VI Congresso
Nacional de Mecânica Aplicada
e Computacional
University of Aveiro, Portugal
17-19 April 2000

OBJECT ORIENTED AUTOMATIC DIFFERENTIATION AND LEXICAL ANALYSIS IN ENGINEERING OPTIMIZATION

Luís F. D. Brás[1] and Álvaro F. M. Azevedo[2]

ABSTRACT

Engineering optimization problems may be formulated as nonlinear programs (NLP), defined by an objective function and a set of inequality and equality constraints. This NLP may be solved by several types of methods, requiring or not the calculation of first and second derivatives of the functions. Second-order methods have some advantages, namely quadratic convergence and precision. The main drawback is the necessity of second derivatives and their storage in a large Hessian matrix. When the sparsity of the Hessian is conveniently exploited, efficient algorithms may be developed. With this formulation, problems with thousand of variables and constraints have already been solved. The problems related to the evaluation of first and second derivatives may be overcome with an automatic differentiation algorithm (AD). This approach avoids the difficulties associated to the numerical evaluation of the derivatives and enables the application of the algorithm to any type of expression. Each nonlinear program is described by a set of functions that are interpreted by an object oriented parser. Automatic differentiation is based on operator overloading and Rall numbers. All the code is written in C++ and exploits the advantages of inheritance and polymorphism. In order to demonstrate some of these features a structural optimization problem is presented.

1. INTRODUCTION

Structural optimization problems are usually formulated with nonlinear functions and are subject to convergence difficulties due to local minimums, numerical problems or unsuitable initial values. Techniques such as scaling and line search are employed to reduce such problems. Programming complex techniques with a standard language, in particular when a parser and automatic differentiation are required, is usually cumbersome, error prone and hard to maintain. Object oriented programming introduces a new concept in terms of code organization with each individual component treated as an object. An object can interact with

[1] MSc Student, Faculty of Engineering, University of Porto, Portugal, http://civil.fe.up.pt/cv/megabyte
[2] Assistant Professor, Faculty of Engineering, University of Porto, Portugal, http://www.fe.up.pt/~alvaro

the others using methods or operators. Object oriented compilers often lead to a performance decrease, requiring adequate programming techniques in order to maintain a reasonable efficiency. In this paper the implementation of the optimization algorithm in C++ is described and a structural optimization example is presented.

2. LAGRANGE-NEWTON METHOD

Solving a mathematical program consists on the minimization of a function subject to inequality and equality constraints. The following general formulation is used

$$\text{Min. } f\left(\underset{\sim}{x}\right) \qquad \underset{\sim}{x}=(x_1,...,x_n) \tag{1}$$

subject to

$$g\left(\underset{\sim}{x}\right)\leq \underset{\sim}{0} \qquad \underset{\sim}{g}=(g_1,...,g_m) \tag{2}$$

$$h\left(\underset{\sim}{x}\right)=\underset{\sim}{0} \qquad \underset{\sim}{h}=(h_1,...,h_p) \tag{3}$$

Squared slack variables are added to the inequality constraints, allowing for a generic treatment of all the constraints as equalities.

$$g_i\left(\underset{\sim}{x}\right)\leq 0 \longrightarrow g_i\left(\underset{\sim}{x}\right)+s_i^2=0 \tag{4}$$

This operation causes a significant growth in the total number of variables of the nonlinear problem. With the implementation of appropriate techniques this inconvenience may be overcome[1].

The Lagrangian of the nonlinear program is given by

$$L\left(\underset{\sim}{s},\ \underset{\sim}{l}^g,\underset{\sim}{x},\ \underset{\sim}{l}^h\right)=f\left(\underset{\sim}{x}\right)+\sum_{k=1}^{m}\left[\ l_k^g\left(g_k\left(\underset{\sim}{x}\right)+s_k^2\right)\right]+\sum_{k=1}^{p}\left[\ l_k^h\,h_k\left(\underset{\sim}{x}\right)\right] \tag{5}$$

In this expression $\underset{\sim}{l}^g$ and $\underset{\sim}{l}^h$ are the Lagrange multipliers associated with the inequality and equality constraints, respectively. Vector $\underset{\sim}{X}$ contains all the variables ordered in a carefully selected manner, in order to facilitate the solution of the system of linear equations whose coefficient matrix is the Hessian of the Lagrangian (see below).

$$\underset{\sim}{X}=\left(\underset{\sim}{s},\ \underset{\sim}{l}^g,\underset{\sim}{x},\ \underset{\sim}{l}^h\right) \tag{6}$$

The solution of the nonlinear program is a saddle point of the Lagrangian and the following necessary condition must hold

$$\nabla L=\underset{\sim}{0} \tag{7}$$

The Newton method can be used to calculate the solution of the system of nonlinear equations that corresponds to (7). When this technique is employed, the optimization method is termed Lagrange-Newton. For each Newton iteration (q) the following system of linear equations must be solved

$$\underset{\sim}{H}\left(\underset{\sim}{X}^{q-1}\right)\Delta\underset{\sim}{X}^{q}+\nabla L\left(\underset{\sim}{X}^{q-1}\right)=\underset{\sim}{0} \tag{8}$$

Matrix $\underset{\sim}{H}$ is the Hessian of the Lagrangian. In the current implementation this system of linear equations may be solved by Gaussian elimination or iteratively using the conjugate gradient method.

In each Newton iteration vector $\Delta\underset{\sim}{X}^{q}$ is used to update the current solution. In order to increase the reliability of the process and the rate of convergence $\Delta\underset{\sim}{X}^{q}$ is multiplied by a scalar value α^{q}. A standard line search algorithm is used to calculate the value of α^{q} that minimizes the error in the direction $\Delta\underset{\sim}{X}^{q}$.

$$Error\left(a^{q}\right)=\left\|\nabla L\left(\underset{\sim}{X}^{q-1}+a^{q}\,\Delta\underset{\sim}{X}^{q}\right)\right\| \tag{9}$$

Algorithms based on the Newton method are much more reliable when the initial solution is close to a minimum and when the problem is not ill-conditioned. These conditions are more easily fulfilled when the original nonlinear problem is subject to suitable transformations. Variable substitution and constraint normalization are used to increase the reliability of the optimization process.

3. AUTOMATIC DIFFERENTIATION

Second-order methods, such as the Lagrange-Newton method (see Section 2), depend on the calculation of first and second derivatives. Its convergence rate and accuracy are directly associated with the error of the derivatives. Therefore using a method that enables the calculation of the exact derivatives is highly advantageous. Automatic differentiation can be used to accurately calculate the derivative of any function and also perform its evaluation. By applying the same algorithm to the previously derived expression, derivation to any order can be easily performed. The current implementation of automatic differentiation relies on operator overloading, i.e., redefinition of built-in operators. When an object oriented programming language is used, overloaded operators may be implemented as global functions, friend functions or class members[2].

With the automatic differentiation algorithm described in this paper, the evaluation of a function and the calculation of its derivatives are simultaneously performed. Some of these techniques are exemplified with the following function

$$f\left(x_1,x_2\right)=x_1+2.0\;x_2 \tag{10}$$

The first derivative is determined with the following steps, which reproduce the tasks performed by the algorithm.

$$\frac{\partial f(x_1, x_2)}{\partial x_1} = \frac{\partial}{\partial x_1}(x_1 + 2.0\ x_2) \tag{11}$$

$$= \frac{\partial}{\partial x_1}(x_1) + \frac{\partial}{\partial x_1}(2.0\ x_2) \tag{12}$$

The derivative of the product of a pair of function is given by

$$\frac{\partial}{\partial x_j}(f\ g) = \frac{\partial f}{\partial x_j} g + f\ \frac{\partial g}{\partial x_j} \tag{13}$$

Using this equation, (12) becomes

$$\frac{\partial}{\partial x_1}(x_1) + \frac{\partial}{\partial x_1}(2.0)\ x_2 + 2.0 \frac{\partial}{\partial x_1}(x_2) \tag{14}$$

Considering $x_1 = 10.0$ and $x_2 = 20.0$, the final result is

$$1.0 + 0.0 \times 20.0 + 2.0 \times 0.0 = 1.0 \tag{15}$$

In each step it is only necessary to store the result of the function evaluation and the numerical value of each derivative. The basic components of (10) are the variables x_1 and x_2 and the constant 2.0. Their derivatives in order to x_1 and x_2 are stored in the following vectors

$$\frac{\partial}{\partial \underset{\sim}{x}}(x_1) = [1.0, 0.0] \tag{16}$$

$$\frac{\partial}{\partial \underset{\sim}{x}}(x_2) = [0.0, 1.0] \tag{17}$$

$$\frac{\partial}{\partial \underset{\sim}{x}}(2.0) = [0.0, 0.0] \tag{18}$$

These vectors are replaced in expression (14), which is generalized in order to include the derivatives in order to x_1 and x_2.

$$[1.0, 0.0] + [0.0, 0.0] \times 20.0 + 2.0 \times [0.0, 1.0] \tag{19}$$

$$= [1.0, 0.0] + [0.0, 2.0] \tag{20}$$

$$= [1.0, 2.0] \tag{21}$$

The vector (21) represents the derivatives of function (10) in order to x_1 and x_2.

For each expression component, the only storage requirements are a real number for the result of its evaluation and a vector for the values of the partial derivatives. This automatic differentiation technique can be encapsulated in a C++ class, whose operators are prepared to deal with the function and its derivatives. This class was named CRall after L. B. Rall, who applied the technique in an extended version of Pascal[3]. When evaluating a function or calculating its derivatives, priority rules are used in order to preserve the usual operator precedence.

Second-order derivatives can be evaluated in a similar manner, requiring the storage of the Hessian of each expression component in a $n \times n$ matrix. Second-order derivatives of the product of a pair of functions can be obtained with

$$\frac{\partial^2}{\partial x_i \partial x_j}(f\,g) = \frac{\partial^2 f}{\partial x_i \partial x_j} g + \frac{\partial f}{\partial x_j}\frac{\partial g}{\partial x_i} + \frac{\partial f}{\partial x_i}\frac{\partial g}{\partial x_j} + f\frac{\partial^2 g}{\partial x_i \partial x_j} \tag{22}$$

Some problems may arise when the number of variables of the optimization problem is large and the Hessian matrix is required, such as insufficient storage and lack of efficiency. To overcome these problems a container class was developed, which takes into account the sparsity of the matrix.

In practice, automatic differentiation can be useful in the analysis of complex multivariant functions. Whereas it is an effective technique with any programming language, automatic differentiation becomes even more attractive when object oriented concepts such as operator overloading are used. Automatic differentiation is usually more time consuming than a direct implementation of a hand or machine computed analytical derivative[3].

4. OBJECT ORIENTED EXPRESSION PARSER

When automatic differentiation is used to calculate the derivatives of an expression, the behavior of the operators between two Rall numbers has to be defined, according to the rules described in Section 3. This behavior is not dependent on the type of expression parsing. In terms of expression parsing two approaches are available: the easier alternative is to rely on the C++ compiler[3] and the other consists on the development of a parsing engine[4]. The former requires a recompilation of the code whenever the expression is modified and is troublesome when scaling techniques need to be applied. The later is not subject to these drawbacks but requires the availability or the development of a complete expression parser.

The expression parser described in this section is based on the work of Joey Rogers[4]. Some bugs were removed and new features were added, such as the support for the most common intrinsic functions (e.g., *sin*, *cos*, *sqrt*, *log*, *pow*) and the implementation of some scaling techniques.

Three types of tokens can be extracted from an expression: constants, variables and operators. According to the priority rules of the parentheses and operators, all the tokens are inserted in a binary tree. The evaluation of the root object causes a postorder traversal of the

binary tree, resulting in the value of the expression as a whole. This technique is exemplified with the following function

$$1+\frac{2\times 3}{4-2} \tag{23}$$

The corresponding tree is shown in Fig. 1.

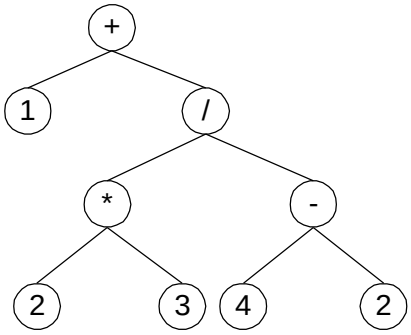


Fig.1 – Binary tree of the expression (23)[4].

Operators can be connected to terminal objects, such as constants and variables, or to subtrees.

Fig. 2 shows the hierarchy diagram of the classes that implement the behavior of all the expression components.

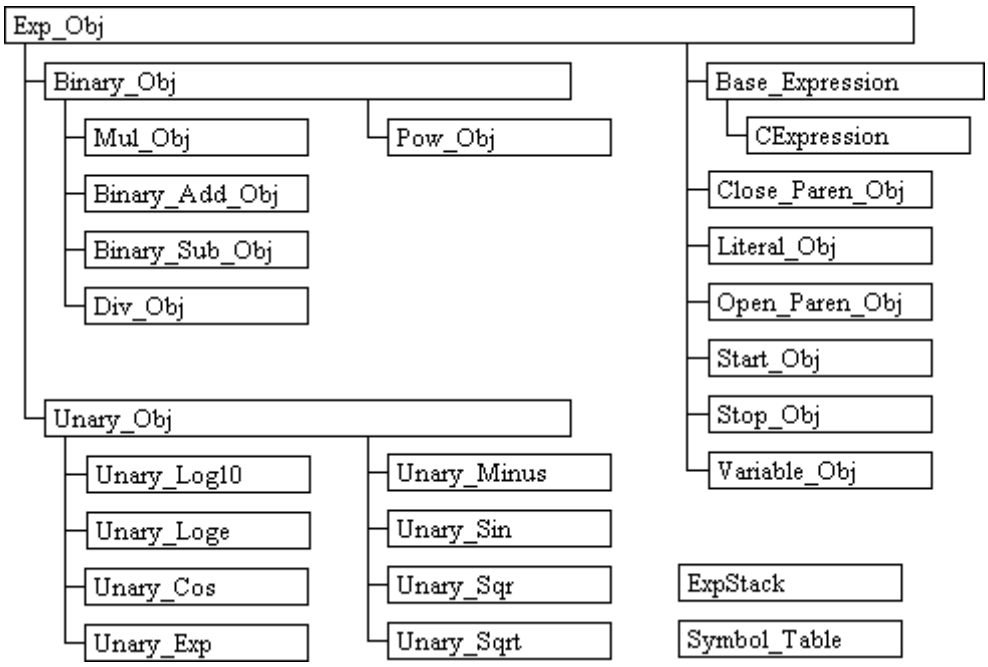


Fig. 2 – Hierarchy diagram of the expression parser.

The base class *Exp_Obj* defines the common interface of all the derived classes. Polymorphism is used to facilitate an easy manipulation of the expression objects, regardless of their specified type or functionality.

Table 1 exemplifies the creation of the binary tree for the function (23).

Table 1 – Building the binary tree of the expression (23)[4].

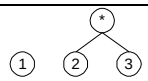
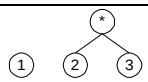
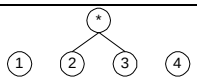
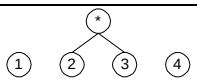
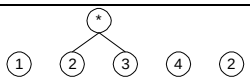
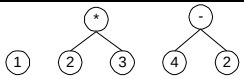
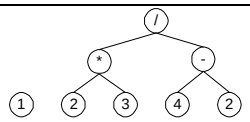
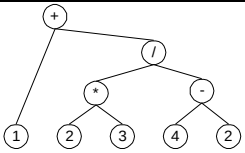
Expression	Expression Stack	Operator Stack
1+2*3/(4-2)		
+2*3/(4-2)	1	
2*3/(4-2)	1	+
*3/(4-2)	1 2	+
3/(4-2)	1 2	+*
/(4-2)	1 2 3	+*
(4-2)		+/
4-2)		+/(
-2)		+/(
2)		+/(-
)		+/(-
		+/
		+
		

Fig. 3 shows the complete algorithm that is used to create the binary tree of a generic expression.

Only the variable name is stored in the class *Variable_Obj* (Fig. 2). Its value and scaling factor comes from a symbol table that must be supplied when the expression is evaluated. The symbol table is implemented as a singly linked list. When an expression is evaluated, its first and second derivatives are also evaluated, due to the behavior of the Rall numbers.

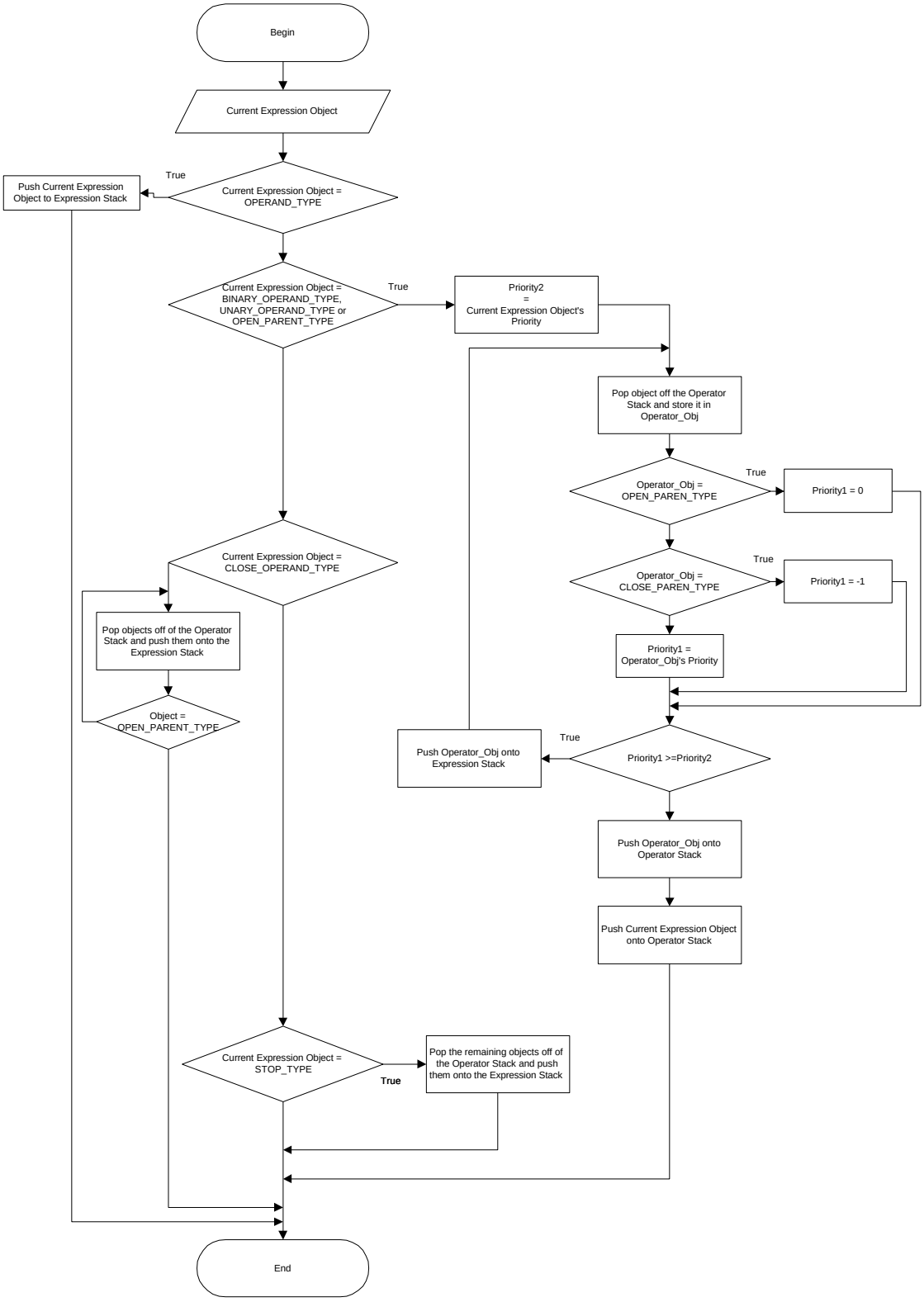
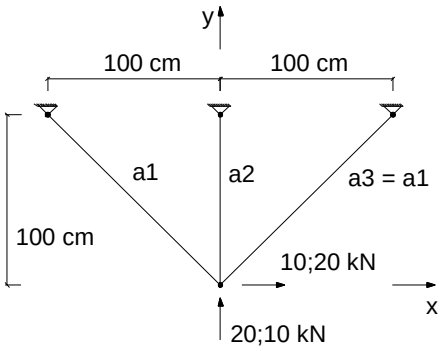


Fig. 3 – Algorithm for the creation of the expression tree.

5. EXAMPLE

A truss optimization problem[1] and an excerpt of the C++ source code describing the problem are shown in Fig. 4. The truss has three bars, two load cases and the properties indicated in Fig. 4b).



a) Truss bar

$E = 20000\,kN/cm^2$

$\bar{s} = 14\,kN/cm^2$

$a_{\min} = 0.15\,cm^2$

b) Properties

```
COpt< double > Opt;
COptionsOpt< double > OptOpt;
int Iterat;
double Error;
double ObjFunc;
Symbol_Table< double > SymbTab;

m_dOpt.InitializeWith(4,14);
m_dOpt.SetObjFunction("565.6854*a1+100*a2");
m_dOpt.SetEqConstr("141.4213*a1*d_1_1-10",1);
(…)
m_dOpt.SetEqConstr("141.42*a1*d_2_2+200*a2*d_2_2-10",4);
m_dOpt.SetIneqConstr("-a1+0.15",1);
m_dOpt.SetIneqConstr("-a2+0.15",2);
m_dOpt.SetIneqConstr("-100*d_1_1+100*d_1_2-14",3);
(…)
m_dOpt.SetIneqConstr("-100*d_2_1-100*d_2_2-14",14);
m_dOpt.SetVarByType("a1", 1.5, INDEPENDENT);
m_dOpt.SetVarByType("a2", 1.5, INDEPENDENT);
m_dOpt.SetVarByType("d_1_1", 1.0, DEPENDENT);
(…)
m_dOpt.SetVarByType("d_2_2", 1.0, DEPENDENT);

m_dOpt.Solve(OptOpt, Iterat, Error, ObjFunc, SymbTab);

cout << "Results" << endl;
cout << "Final error          = " << Error << endl;
cout << "Number of Iterations = " << Iterat << endl;
cout << "Objective function   = " << ObjFunc << endl;
SymbTab.ShowValues();
```

c) Source code

Fig. 4 – Optimization of a three bar truss.

The results are presented in Table 2 and Fig. 5.

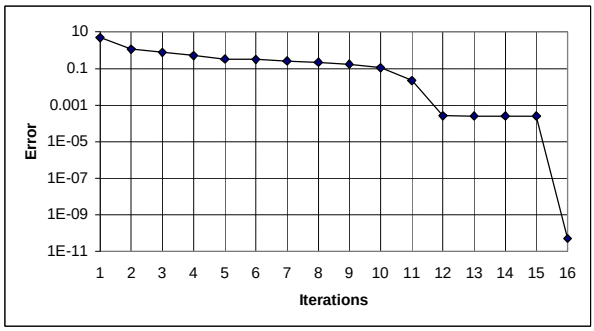
Table 2 – Optimal solution.

a1	a2	d_1_1	d_1_2	d_2_1	d_2_2
1.280127	0.788262	0.055237	0.059051	0.110475	0.029526

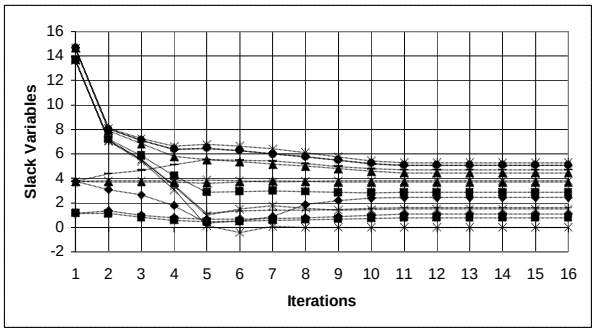
CONCLUSION

The work presented in this paper consists on the combination of a robust second-order optimization method, an object oriented parser and an automatic differentiation algorithm based on operator overloading and Rall numbers. Some preliminary numerical experimentation indicates that the code is very versatile, i.e., its adaptation to new types of optimization problems is very easy. Code maintenance and the implementation of alternative numerical techniques is facilitated by the object oriented design and its inherent features, such as polymorphism and inheritance. Some work is still needed to improve the global efficiency of the optimization process.

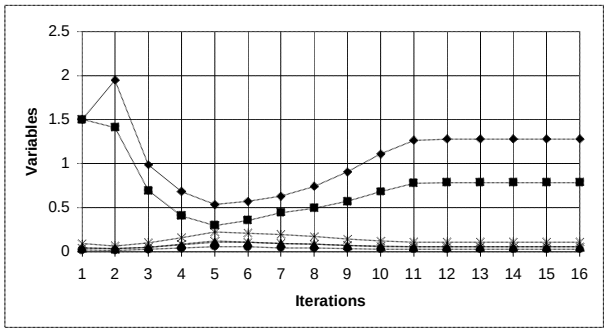
a) History of the objective function.



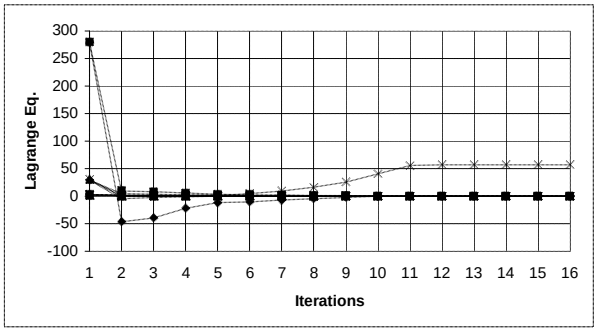
b) History of the error.



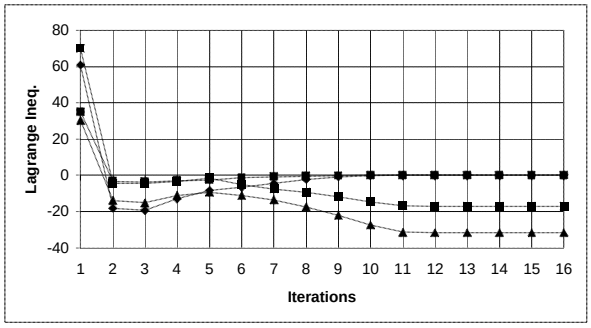
c) History of the slack variables.



d) History of the variables.



e) History of the Lagrange multipliers of the equality constraints.



f) History of the Lagrange multipliers of the inequality constraints.

Fig. 5 – History of optimization of the three bar truss.

REFERENCES

[1] Azevedo, A. F. M., "Optimization of Structures with Linear and Nonlinear Behavior", Ph.D. thesis (in Portuguese), Faculty of Engineering, University of Porto, 1994.

[2] Stroustrup, Bjarne, "The C++ Programming Language", Third Edition, Addison-Wesley, 1997.

[3] Barton, John J. and Nackman, Lee R., "Automatic Differentiation", C++ Report, pp. 61-63, February 1996

[4] Rogers, Joey, "An Object-Oriented Expression Evaluator", C/C++ Users Journal, pp. 43-51, April 1996.