial x)_1$ by $1.0 \times 2\text{e}/\pi$, and increment $(\partial x)_2$ by $1.0 \times \pi/2$.

Line 1

Oper	ArgL	ArgR	IndexL	IndexR	ValueL	ValueR	NodeValue	(∂V)
OpSIN	X		1				1.0	1.0

Increment $(\partial x)_1$ by $1.0 \times \cos(\pi/2)$.

We then have $\partial f_r/\partial x_1 = 2\text{e}/\pi + \cos(\pi/2) = 2\text{e}/\pi$ and $\partial f_r/\partial x_2 = \pi/2$ at the specified point $(\pi/2, 2\text{e}/\pi)$.

4 The extended MPS (xMPS) file format

As mentioned in Sec. 1, the main criteria for a file format for non-linear programs is that it (a) be simple to use, (b) be compatible with some popular format used for linear programs, (c) extend in a simple way to memory representation, and (d) allow for automatic (analytical) differentiation. This would ensure, e.g., that (a) benchmarks problems could be easily transferred between systems, (b) existing LP/MIP readers would only have to be slightly modified to read the new NLP format, (c) problems could be transferred in an efficient manner from the modeler to the optimizer, and (d) optimizers could easily be provided with exact derivatives at every point in the solution space without the derivative functions being explicitly included in the model.

The MPS file format is a widely accepted standard for expressing LP/MIP problems, recognized by many LP/MIP solvers and modeling languages. In this section we are going to describe a special instance of MOI, a file format for non-linear programs, called xMPS, which extends MPS. More information on MPS can be found in [11] and how different software packages implement MPS in [5, 10, 12]. We describe below only our additions to MPS, a full description of xMPS is given in *xMPS, the Extended MPS Format for Non-Linear Programs* [19].

4.1 The xMPS file format

The MOI interface is implemented in the the xMPS file format by adding two new indicator records to the relaxed MPS format [19]:

NONLINEAR indicator record. The corresponding data records contain the non-linear part of a constraint. A complete constraint is the sum of the linear part specified by the COLUMNS section and the non-linear part specified by this section. This indicator record should appear directly after the COLUMNS section.

INITIAL indicator record. The corresponding data records are the initial (non-zero) values for each variable. This indicator record should appear directly after the BOUNDS section.

We describe the data records in more detail below.

NONLINEAR. The non-linear part of each constraint is represented by a list of lines where each line is of the type:

Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
	Constraint name	Line name	Operator	Argument 1	Argument 2

For each constraint the lines form a stack machine which expresses the non-linear part in the same manner as in the MOI interface. The lines for a particular constraint should appear contiguously, i.e., the NONLINEAR section is divided into blocks based on the constraint names. The line names within each block should be distinct and different from the variable and constraint names. The last line name in each block should be RES, indicating the end, and therefore the result, of the expression.

The operator is in the set of known keywords for operators, see App. A or www.maximal-usa.com/xmps for an updated version.

The arguments are either constants, variable names drawn from the COLUMN section or line names of lines above the current line in the current block.

Variables that are only referred to in nonlinear equations (i.e., do not occur linearly in any constraint) still need to be declared in the COLUMNS section.

INITIAL. Data records in the INITIAL section specify the values of the initial solution values. The data records have the following form:

Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
	Init. name	1^{st} var. name	1^{st} var. value	2^{nd} var. name	2^{nd} var. value

where fields 5 and 6 are optional. If they are used, they contain another variable/value pair, as in fields 3 and 4, respectively. Init. name is the name of the initialization vector.

4.1.1 An example

To illustrate the xMPS format let us consider example (1) from section 2.1 again, starting the optimization at $x^0 = (x_1, x_2) = (1, 1)$. Examining (1) we notice that x_1 appears linearly in (g_1) and (g_2), and x_2 appears linearly in (obj) and (g_1). In addition, (obj) and (g_2) have some non-linear elements to them. This problem would be represented in xMPS as follows:

NAME	demo.xmps				
OBJSENSE					
MIN					
ROWS					
N	obj				
L	g1				
G	g2				
COLUMNS					
	x1	g1	1	g2	1
	x2	obj	2	g1	1
NONLINEAR					
	obj	v1	sin	x1	
	obj	v2	mult	x1	x2
	obj	RES	add	v1	v2
	g2	v1	mult	x1	x2
	g2	v2	ln	v1	
	g2	RES	mult	4	v2
RHS					
	rhs	g1	4	g2	1
INITIAL					
	init	x1	1	x2	1
ENDATA					

The ROWS section starts by declaring the constraints, (obj), (g_1) and (g_2) and stating their sense. The COLUMNS section then declares the variables x_1 and x_2 and the linear part of each constraint. We note, e.g., that according to this description the linear part of constraint (obj), the objective function, is $2x_2$.

The NONLINEAR section then describes the non-linear part of each constraint. For example, the description of (obj) would be parsed in the following manner: We would first take $\sin(x_1)$ and label it v_1. We then take $x_1 \times x_2$ and label it v_2. According to the last line in the (obj) subsection, the result should be $v_1 + v_2$, which is $\sin(x_1) + x_1 x_2$.

As mentioned before, each constraint of the problem is then the sum of the linear and non-linear parts. Therefore, constraint (obj) is $\sin(x_1) + x_1 x_2 + 2x_2$.

5 Related work

5.1 Automatic differentiation

The automatic differentiation algorithm presented in Sec. 3.2 is an example of reverse mode automatic differentiation. Extensive research has been done on automatic differentiation (see, e.g., [9]), both in trying to take advantage of repetitive structures and also in evaluating derivatives of more complex functions than the closed form functions that we have limited ourselves to. Advanced optimization packages may be able to benefit from this research.

The example we gave of an automatic differentiation is intended as a simple example of how automatic differentiation can be used in conjunction with optimization. This is with a minimal implementation effort and can in fact be implemented in less than 100 lines of code. Even so, this derivative evaluation method is order of number of variables faster than evaluating derivatives using the finite difference methods that are currently employed in many optimization packages.

We believe that this method has several benefits over having the modeler link automatic differentiation packages to his code. As the automatic differentiation code is directly linked with the solver we can ensure that the communication cost between the derivative evaluation code and the solver is minimized. We also believe that the effort on part of the modeler of writing the model in the MOI format is considerably less than making the function evaluation routines amenable to some of the common automatic derivative codes such as ADIC [3] and ADOL-C [8].

5.2 Explicit differentiation

The MOI data structure, whether in file format or memory representation, can be easily converted into expression trees. Expression trees that contain the derivatives can be explicitly constructed from them. This method for calculating derivatives is called explicit differentiation and is an alternate to the proposed method of automatic differentiation. Deriving efficient algorithms for explicit differentiation may be a worthwhile line of research.

5.3 Interfaces to modeling languages

The MOI format was originally implemented as an interface between the MPL modeling language and non-linear solvers. Some other commonly used modeling languages such as AMPL [7] and GAMS [6] also have interfaces to non-linear programming solvers. These implementations have some excellent features. For instance, AMPL appears to do a good work of trying to extract features that are important for efficient evaluation of derivatives. Their main drawbacks are however:

- The communication language is proprietary. We note that the adoption of the MOI interface is not dependant on the adoption of any specific modeling language, such as MPL, AMPL or GAMS.
- Some of the communications is done via file systems, which is slower than through memory.
- A clear separation between the modeler and the optimizer is not provided as the modeler is responsible for evaluating derivatives, providing for bottlenecks in communications. The optimizer also relies on the modeling language for implementing efficient algorithms to exploit the optimizers structures. This division of labor is generally unreasonable.
- The optimizer cannot "see" the model and does not have a clear way of extracting attributes from it that can enable it to reduce the model. An example of this would be if the optimizer would like to branch on integer variables. These branchings may significantly reduce the model and the optimizer will want to exploit that fact.

5.4 Lancelot SIF format

The Lancelot SIF format [4] is a file format for transporting non-linear programs. It is, however, not in widespread use. We believe this is mainly because the format is too complicated and tries too heavily to exploit some specific problem structures. We believe that such structure exploitation should be passed independently of the problem itself.

6 Conclusion

We have described a general interface (MOI) for NLP solvers that provides a separation between the modeler and the solver. It directly extends common interfaces to LP solvers and allows the solver to take advantage of automatic differentiation techniques.

MOI also gives the optimizer a complete description of the model as a series of expressions. This allows the optimizer to have a global viewpoint of the model instead of the traditional local viewpoint of only being able to calculate function and derivative values at any given point. This global viewpoint is essential if the optimizer wants to automatically detect how to best distribute the model between optimization components. Furthermore it gives rise to new and interesting research directions in non-linear programming such as operator and expression specific optimization.

As a particular application we have provided an extension (xMPS) for non-linear programming to the widely used linear and mixed integer programming file format MPS.

Both MOI and xMPS have been implemented for the commercial modeling system MPL, which is available from Maximal Software at www.maximal-usa.com, using the NLP solvers CONOPT [2] and GRG2/LSGRG2 [17].

7 Acknowledgments

The authors would like to thank Arne Drud for providing the initial draft of the non-linear file format, and Jochen Könemann for making valuable comments when proofreading this paper.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison–Wesley Publishing Company, 1986.

[2] Arki Consulting and Development. *CONOPT, reference manual*, 1998.

[3] Christian Bischof, Lucas Roh, and Andrew Mauer. ADIC—an extensible automatic differentiation tool for ANSI-C. Technical report, Argonne National Laborotory, 1996.

[4] Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. *The LANCELOT Specification File.* www.numerical.rl.ac.uk/lancelot/spec/spec.html.

[5] Dash Optimization, Inc. *XPRESS–MP: User Manuals*, 1999. www.dashopt.com.

[6] GAMS Development Corpartation. *GAMS, A Users Guide.* www.gams.com.

[7] David M. Gay. Hooking your solver to AMPL. Technical report, Bell Laboratories, 1997. www.ampl.com.

[8] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, and A. Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM TOMS*, 22(2):131–167, June 1996.

[9] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, Amsterdam, 1989.

[10] IBM. *The IBM Optimization Solutions and Library (OSL) for Multi-platforms Version 2.* ism.boulder.ibm.com/es/oslv2/features/welcome.htm.

[11] IBM. *Mathematical Programming System Extended/370 (MPSX/370) Program Reference Manual.*

[12] ILOG. *Using the CPLEX Callable Library*, 1997. www.cplex.com.

[13] Ketron Management Science. *MPSIII, Mathematical Programming Software*, 1998. www.ketron.com.

[14] Robert L. Kruse. *Programming with Data Structures.* Prentice Hall, 1989.

[15] Maximal Software Inc. *MPL Modeling System 4.1, User's guide*, 1999. www.maximal-usa.com.

[16] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer Series in Operations Research. Springer Verlag, August 1999.

[17] Optimal Methods Inc. *GRG2 Users Guide.* www.optimalmethods.com.

[18] Robert Sedgewick. *Algorithms in C.* Addison–Wesley Publishing Company, 1990.

[19] Erlendur S. Thorsteinsson. xMPS, the extended MPS format for non-linear programs. Technical Report 99–224, Carnegie Mellon University Mathematical Sciences Department, December 1999.

A Non-linear operators[*]

Operator (O)	Tag	Keyword	Constant	$\partial O/\partial L$	$\partial O/\partial R$
	OpNONE	NONE	0		
$L+R$	OpADD	ADD	1	1	1
$L-R$	OpSUB	SUB	2	1	-1
$L\times R$	OpMUL	MULT	3	R	L
L/R	OpDIV	DIV	4	$1/R$	$-L/R^2$
$-L$	OpNEG	NEG	5	-1	
$L+R$	OpSUM	SUM	6	1	1
L^2	OpSQR	SQR	7	$2L$	
L^R	OpPOW	POW	8	L^{R-1}	$\ln L\times L^R$
$\sqrt{L}$	OpSQRT	SQRT	9	$1/(2\sqrt{L})$	
$L\%R$	OpMOD	MOD	10		
e^L	OpEXP	EXP	11	e^L	
$\ln L$	OpLOG	LOG	12	$1/L$	
$\log L$	OpLOG10	LOG10	13	$1/(L\ln 10)$	
$\sin L$	OpSIN	SIN	14	$\cos L$	
$\cos L$	OpCOS	COS	15	$-\sin L$	
$\tan L$	OpTAN	TAN	16	$1/\cos^2 L$	
$\arcsin L$	OpASIN	ASIN	17	$1/(\sqrt{1-L^2})$	
$\arccos L$	OpACOS	ACOS	18	$-1/(\sqrt{1-L^2})$	
$\arctan L$	OpATAN	ATAN	19	$1/(1+L^2)$	
$\arctan(L/R)$	OpATAN2	ATAN2	20	$1/(L(1+(L/R)^2))$	$-R/(L^2+R^2)$
$\sinh L$	OpSINH	SINH	21	$\cosh L$	
$\cosh L$	OpCOSH	COSH	22	$\sinh L$	
$\tanh L$	OpTANH	TANH	23	$1/(\cosh^2 L)$	
$\text{arcsinh}L$	OpASINH	ASINH	24	$1/\sqrt{1+L^2}$	
$\text{arccosh}L$	OpACOSH	ACOSH	25	$-1/\sqrt{1+L^2}$	
$\text{arctanh}L$	OpATANH	ATANH	26	$1/(1-L^2)$	
$\text{sign}(L)$	OpSIGN	SIGN	27		
$\mid L \mid$	OpABS	ABS	28		
$\lceil L\rceil$	OpCEIL	CEIL	29		
$\lfloor L\rfloor$	OpFLOOR	FLOOR	30		
$\text{round}(L)$	OpROUND	ROUND	31		
$\text{trunc}(L)$	OpTRUNC	TRUNC	32		

Table 1: Non-linear operators.

[*]Please visit www.maximal-usa.com/xmps for a current version of supported operators. Suggestions for additional operators should be submitted to xmps@maximal-usa.com.