Object oriented implementation of a second-order optimization method

L. F. D. Bras[1] & A. F. M. Azevedo[1]
[1]*Civil Engineering Department, Faculty of Engineering, University of Porto, Portugal.*

Abstract

A structural optimization problem may be formulated as a single nonlinear program (NLP) and solved with a second-order method. This class of methods requires first and second derivatives of the objective function and of the inequality and equality constraints. When the size of the problem is large, derivative calculation may become tedious and error prone. Automatic differentiation (AD) techniques provide exact values for the derivatives without user intervention. In this paper the object oriented implementation of a Lagrange-Newton optimization algorithm is presented. An object oriented parser is used to interpret all the functions that describe the NLP. First and second derivatives are calculated with AD techniques using Rall numbers and their associated overloaded operators. A shape optimization example is presented to illustrate the proposed techniques.

1 Introduction

Object oriented programming techniques provide higher-level tools to manipulate information. Before coding a numerical algorithm classes must be developed in order to implement the basic operations that will be required at a later stage. These classes are composed of sets of variables that store the information related to the class, and operations that safely manipulate that data. Vectors, sparse matrices, functions and Rall numbers [1] are examples of classes and addition, multiplication, evaluation, entry removal and partial derivation are examples of operations. The computer code that is described later required the development of a large number of basic classes. Object oriented techniques, such as inheritance, polymorphism and templates [2], played an important role in the organization of the relationship between classes and provided significant economies in terms of development time and code extension. The higher-level algorithm utilizes the basic classes and implements new ones in order to perform its task.

2 Nonlinear programming

In the present work, structural optimization problems are formulated as a single nonlinear program (NLP), whose general form is shown in eqn (1).

$$\begin{aligned} &Min.\quad f(x_1,\dots,x_n) \\ &\quad subject\ to \\ &\quad g_j(x_1,\dots,x_n)\le 0 \;\rightarrow\; g_j(x_1,\dots,x_n)+s_j^2=0 \quad (j=1,\dots,m) \\ &\quad h_k(x_1,\dots,x_n)=0 \qquad\qquad\qquad\qquad\qquad (k=1,\dots,p) \end{aligned} \tag{1}$$

Squared slack variables are used to allow for the replacement of all the inequality constraints by equality constraints. The NLP (1) is solved by the Lagrange-Newton method using the necessary condition $\nabla L=0$, where L is the Lagrangian. The resulting system of $n+2m+p$ nonlinear equations (2) is solved by the Newton method [3].

$$\begin{aligned} &\frac{\partial f}{\partial x_i}+\sum_{j=1}^{m}\lambda_j^g\frac{\partial g_j}{\partial x_i}+\sum_{k=1}^{p}\lambda_k^h\frac{\partial h_k}{\partial x_i}=0 && (i=1,\dots,n) \\ &2\lambda_j^g\, s_j=0 && (j=1,\dots,m) \\ &g_j+s_j^2=0 && (j=1,\dots,m) \\ &h_k=0 && (k=1,\dots,p) \end{aligned} \tag{2}$$

In each Newton iteration (q) the following system of linear equations has to be solved

$$\underset{\sim}{H}\left(\underset{\sim}{X}^{q-1}\right)\Delta\underset{\sim}{X}^{q}+\nabla L\left(\underset{\sim}{X}^{q-1}\right)=0 \tag{3}$$

In eqn (3) H is the Hessian matrix and X is a vector containing all the variables (x_i, s_j, λ_j^g, λ_k^h). The current solution is updated with

$$\underset{\sim}{X}^{q}=\underset{\sim}{X}^{q-1}+\alpha^q\,\Delta\underset{\sim}{X}^{q} \tag{4}$$

where α^q is the line search parameter. Scaling techniques are used to improve the robustness and efficiency of the iterative process (see Section 6). The sparsity pattern of the Hessian matrix H is exploited in order to save storage and unnecessary operations. With this approach, optimization problems with thousands of independent variables have already been solved [4].

3 Automatic differentiation

First and second order derivatives of the functions *f*, *g* and *h* are required in order to calculate the components of eqn (3). The robustness and convergence rate of the Newton algorithm are highly dependent on the precision of these derivatives. When numerical differentiation is used, each variable has to be shifted. The size of this perturbation has a significant influence in the precision of the derivative and a suitable value for its size may be difficult to estimate [5]. The calculation of the Hessian matrix requires a large number of perturbations and function evaluations. To avoid errors associated with numerical differentiation, the user of the optimization code might be obliged to supply derived functions. When the size of the NLP is large, this technique becomes cumbersome and error prone. A possible solution to these problems is the development of code that applies the rules of differentiation. This technique is called automatic differentiation (AD) and is difficult to implement, due to the complexity of the algorithms involved. Object oriented programming techniques provide tools that relieve the task of programming the differentiation of user-supplied functions. In the present work, an expression parser is used to decompose each function into operands and operators (see Section 5). The operators are redefined in order to apply the rules of differentiation.

4 Rall numbers

A Rall number is a set of information that contains the numerical value of an operand, the numerical value of its gradient and the numerical value of its Hessian. When an operator is applied to an operand or to a pair of operands, the differentiation rules are used in order to perform the necessary tasks that lead to the correct evaluation of the gradient and Hessian of the result of the operation. In the present work a Rall number is implemented as a C++ class [1]. Its data members and the multiply operator are shown in the following example involving the functions $f(x_1, x_2)$ and $g(x_1, x_2)$ (see Table 1). Eqns (5) and (6) correspond to the evaluation of the first and second derivatives of the product of a pair of functions.

$$\frac{\partial}{\partial x_1}(f\,g) = \frac{\partial f}{\partial x_1}\,g + f\,\frac{\partial g}{\partial x_1} \tag{5}$$

$$\frac{\partial^2}{\partial x_1 \partial x_2}(f\,g) = \frac{\partial^2 f}{\partial x_1 \partial x_2}\,g + \frac{\partial f}{\partial x_1}\frac{\partial g}{\partial x_2} + \frac{\partial f}{\partial x_2}\frac{\partial g}{\partial x_1} + f\,\frac{\partial^2 g}{\partial x_1 \partial x_2} \tag{6}$$

When a Rall number is a constant, its data members are initialized with the following values (see Table 1):

```
x = constant value;   v = [0,0];   m = [[0,0],[0,0]]
```

When a Rall number represents one of the existing variables (x_1 or x_2 in this example), its initialization becomes:

```
x = value of x₁;   v = [1,0];   m = [[0,0],[0,0]]
  or
x = value of x₂;   v = [0,1];   m = [[0,0],[0,0]]
```

Table 1: Abridged definition and implementation of the class `CRall`.

```
class CRall {
  double x; // Operand value
  double v[2]; // df/dx1, df/dx2
  double m[2][2]; // d2f/dxi dxj
public:
  CRall CRall::operator* (const CRall & g) const {
    CRall t;
    t.x = x * g.x;

    t.v[0] = v[0]*g.x + x*g.v[0];
    t.v[1] = v[1]*g.x + x*g.v[1];

    t.m[0][0] = m[0][0]*g.x+v[0]*g.v[0]+v[0]*g.v[0]+x*g.m[0][0];
    t.m[0][1] = m[0][1]*g.x+v[0]*g.v[1]+v[1]*g.v[0]+x*g.m[0][1];
    t.m[1][0] = m[1][0]*g.x+v[1]*g.v[0]+v[0]*g.v[1]+x*g.m[1][0];
    t.m[1][1] = m[1][1]*g.x+v[1]*g.v[1]+v[1]*g.v[1]+x*g.m[1][1];

    return t;
  }
};
```

All the other operations involving Rall numbers are implemented in a similar manner (e.g., addition, subtraction, division, exponentiation, trigonometric functions, logarithm). In the current version of the optimization software, vectors and matrices are stored in sparse arrays.

5 Expression parser

When a computer program needs to evaluate an expression, the most straightforward strategy is to hard code the expression before the compilation of the module. Whenever the expression has to be modified, a new compilation is required. The alternative of coding an expression parser has also some disadvantages, such as code complexity and a decreased performance. An object oriented programming language (e.g., C++) can lighten the task of coding and reusing complex algorithms, by means of supplying tools that allow for an higher abstraction level. Examples of such techniques are operator overloading, templates, inheritance and polymorphism [2].

The expression parser described in this Section is based on the work of Rogers [6]. Some enhancements 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 scaling techniques (see Section 6).

Three types of entities can be extracted from an expression: constants, variables and operators. According to the priority rules of the parentheses and operators, all the objects are inserted in a binary tree (see Figure 1). A symbol table is an independent object where the names of all the variables and their values are stored. The evaluation of the root object causes a postorder traversal of the binary tree. When a variable is found, the supplied symbol table is searched and the corresponding value is extracted. These operations produce the result of the evaluation of the whole expression and are exemplified with eqn (7) (see also Figure 1).

$$(x_1 + 8\,x_2)\,/\,(6 - x_3^2) \tag{7}$$

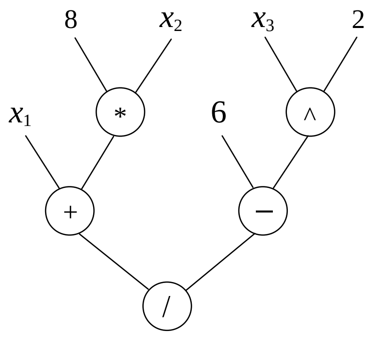

Figure 1: Binary tree obtained from eqn (7).

In the tree shown in Figure 1, the terminal nodes are replaced with Rall numbers (see Section 4). The operations between Rall numbers are subject to the same priority rules that are used in the evaluation of the expression. Since the Rall numbers also operate with the gradient and Hessian of the function, their numerical values are simultaneously evaluated. In order to avoid the systematic manipulation of all the variables that are present in an optimization problem, a data member is used to store the list of variables that need to be considered in each function.

Figure 2 shows the hierarchy diagram of the classes that implement the behavior of all the expression components. The base class *Exp_Obj* defines the common interface of all the derived classes. Polymorphism is used to facilitate the manipulation of the expression objects, regardless of their specified type or functionality.

During the construction of the binary tree, the following tasks are performed: read a token, check the type of the token and add the token to a stack. Variables and literals are added to the expression stack and operators are added to the operator stack. Each operator and each parenthesis has its priority level and is processed accordingly.

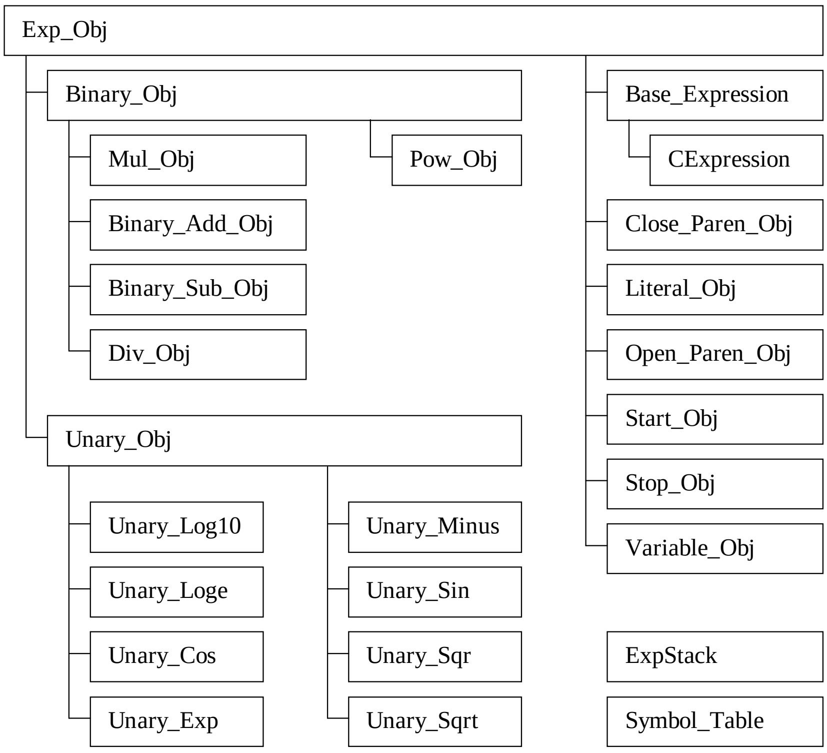

Figure 2: Hierarchy diagram of the expression parser.

6 Scaling

In a mathematical program several types of variables appear in the objective function and constraints. It is common to have variables and function values in different units, exhibiting dissimilar orders of magnitude (e.g., Young’s modulus and rotation angle). When this happens the optimization algorithms may experience numerical instabilities, slow convergence or even a global failure. These problems are caused by round off errors, due to the limited precision of the computer calculations and due to a high condition number in some matrices. Two simple techniques may be implemented to alleviate these problems, namely variable scaling and constraint normalization. Variable scaling can be performed by means of the replacement of each variable x_i by the product $Z_i\, y_i$ where Z_i is the scaling factor (usually the initial value of x_i), and y_i is the new variable in the mathematical program, having a unitary initial value. Constraint normalization consists in the multiplication of the objective function and each constraint by a constant, whose value sets the initial Euclidean norm of the gradient equal to one. These techniques are illustrated with the nonlinear program (8).

$$\begin{aligned}
&\textit{Min.}\ 2000\, x_1 \\
&\quad \textit{subject to} \\
&\qquad -x_1 + 200 + x_3^2 = 0 \\
&\qquad x_2 - 0.2 + x_4^2 = 0 \\
&\qquad -10\, x_1\, x_2 + 500 = 0
\end{aligned} \tag{8}$$

Assuming the initial solution (500, 0.1, √300, √0.1), and applying the aforementioned techniques, (8) is replaced by the NLP (9) [3].

$$\begin{aligned}
&\textit{Min.}\ y_1 \\
&\quad \textit{subject to} \\
&\qquad -0.640\, y_1 + 0.256 + 0.384\, y_3^2 = 0 \\
&\qquad 0.447\, y_2 - 0.894 + 0.447\, y_4^2 = 0 \\
&\qquad -0.707\, y_1\, y_2 + 0.707 = 0
\end{aligned} \tag{9}$$

The scaled version of the mathematical program is numerically more stable and has a faster convergence [3]. The solution of the original NLP (8) can be recovered from the solution of (9).

The implementation of the scaling of the variables implied the inclusion of a new data member in each entry of the symbol table in order to store the scaling factor. The constraint normalization factor is stored in a data member of the class *Base_Expression* (see Figure 2). The computation of derivatives has to take into account these modifications of the expressions.

7 Numerical example

A simple shape optimization problem is presented in order to illustrate the main characteristics of the proposed algorithm. It consists on the minimization of the weight of a two bar truss, subjected to a single load case (see Figure 3) [7]. Dead load is not considered and buckling may be ignored due to the fact that both bars are under tension. The nonlinear program describing the optimization problem is defined in (10).

$$\begin{aligned}
&Min.\ \ w(x_1,x_2) = C_1\, x_1 \sqrt{1+x_2^2} \\
&\quad subject\ to \\
&\qquad \sigma_1(x_1,x_2) = C_2 \sqrt{1+x_2^2}\left(\frac{8}{x_1}+\frac{1}{x_1\,x_2}\right) \le 1 \\
&\qquad \sigma_2(x_1,x_2) = C_2 \sqrt{1+x_2^2}\left(\frac{8}{x_1}-\frac{1}{x_1\,x_2}\right) \le 1 \\
&\qquad 0.2 \le x_1 \le 4.0 \ \ ; \ \ 0.1 \le x_2 \le 1.6
\end{aligned} \tag{10}$$

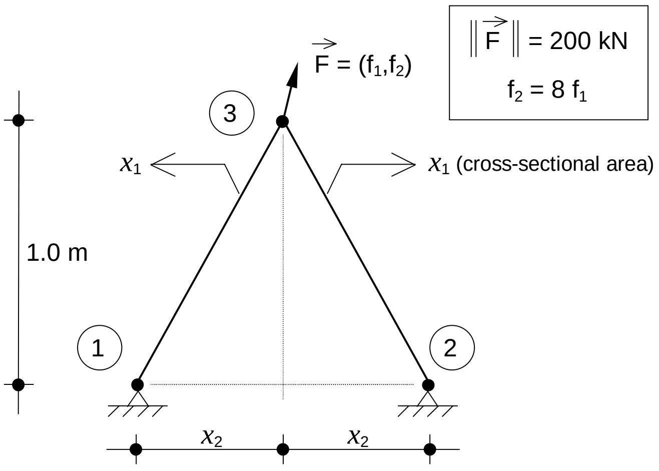

Figure 3: Shape optimization of a two bar truss.

In the nonlinear program (10), $C_1 = 1.0$ and $C_2 = 0.124$. These values are not modified during the iteration process. The initial values of x_1 and x_2 are 1.5 cm^2 and 0.5 m respectively. Table 2 shows the syntax of the input file of the optimization program, containing the description of the nonlinear program (10). The last four lines declare the type of each variable, its initial value and its name.

Table 2: Input file of the optimization program.

```
# Main title
Shape optimization of a two bar truss
# N. of eq. constr.; N. of ineq. constr.
                0                     6
# Objective Function
  C1*x1*sqrt(1+x2^2);
# Allowable stress - bar 1
  C2*sqrt(1+x2^2)*(8/x1+1/x1/x2)-1;
```

Table 2 (cont.): Input file of the optimization program.

```
# Allowable stress - bar 2
  C2*sqrt(1+x2^2)*(8/x1-1/x1/x2)-1;
# Minimum x1
  -x1+0.2;
# Maximum x1
  x1-4.0;
# Minimum x2
  -x2+0.1;
# Maximum x2
  x2-1.6;
# N. of variables
  4
SUBSTITUTED, 1.000, C1;
SUBSTITUTED, 0.124, C2;
INDEPENDENT, 1.5, x1;
INDEPENDENT, 0.5, x2;
```

Figure 4 shows the iteration history of the objective function whose optimal value is $w = 1.509$. The intermediate solutions exhibiting objective function values that are smaller than the optimum correspond to infeasible points. Since the Lagrange-Newton method is based on the search of a saddle point of the Lagrangian, there is no guarantee that the optimum be approached from the interior of the feasible region. The optimal value of the independent design variables is $x_1 = 1.412$ cm^2 and $x_2 = 0.377$ m.

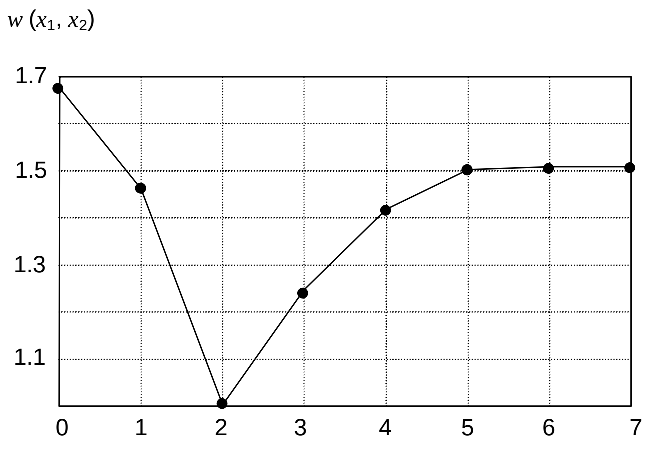

Figure 4: Iteration history – objective function *w*.

Several optimization problems described in the books of Arora [8] and Azevedo [3] were solved with this computer code and the results were compared with the solutions obtained with other computer programs. Since a good agreement was always obtained, the proposed algorithm and the complex C++ coding have been validated.

8 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 are 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.

References

[1] Barton, J. J., Nackman, L. R. Automatic differentiation. *C++ Report*, pp. 61-63, February 1996.
[2] Stroustrup, B. *The C++ Programming Language*, Third Edition, Addison-Wesley, 1997.
[3] Azevedo, A. F. M. *Optimization of Structures with Linear and Nonlinear Behavior*, Ph.D. Thesis (in Portuguese), Faculty of Engineering, University of Porto, Portugal, 1994.
[4] Azevedo, A. F. M., Second-order structural optimization, *Computer Aided Optimum Design of Structures IV-OPTI 95*, eds. S. Hernandez, M. El-Sayed & C. A. Brebbia, Computational Mechanics Publications, pp.67-74, 1995.
[5] Nash, S. G., Sofer, A. *Linear and Nonlinear Programming*, McGraw-Hill, 1996.
[6] Rogers, J. An object oriented expression evaluator. *C/C++ Users Journal*, pp. 43-51, April 1996.
[7] Svanberg, K. The method of moving asymptotes-a new method for structural optimization. *International Journal for Numerical Methods in Engineering*, **24**, pp. 359-373, 1987.
[8] Arora, J. *Introduction to Optimum Design*, McGraw-Hill, 1989.