2012 4th International Conference on Computer Engineering and Technology (ICCET 2012)
IPCSIT vol.40 (2012) © (2012) IACSIT Press, Singapore

Towards the Application of Algorithmic Differentiation on the NAG Library

Bui Ngoc Linh [1], Johannes Lotz [2], Jan Riehme [3], and Uwe Naumann [4]

[1] The Sirindhorn International Thai-German Graduate School of Engineering (TGGS), King Mongkut's University of Technology North Bangkok (KMUTNB), Bangkok, Thailand

[2, 3, 4] LuFG Informatik 12: Software and Tools for Computational Engineering, RWTH Aachen University, Aachen, Germany

Abstract. Algorithmic differentiation is a growing area of research that is concerned with the accuracy and efficient evaluation of derivatives for numerical functions defined by computer programs. In this paper, algorithmic differentiation by overloading is introduced and applied to the linear solver F07AEF of the NAG library. As a case study and for the verification purpose, its result is used to compute the sensitivity of the Newton method in solving a nonlinear system of equation generated by a minimum problem. The function chosen for the case study is Extended Rosenbrock function.

Keywords: Algorithmic differentiation, Tangent-linear model, Algorithmic differentiation by overloading, Linear solver, Newton method.

1. Introduction

The sensitivity of function values with respect to parameters is the requirement rising frequently in many fields of applications. Differentiation can give reliable answers.

Typical methods for computing derivatives are finite difference (FD) and algorithmic differentiation (AD)[1],[6]. While the traditional method FD is simple, easy to use but lack of accuracy in many cases, AD is able to compute derivatives of implemented mathematical functions up to machine accuracy.

Based on the fact that every computer program from a simple to a complicated one is a combination of elementary arithmetic operations, AD is a set of techniques that apply the chain rule of differential calculus to calculate the partial derivative of each of these operations. AD contains two modes: forward mode (tangent-linear model) and reverse mode (adjoint model). Using AD, derivative code can be generated either by source transformation, meaning each statement is augmented with one more statement to calculate its derivative, or by overloading technique in supported overloading programming languages such as C++ and FORTRAN. In this paper we will focus on the forward mode AD by overloading in FORTRAN language, concerning its later advantages in many complex computations.

In the field of computational software, the NAG library - a product of Numerical Algorithms Group (NAG) company[1] - is one of the libraries widely used for computational applications. The NAG library provides a lot of procedures, which cover a wide range of mathematical and statistical subroutines such as optimization, linear, quadratic, integer and nonlinear programming. It is available in C and FORTRAN. Until now, the NAG library does not have any differentiated version. This paper will present an approach to generate derivative code by overloading, where the underlying program calls an external procedure from the NAG library. Although the NAG library is not distributed freely, due to the collaboration between LuFG Informatik 12 and the NAG company, the source code is provided.

[1]http://www.nag.com/

In the field of application, assume that we have a nonlinear equation of unique solution $y = f(\boldsymbol{x}, \lambda)$ with some parameters λ and it is solved by Newton method. At the solution $\boldsymbol{x}^*$ we have $f(\boldsymbol{x}^*, \lambda) = y^*$. Here we can see the solution $\boldsymbol{x}^*$ as a function of parameters λ: $\boldsymbol{x}^* = S(\lambda)$. What we are interested in is how much the solution will change if λ is changed, meaning the sensitivities of $\boldsymbol{x}^*$, y^* with respect to the parameter λ: $\frac{\partial S(\lambda)}{\partial \lambda}$, $\frac{\partial f(\boldsymbol{x}^*, \lambda)}{\partial \lambda}$.

In more detail, provided that $f(\boldsymbol{x}, \lambda)$ is twice-differentiable function, taking a closer look at the Newton algorithm [2] for solving problem of an unconstrained nonlinear equation $f(\boldsymbol{x}, \lambda) = y$:

Algorithm 1 Newton algorithm

```
1:  y = f(x, λ)
2:  while ‖ y ‖> ϵ do
3:      (y, J) = f'(x, λ)
4:      dx = s(y, J)
5:      x = x − dx
6:      y = f(x, λ)
7:  end while
```

in which, J is Jacobian of y, we can see that the statement in line 4 of the algorithm is a linear equation and composes the most expensive part of the algorithm. We therefore choose an external library to solve that equation. Linear solver F07AEF [4] of the chapter F07 [3] of the NAG library about linear equations is of the choice for its popularity in using. Motivated by that, this paper will operate differentiation on F07AEF procedure.

This paper is organized as follows: Section 2 is basic knowledge about AD and AD by overloading. In section 3, the implementation of this technique for F07AEF procedure is introduced. Section 4 brings out a case study as an application for verification.

2. Algorithmic differentiation by overloading

As mentioned in the introduction section, basically AD provides two modes, the forward mode (tangent-linear model (TLM)) and the reverse mode (adjoint model (AM)).

For a vector function F: $\mathbb{R}^n \to \mathbb{R}^m$, $y = F(\boldsymbol{x})$, the forward mode propagates the directional derivative of **y** with respect to **x** according to :

$$\dot{\boldsymbol{y}} \equiv \nabla F(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}} . \tag{1}$$

In which ∇F is Jacobian of F. That is, for a given input $\boldsymbol{x} \in \mathbb{R}^n$ and a direction $\dot{\boldsymbol{x}} \in \mathbb{R}^n$, a differentiated version of a program implementing $\boldsymbol{y} = F(\boldsymbol{x})$, computes the function value **y** as well as the directional derivative $\dot{\boldsymbol{y}}$.

An example: Having a function F: $\mathbb{R}^2 \to \mathbb{R}$, $y = F(\boldsymbol{x}) = x_1 \cdot x_2^2$ with $\boldsymbol{x} \equiv (x_1, x_2) = (1,2)$, $(\dot{v}_1, \dot{v}_2) = (1,0)$, the tangent-linear model is computed as:

$$\begin{array}{llllll}
\dot{v}_1 & = \dot{x}_1 & = 1\,; v_1 & = x_1 & = 1 \\
\dot{v}_2 & = \dot{x}_2 & = 0\,; v_2 & = x_2 & = 2 \\
\dot{v}_3 & = 2v_2 \cdot \dot{v}_2 & = 0\,; v_3 & = v_2^2 & = 4 \\
\dot{v}_4 & = v_3 \cdot \dot{v}_1 + v_1 \cdot \dot{v}_3 & = 4\,; v_4 & = v_1 \cdot v_3 & = 4 \\
\dot{y} & = \dot{v}_4 & = 4\,; F(\boldsymbol{x}) & = v_4 & = 4.
\end{array}$$

For more detail about forward mode by AD, see [6].

The concept of overloading has brought up an effective method to compute derivative. In the context of AD, overloading means to use the same arithmetic operators and intrinsic functions of a programming language but to operate on new data types. Concretely, a new data type is defined containing one more derivative variable in addition to the original variable. E.g. If the variable name v, then the new data type will contain $(v, \dot{v})$. All operators and intrinsic functions to calculate values are overloaded by routines with the same names but to calculate both values and values' derivative. For example: With three floating point

parameters a, b, c, operator *multiple* to calculate $c = a \cdot b$ is changed to operator *multiple* to calculate $c = a \cdot b$ and $\dot{c} = \dot{a} \cdot b + a \cdot \dot{b}$ in overloading mode. We notice that the function value is computed simultaneously with the derivative propagation.

One of the obvious advantage of this method is it requires no change in the sequence of operation of the original source code; you have to change the data type of operations only.

LuFG Informatik 12 (Software and Tools for Computational Engineering) at RWTH Aachen University, Germany has introduced a tool called dco/f (derivative code by overloading/FORTRAN) for implementing ADby overloading in FORTRAN language [5]. In this paper, we will use the tangent-linear module of this tool for the implementation.

3. Implementation

F07AEF procedure is used to solve real systems of linear equations with multiple right-hand sides $AX = B$ or $A^T X = B$ where A, X, B are matrices. Each column in X and B corresponds to a linear system $A\boldsymbol{x} = \boldsymbol{b}$. In this procedure, matrix A must have been factorized by F07ADF in advance. We need to implement derivative of F07AEF.

A procedure in the NAG library usually contains a complex subroutines structure. Hence, as the first step in the implementation we need to extract the structure of F07AEF. The internal structure of F07AEF is a tree-call of 71 other procedures, any intervention to one file might affect the whole structure.

The next step to compute AD by overloading is to change the data type of active variables to the new type. This changing of all active variables of the type REAL or DOUBLE is required to do in all 71 procedures. Applying the dco/f stated in (1), we shift all the type of REAL(KIND = WP), REAL(KIND = RP) to the type of (DCOF_T1S_TYPE). DCOF_T1S_TYPE means the first tangent-linear derivative type. We also include in the header of each interface a reference USE the module. For example, a change in a procedure of the tree-call is as follows:

Listing 1: Original code

```
1   ! . . Use Statements . .
2   USE NAG_ENUM_CONSTS, ONLY :
        NAG_COL MAJOR
3   USE NAG_PRECISIONS , ONLY : WP
4   ! . . Implicit None Statement. .
5   IMPLICIT NONE
6   ! . . Parameters . .
7   INTEGER, PARAMETER : : SORDER
            = NAG_COL_MAJOR
8   CHARACTER (*) , PARAMETER : :
    SRNAME = 'F06YAF /DGEMM'
9       ! . . Scalar Arguments . .
10  REAL (KIND=WP) , INTENT ( IN ) : ALPHA,
    BETA
11
```

Listing 2: Data type changed

```
1   ! . . Use Statements . .
2   USE NAG_ENUM_CONSTS, ONLY :
            NAG_COL_MAJOR
3   USE NAG_PRECISIONS , ONLY : WP
4     USE DCOF_T1S
5     ! . . Implicit None Statement. .
6   IMPLICIT NONE
7       ! . . Parameters . .

8   INTEGER, PARAMETER : : SORDER
            = NAG_COL_MAJOR
9   CHARACTER (*) , PARAMETER  : : SRNAME
            = 'F06YAF /DGEMM'
10      ! . . Scalar Arguments . .
11  TYPE (DCOF T1S TYPE) , INTENT ( IN ) : :
        ALPHA, BETA
```

After that, we need to compile and link those new files together. For the convenience of compiling, all 71 files are put into one folder. Moreover, these files have to place in the right order in compiling. Any wrong placing will make the build failed. We have to explicitly point out the directory to the dco/f library.

4. Case study

As the underlying problem solved by the Newton method, we choose the n dimensional Extended Rosenbrock function, which is original from the test problem Rosenbrock [7] with inputs $x_i\ (i = 1, \ldots, n)$, and parameters $\lambda_i\ (i = 1, \ldots, n-1)$:

$$y = f(\boldsymbol{x}, \lambda) = \sum_{i=1}^{n-1} [\,(\lambda_i - x_i)^2 + 100 \cdot \lambda_i^2 (x_{i+1} - x_i^2)^2] \tag{2}$$

From the Eq.2, the problem solved by Newton algorithm is: $min_{x \in \mathbb{R}^n} f(\boldsymbol{x}, \lambda)$.

This section will apply the derivative version of the linear solver F07AEF computed in the implementation part in computing the sensitivity of $f(\boldsymbol{x})_{min}$ of Extended Rosenbrock function with respect to λ.

By solving $f(\boldsymbol{x})_{min}$ by the Newton algorithm - which calls the external linear solver F07AEF - we have $y^* = f(\boldsymbol{x}^*)$ as the solution. We need the sensitivity: $\frac{\partial \boldsymbol{x}^*}{\partial \lambda i}$, meaning we differentiate the Newton algorithm. This is where we plug the derivative version of the linear solver F07AEF into. To verify the result, a version of the derivative of linear solver retrieved by FD is also computed.

With n = 5, we analyze the result of $\frac{\partial x_1^*}{\partial \lambda}$ by FD and AD with different perturbation h. The formula of FD is central FD: $\frac{\partial y}{\partial x} \approx \frac{\boldsymbol{x}^*(\lambda+h) - \boldsymbol{x}^*(\lambda-h)}{2.h}$. The graph draws the difference between those results. In the graph, the horizontal axis corresponds to different e where $h = 10^{-e}$ (h ranges from 10^{-1} to 10^{-13}). The same type of graph goes for other values of derivative $\frac{\partial x_i^*}{\partial \lambda i}$.

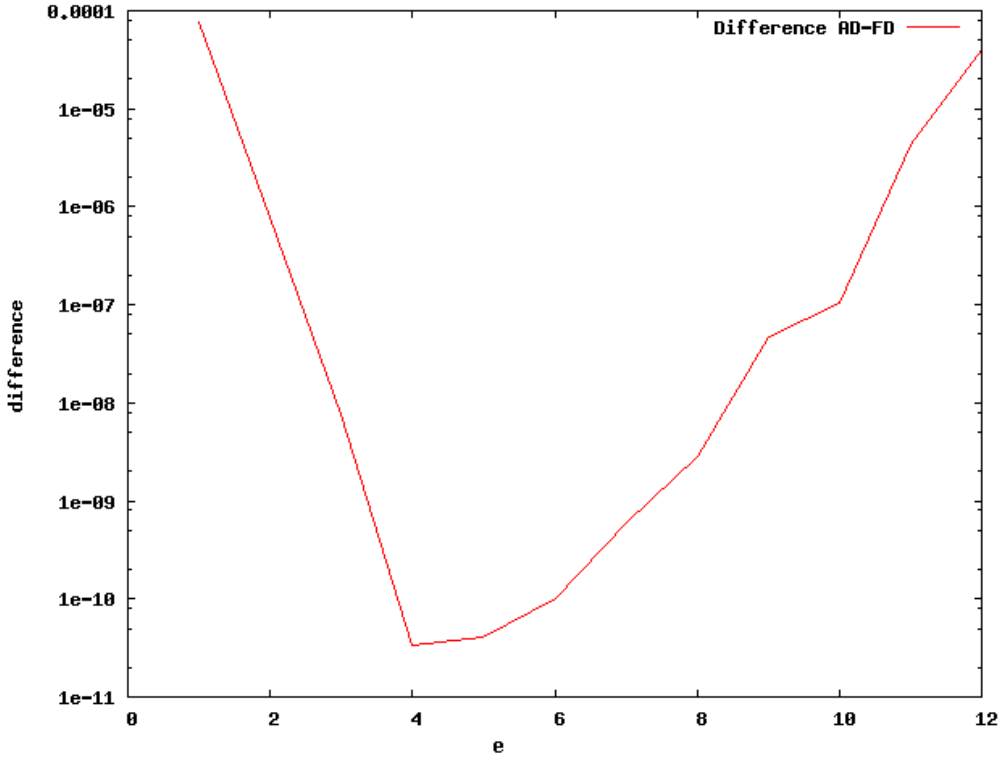

Fig 1: The difference between AD-FD

As can be seen from the graph, the difference is the smallest when $h = 10^{-5}$, meaning at that point FD gave the best result in compare with AD by overloading. The result is depended on perturbation h is one of the disadvantages of FD because we could see the difference become more significant when the perturbation h changed. In many cases such as $h = 10^{-12}$, FD may no longer be trusted.

5. Conclusion

Overall, AD by overloading is a convenient and reliable way to implement derivative for the NAG library. It calculates the derivative result with about the same accuracy as the original function.

By applying the same method of computing AD by overloading for F07EAF in this paper, we could yield the derivative version of other procedures in the NAG library.

For the future work, based on the paper's implementation, we can exploit the reverse mode by overloading and use it interchangeably with forward mode.

6. References

[1] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithm differentiation*, Society for Industrial and Applied Mathematics (SIAM), 2008.

[2] C. Kelly, *Solving Nonlinear Equations with Newton's Method*. SIAM, 2003.

[3] NAG Library Manual-Mark 23, *Nag library chapter introduction, f07 linear equations (lapack),* 2011.

[4] NAG Library Manual-Mark 23, *Nag library routine document, f07aef(dgetrs),* 2011.

[5] U. Naumann and J. Riehme, *A differentiation-enabled Fortran 95 compiler*, ACMTransactions on Mathematical Software 31 (2005), no. 4, 458–474.

[6] U. Naumann, *The Art of Differentiating Computer Programs*. Society for Industrial and Applied Mathematics (SIAM), 2011.

[7] Eric W Weisstein, *Rosenbrock function. from mathworld–a wolfram web resource*., http://mathworld.wolfram.com/RosenbrockFunction.html.