NPL Report CMSC 26/03

Report to the National Measurement Directorate, Department of Trade and Industry

From the Software Support for Metrology Programme

Automatic Differentiation Techniques and their Application in Metrology

By
R Boudjemaa, M G Cox, A B Forbes and P M Harris

June 2003

Automatic Differentiation Techniques and their Application in Metrology

R Boudjemaa, M G Cox, A B Forbes and P M Harris
Centre for Mathematics and Scientific Computing

June 2003

ABSTRACT

This report describes Automatic Differentiation techniques and their use in metrology. Automatic Differentiation (AD) is a term used in general to describe the various numerical techniques for computing the derivatives of a function of one or more variables. While the rules for differentiation are straightforward to understand, their implementation by hand is often time consuming and error prone. AD techniques help to overcome both these drawbacks and offer the user involved in nonlinear modelling and optimisation the promise of accurate and correct derivative information with seemingly minimal effort. In this report we describe the main AD techniques and show how they can be applied in a number of areas important to metrology.

ISSN 1471–0005

Authorised by Dr Dave Rayner,
Head of the Centre for Mathematics and Scientific Computing

National Physical Laboratory,
Queens Road, Teddington, Middlesex, United Kingdom TW11 0LW

Contents

1 Introduction **1**

2 Nonlinear problems in data analysis in metrology **3**
2.1 Nonlinear least squares approximation 3
2.2 Uncertainty evaluation 5
2.3 Maximum likelihood estimation and numerical optimisation . 5
2.4 Function approximation 6
2.5 Requirements for automatic differentiation 6

3 Analytical methods **8**
3.1 Description 8
3.2 Advantages 8
3.3 Disadvantages 8

4 Symbolic differentiation **9**
4.1 Description 9
4.2 Advantages 9
4.3 Disadvantages 9

5 Finite differences **10**
5.1 Description 10
5.2 A quick check of derivatives using finite differences 11
5.3 Second derivatives 12
5.4 Advantages 14
5.5 Disadvantages 14

6 The complex step method **15**
6.1 Description 15
6.2 Second derivatives 20
6.3 Advantages 21
6.4 Disadvantages 21
6.5 Example application: nonlinear regression 21

7 Program differentiation techniques **24**

8 Forward Automatic Differentiation **25**
8.1 Description 25
8.2 Second derivatives 28
8.3 Advantages 28
8.4 Disadvantages 29

9 Reverse Automatic Differentiation **30**

9.1 Description 30
9.2 Second derivatives 32
9.3 Advantages 33
9.4 Disadvantages 33
9.5 Example application: maximum likelihood estimation 33

10 Source to source **35**
10.1 Description 35
10.2 Advantages 35
10.3 Disadvantages 36

11 Numerical examples **37**
11.1 Example 1: $\sin 1/x$ 37
11.2 Example 2: $x^{-1} \cos x^{-1}$ 37
11.3 Gradients associated with maximum likelihood estimation . . 39

12 Conclusions and recommendations **43**

References **45**

1 Introduction

There are a wide variety of mathematical and scientific problems in which it is necessary to obtain accurate derivative evaluations. We recall that if $f(x)$ is a function of a variable x, the derivative

$$\frac{df}{dx}(x) \quad \text{or} \quad f'(x)$$

is also a function of x and represents the slope of (the tangent to) f at x. Mathematically, the derivative is defined as a limit

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{1}$$

The fraction on the right represents the slope of the line passing through $(x, f(x))$ and nearby point $(x+h, f(x+h))$. For a function of a number of variables $f(x_1, \ldots, x_n)$, the partial derivative of f with respect to x_j is defined as

$$\frac{\partial f}{\partial x_j} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_j + h, \ldots, x_n) - f(x_1, \ldots, x_j, \ldots, x_n)}{h}.$$

The differentiation process can be repeated any number of times, e.g.,

$$f'' = f^{(2)} = \frac{d^2 f}{dx^2} = \frac{d}{dx}\left(\frac{df}{dx}\right), \quad \frac{\partial^2 f}{\partial x_j \partial x_k} = \frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial x_k}\right),$$

but most applications do not require more than first or second derivatives. The derivatives of common functions can be detemined from first principles from the definition (1), e.g.,

$$\frac{d}{dx} x^n = n x^{n-1}, \quad \frac{d}{dx} \sin x = \cos x.$$

The rules for differentiating sums, products, quotients, etc., of functions are straightforward to derive and reasonably easy to remember. For example, given two functions $f(x)$ and $g(x)$, then

$$(f+g)' = f' + g', \quad (fg)' = f'g + fg'.$$

Using these rules, the derivatives of complex functions can be expressed in terms of the derivatives of their simpler, component functions. However, even for only moderately complicated functions, the hand calculation of derivatives can lead to pages of tedious algebra with an increasing likelihood of a mistake entering into the computation.

If the function evaluation is encoded in a software component, it is natural to ask if it possible to compute the derivatives automatically using the function evaluation component. Until recently, the standard method of evaluating derivatives numerically was to use finite differences, essentially evaluating the right-hand side of (1) with h a small, preassigned nonzero number. This approach generally gives an approximate value. In recent years, advances in computers and computer languages have allowed the development of a new method for obtaining accurate derivatives of any programmable function. The term *automatic differentiation* (AD) [17] generally applies to techniques that produce, from a function evaluation software component, a computational scheme, also implemented in software, that calculates the derivatives. These techniques have evolved and are still evolving both in terms of their theoretical basis and, more markedly, in the software engineering aspects of their implementation. Reasonably mature AD software implementations appeared in the early 1990's [3] and the process now comes in two 'flavours', *reverse automatic differentiation* [19] and *forward automatic differentiation* [3].

In a separate development, attention is now being paid to the *complex step method* which uses complex arithmetic to evaluate accurate derivatives. The method is particularly easy to implement in Matlab [24], largely because the Matlab default variable type is complex. In this environment, it has been used successfully at NPL in many applications.

In this report we describe the main techniques for function differentiation, giving a summary of their advantages and disadvantages in terms of both accuracy and easy of implementation. The remainder of this report is organised as follows. In section 2, we give a number of metrological application areas in which the calculation of derivatives is required. In sections 3–10, we give descriptions of the main techniques for calculating derivatives and summarise their advantages and disadvantages. Numerical examples are given in section 11. Our concluding remarks and recommendations are presented in section 12.

2 Nonlinear problems in data analysis in metrology

In this section, we consider some of the main areas where derivative evaluation is required in metrology. These include nonlinear least squares approximation, uncertainty evaluation, maximum likelihood estimation and function approximation.

2.1 Nonlinear least squares approximation

In many calibration problems, the observation equation involving measurement y_i can be expressed as $y_i = \phi_i(\mathbf{a}) + \epsilon_i$, where ϕ_i is a function depending on parameters $\mathbf{a} = (a_1, \ldots, a_n)^{\mathrm{T}}$ that specify the behaviour of the instrument and ϵ_i represents the mismatch between the model specified by $\mathbf{a}$ and the measurement y_i. Given a set of measurement data $\{y_i\}_1^m$, estimates of the calibration parameters $\mathbf{a}$ can be determined by solving

$$\min_{\mathbf{a}} F(\mathbf{a}) = f_1^2(\mathbf{a}) + f_2^2(\mathbf{a}) + \ldots + f_m^2(\mathbf{a}) = \sum_{i=1}^{m} f_i^2(\mathbf{a}), \tag{2}$$

where $f_i(\mathbf{a}) = y_i - \phi_i(\mathbf{a})$. Least squares techniques are appropriate for determining parameter estimates for a broad range of calibration problems [10].

A common approach to solving least squares problems (2) is the Gauss-Newton algorithm. If $\mathbf{a}$ is an estimate of the solution and J is the *Jacobian matrix* defined at $\mathbf{a}$ by $J_{ij} = \partial f_i / \partial a_j$, then an updated estimate of the solution is $\mathbf{a} + \alpha\mathbf{p}$, where $\mathbf{p}$ solves the Jacobian system

$$J\mathbf{p} = -\mathbf{f}, \tag{3}$$

in the least squares sense, i.e., $\mathbf{p}$ solves

$$\min_{\mathbf{p}} \|J\mathbf{p} + \mathbf{f}\|^2, \tag{4}$$

and α is a step length chosen to ensure that adequate progress to a solution is made. Starting with an appropriate initial estimate of $\mathbf{a}$, these steps are repeated until convergence criteria are met. From the optimality conditions associated with (4) we can derive the *normal equations*

$$J^{\mathrm{T}}J\mathbf{p} = -J^{\mathrm{T}}\mathbf{f}, \tag{5}$$

which determines $\mathbf{p}$ as the solution of a square system of equations. It should be noted that we do not necessarily recommend finding $\mathbf{p}$ by solving the

normal equations. Rather, we suggest approaches based on matrix factorisation techniques [10, 16]. Standard library software for solving nonlinear least squares problems requires the user to supply a component FGUSER to calculate the function values f_i and the Jacobian matrix J of partial derivatives. The solver controls the iterative scheme using the user-supplied component to provide function and gradient information from which the Gauss-Newton step is calculated.

A typical specification of the user-supplied component in Fortran 90 [25] might look like:

```
  subroutine fguser(imode,m,n,a,f,j)
! ------------------------------------------------
! Function and Jacobian evaluation for least squares
! ------------------------------------------------
  integer, intent(inout) :: imode
  integer, intent(in)    :: m,n
  real(8), intent(in)    :: a(n)
  real(8), intent(out)   :: f(m),j(m,n)
! ------------------------------------------------
! Parameter description
! ---------------------
! IO imode     INTEGER
!              If imode = 1, fguser is required to calculate
!                            function values f.
!                       = 2, fguser is required to calculate
!                            Jacobian matrix J.
!
! IN m         INTEGER
!              Number of functions f_i.
!
! IN n         INTEGER
!              Number of optimisation parameters.
!
! IN a         DOUBLE PRECISION 1-dimensional array
!              n x 1
!              Estimate of parameters.
!
! OU f         DOUBLE PRECISION 1-dimensional array
!              m x 1
!              Functional values.
!
! OU j         DOUBLE PRECISION 2-dimensional array
!              m x n
!              Jacobian matrix.
!              If imode = 2 on entry, then on exit
!              j(i,k) should contain the partial
!              derivative of the ith function
!              with respect to the jth parameter.
```

In Matlab [24], the function and gradient evaluation component may have

a specification of the form:

```
  [f, J] =  fguser(a)
% ------------------------------------------------
% Function and Jacobian evaluation for least squares
% ------------------------------------------------
% Parameter description
% ---------------------
% IN a          1-dimensional array
%               n x 1
%               Estimate of parameters.
%
% OU f          1-dimensional array
%               m x 1
%               Functional values.
%
% OU J          2-dimensional array
%               m x n
%               Jacobian matrix.
```

2.2 Uncertainty evaluation

If $U_{\mathbf{a}}$ is the uncertainty (covariance) matrix associated with a set of parameters $\mathbf{a}$ and $f(\mathbf{a})$ is a function of $\mathbf{a}$ then the standard uncertainty $u(f)$ associated with f is estimated from

$$u^2(f) = \mathbf{g}^{\mathrm{T}} U_{\mathbf{a}} \mathbf{g}, \tag{6}$$

where $\mathbf{g}$ is the gradient vector of f, i.e., the vector of first partial derivatives of f with respect to the parameters a_j [8, 2]. These partial derivatives are often referred to as *sensitivity coefficients*. If the uncertainty matrix is diagonal with

$$U_{ii} = u_i^2, \quad U_{ij} = 0, \quad i \neq j,$$

then (6) simplifies to the perhaps more familiar

$$u^2(f) = \left(\frac{\partial f}{\partial a_1}\right)^2 u_1^2 + \ldots + \left(\frac{\partial f}{\partial a_n}\right)^2 u_n^2 = \sum_{j=1}^{n} \left(\frac{\partial f}{\partial a_j}\right)^2 u_j^2.$$

2.3 Maximum likelihood estimation and numerical optimisation

Least squares approximation methods are a form of maximum likelihood estimation in which the most likely explanation of the data is sought. For preassigned statistical models in which uncertainty in the measurement data is modelled by Gaussian distributions with a known uncertainty (covariance)

matrix, the maximum likelihood corresponds to minimising a sum of squares [10]. For more general models in which the uncertainty matrix is not known exactly, the maximum likelihood is attained by minimising a function of the form

$$F(\mathbf{a}, \boldsymbol{\sigma}) = h(\boldsymbol{\sigma}) + \sum_{i=1}^{m} f_i^2(\mathbf{a}, \boldsymbol{\sigma}),$$

where $\boldsymbol{\sigma}$ are additional optimisation parameters that specify the unknown input uncertainty matrix [13]. In this more general setting, F is no longer a sum of squares and the Gauss-Newton algorithm can no longer be applied (at least with the same success).

The Gauss-Newton algorithm is in fact derived from the more general Newton's algorithm for minimising a function $F(\mathbf{a})$. In the Newton algorithm, the search direction $\mathbf{p}_N$ is computed as the solution of

$$H\mathbf{p}_N = -\mathbf{g},$$

where $\mathbf{g} = \nabla_{\mathbf{a}} F$ is the gradient vector of first partial derivatives, $g_j = \partial F/\partial a_j$, and $H = \nabla^2_{\mathbf{a}} F$ is the *Hessian* matrix of second partial derivatives

$$H_{jk} = \frac{\partial^2 F}{\partial a_j \partial a_k}.$$

Most optimisation software works best when both first and second derivatives are supplied to the algorithm and in this case the user has to supply one or two components, similar in specification to FGUSER above, to calculate F, the gradient $\mathbf{g}$ and Hessian matrix H. Because of the difficulties in providing these derivatives, many algorithms use some form of approximation scheme to estimate either H or its inverse.

2.4 Function approximation

The Taylor expansion of a function provides a local polynomial approximation in which the coefficients are related to the derivatives of the function:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{3!}f'''(x) + \frac{h^4}{4!}f''''(x) + \ldots \qquad (7)$$

2.5 Requirements for automatic differentiation

In an ideal scenario, we would like our solvers and uncertainty evaluation software to work effectively with function values alone so that the user is only required to supply components of the form:

```
  subroutine fuser(m,n,a,f)
! ------------------------------------------------
! Function evaluation
! ------------------------------------------------
  integer, intent(in)    :: m,n
  real(8), intent(in)    :: a(n)
  real(8), intent(out)   :: f(m),
! ------------------------------------------------
! Parameter description
! ---------------------
! IN m          INTEGER
!               Number of functions f_i.
!
! IN n          INTEGER
!               Number of optimisation paramemeters.
!
! IN a          DOUBLE PRECISION 1-dimensional array
!               n x 1
!               Estimate of parameters.
!
! OU f          DOUBLE PRECISION 1-dimensional array
!               m x 1
!               Functional values.
!
```

in Fortran, or in Matlab:

```
  [f] =  fuser(a)
% ------------------------------------------------
% Function evaluation
% ------------------------------------------------
% Parameter description
% ---------------------
% IN a          1-dimensional array
%               n x 1
%               Estimate of parameters.
%
% OU f          1-dimensional array
%               m x 1
%               Functional values.
%
```

If we could meet this requirement without significant penalty in terms of accuracy or computational efficiency then the main difficulties with working with nonlinear models, etc., would largely disappear. In the sections below we examine the main differentiation techniques used at present and see to what extent they meet this requirement.

3 Analytical methods

3.1 Description

By analytical methods, we mean determining and encoding in software, by hand, analytical expressions for the derivative functions, following the basic rules of calculus. Practically all scientists, engineers, etc., have had to perform these computations and, for many, this type of task is still a significant part of their work from time to time. Tables of derivatives are sometimes useful for special functions.

3.2 Advantages

Although this report is concerned mainly with differentiation methods that do not depend directly on human endeavour, hand-coding has some advantages relative to other methods.

Assuming the formulae for the derivatives have been derived correctly the calculated derivatives will be accurate. In making this statement, we have to be aware that a formula that is correct mathematically may give rise to inaccuracies when evaluated using finite precision arithmetic. We assume that a person aware of the problems of numerical instability will take the same care in formulating expressions for the derivatives as those for the function itself.

The evaluation of the derivatives can be made efficient. With good design, the expressions for the derivatives will exploit the fact that some or many of the terms appear more than once. This is usually achieved by breaking down the function to be evaluated into component functions. The practitioner will also recognise those derivatives that are known to be zero.

Derivatives up to modest order can be generated.

3.3 Disadvantages

For functions with any degree of complexity, the analytical method is time consuming and error prone.

4 Symbolic differentiation

4.1 Description

Symbolic differentiation is part of the more general field of symbolic algebra and has grown in popularity over the past two decades. There are now mature software packages for symbolic algebra such as Mathematica [33] and Maple [21] that include, among many other functions, differential calculus.

Symbolic differentiation begins with an algebraic expression (formula) for the function and produces, from the rules for differentiation and a database of derivatives of special functions, formulae for the derivatives. In this way, the software package is performing essentially the same steps as the practitioner deriving the formulae analytically. The input functions and output formulae can be written or produced in the syntax of standard computer languages so that these formulae can be imported from or included into software.

4.2 Advantages

Once the formula for the function has been entered, all the formulae for the derivatives are calculated automatically and require no human intervention. These formulae can then be incorporated into software.

The formulae for the derivatives are mathematically correct (provided the package implements the rules of calculus correctly).

Derivatives of any order can be generated.

4.3 Disadvantages

The formulae produced by symbolic differentiation will generally involve many more computational steps than well designed, hand-coded formulae. For moderately complex functions, symbolic differentiation tends to produce very unwieldy expressions that represent an inefficient scheme for evaluating the derivatives.

There is no gaurantee that the formulae will represent a numerically stable method of evaluating the derivatives.

Symbolic differentiation requires access to source code for evaluating the function.

5 Finite differences

5.1 Description

Finite-difference methods are still the most commonly used techniques for estimating derivatives for science and engineering applications [28]. The main and simple idea of the method is to approximate the derivative (1) by evaluating f at two nearby points. Typical of finite-difference formulae are the *forward-difference* formula [15, p339]

$$f'(x) \approx \dot{f}_f = \frac{f(x+h) - f(x)}{h} \tag{8}$$

and the *central-difference formula* [15, p340]

$$f'(x) \approx \dot{f}_c = \frac{f(x+h) - f(x-h)}{2h}. \tag{9}$$

Here h is a "step" selected in an appropriate way. It is typically chosen to balance truncation error (the error given by ignoring higher-order terms in the Taylor-series expansion (7) from which the formula is derived) and subtractive-cancellation error (the error incurred when forming differences between function values at closely-neighbouring points). Using the Taylor expansion (7), we have

$$\dot{f}_f - f' = \frac{h}{2}f''(x) + \frac{h^2}{3!}f'''(x) + \frac{h^3}{4!}f''''(x) + \dots$$

and

$$\dot{f}_c - f' = \frac{h^2}{3!}f^{(3)}(x) + \frac{h^4}{5!}f^{(5)}(x) + \dots$$

From this it can be shown that the truncation error in (8) is $h|f''(\xi_1)|/2$, where $\xi_1 \in [x,\ x+h]$, and that in (9) is $h^2|f'''(\xi_2)|/6$, where $\xi_2 \in [x-h,\ x+h]$. For example, if f is a quadratic function

$$f(x) = ax^2 + bx + c,$$

then, in exact arithmetic, (8) estimates the derivative by $2a(x + h/2) + b$ while (9) gives the correct value $2ax + b$. Thus, for appropriately scaled problems [15, p341], an optimal choice of h would be proportional to $\eta^{1/2}$ for (8) and $\eta^{1/3}$ for (9). Here, η is the relative machine precision, i.e., the smallest machine representable positive value η such that the value of $1 + \eta$, computed in floating-point arithmetic, exceeds 1. For IEEE arithmetic, $\eta = 2.2204 \times 10^{-16}$. For appropriately scaled problems, the use of (8) can be expected at best to lose "half a wordlength of accuracy" and the use of (9) to lose one-third of the available figures. From problems involving a number

of parameters each with different scaling, determining a good value of h in order to achieve the above 'best accuracies' is not trivial.

We note that if the function $f(x)$ has already been evaluated at x, then the forward-difference formula requires one additional function evaluation, and the central-difference two, per derivative evaluation.

Table 1 illustrates the performance of the two finite difference schemes (8) and (9) for evaluating the derivative of $f(x) = x^3$ at $x = 0.001$, 0.1, 10.0 and 1000.0. The table presents, for the values of $h = 10^{-1}, 10^{-2}, \ldots, 10^{-15}$, measures P and S of the relative error in the estimated derivatives $\dot{f}_f$ and $\dot{f}_c$, respectively, given by

$$P = -\log_{10}\left|\frac{\dot{f}_f - f'}{f'}\right|; \qquad S = -\log_{10}\left|\frac{\dot{f}_c - f'}{f'}\right|.$$

Rounded to an integer, P gives the number of correct significant figures in the estimate. (A negative value for P indicates that the relative error is greater than 1.) The table shows firstly the superior performance of the central difference scheme and secondly the dependence of the accuracy of the estimated derivatives on h and x. At $x = 1.0$, the forward-difference scheme has its best performance for $h = 10^{-8} \doteq \eta^{1/2}$ while the central-difference scheme performs best at $h = 10^{-6}$. This is in line with the theoretical considerations discussed above. The table also shows the competing effects of truncation and cancellation errors as a function of step-size.

5.2 A quick check of derivatives using finite differences

The following method can be used to check whether there are any errors in the computation of the elements of a Jacobian matrix, for example.

Let $\mathbf{f}$ be a vector of m functions of variables $\mathbf{x} = (x_1, \ldots, x_n)^{\mathrm{T}}$ evaluated at $\mathbf{x}$ and J the corresponding Jacobian matrix also evaluated at $\mathbf{x}$. Let $\mathbf{r}$ be an m vector generated at random (using a random number generator), where $\|r\| \approx 1$, and set $h \doteq \sqrt{\eta}$ (e.g., $h = 10^{-8}$ for IEEE arithmetic). Then, if J is correct, $J\mathbf{r}$ is the vector of derivatives of $\mathbf{f}(t) = \mathbf{f}(\mathbf{x} + t\mathbf{r})$ evaluated at $t = 0$ and can be approximated by $(\mathbf{f}(\mathbf{x} + h\mathbf{r}) - \mathbf{f}(\mathbf{x}))/h$. If these two vectors are in relatively close agreement, then we have more confidence that the derivatives have been calculated correctly. If there has been an error then it is usually (but not always) very apparent. Further checks can be performed with different random vectors $\mathbf{r}$ and different $\mathbf{x}$. The following is a simple Matlab script for performing this quick check.

x	0.001		0.1		1.0		10.0		1000.0	
H	P	S	P	S	P	S	P	S	P	S
1	-4	-4	0	0	1	2	2	4	4	8
2	-2	-2	1	2	2	4	3	6	5	11*
3	0	0	2	4	3	6	4	8	6	10
4	1	2	3	6	4	8	5	11*	7	10
5	2	4	4	8	5	10	6	10	8*	9
6	3	6	5	10	6	11*	7	9	8*	8
7	4	8	6	11*	7	11*	8*	8	6	6
8	5	11*	7	9	9*	9	7	7	6	6
9	6	10	8*	9	7	8	7	7	5	5
10	7	9	7	7	7	7	6	6	3	3
11	8*	8	7	7	7	7	4	4	3	3
12	8*	8	6	5	4	4	4	4	2	2
13	6	6	5	4	3	4	2	2	1	1
14	5	5	3	3	3	3	1	1	0	0
15	4	4	3	3	1	1	0	0	0	0

Table 1: Performance measures P and S for finite difference schemes (8) and (9), respectively, applied to the function $f(x) = x^3$ for different values of $h = 10^{-H}$ and x. The most accurate results are marked by '*'.

```
[f, J ] = fguser(x,p1,...)      % Evaluate f at x

n = length(x)
r = randn(n,1)
h = 1.0e-8

[fr   ] = fguser(x+h*r,p1,...)  % Evaluate f at x+h*r

J*r                              % Directional derivative given by J
(fr-f)/h                         % Directional derivative estimated
                                 % by finite differences

Jr - (fr-f)/h                    % Difference
```

5.3 Second derivatives

From the calculation of $f(x+h)$, $f(x)$ and $f(x-h)$, we can also determine an estimate of $f''(x)$ since

$$f(x+h) + f(x-h) = 2f(x) + h^2 f''(x) + 2\frac{h^4}{4!} f''''(x) + \ldots,$$

	sin		cos		exp		log	
H	S1	S2	S1	S2	S1	S2	S1	S2
2	3	3	3	3	3	3	3	3
3	4	4	4	4	4	4	4	4
4	5	5	5	5	5	5	5	5
5	6	6	6	6	6	6	6	6
6	7	7	7	7	7	7	7	7
7	8	8*	8	8*	8	8*	8	8*
8	9	8*	9	7	9	8*	9	7
9	10	8*	10	8*	10	7	10	7
10	11	6	11	6	11*	6	11	5
11	11	5	13*	5	11*	5	12*	7
12	12*	4	11	3	11*	4	11	3
13	10	3	10	2	9	3	10	2
14	10	2	9	1	9	2	9	2
15	9	2	9	0	9	1	9	0
16	8	1	8	0	9	1	8	0

Table 2: Performance measures S_1 and S_2 for finite difference schemes (9) and (10), respectively, applied to the functions $\sin x$, $\cos x$, e^x and $\log x$ at $x = 2$ for different values of $h = 10^{-H/2}$. The most accurate results are marked by '*'.

so that

$$f''(x) \doteq \ddot{f} = \frac{(f(x+h) - f(x)) - (f(x) - f(x-h))}{h^2}. \tag{10}$$

The Taylor expansion above suggests that, for well-scaled problems, a value of $h = \eta^{1/4}$ should be suitable.

Table 2 illustrates the performance of the finite difference schemes (9) and (10) to evaluate first and second derivatives of $\sin x$, $\cos x$, e^x and $\log x$ at $x = 2$ with steps sizes $h = 10^{-2/2}, 10^{-3/2}, \ldots, 10^{-16/2}$. The table gives the measure S_1 of the relative error in the estimated derivative $\dot{f}$ and corresponding measure

$$S_2 = -\log_{10} \left| \frac{\ddot{f} - f''}{f''} \right|,$$

for the second derivative. As predicted by the theory, the finite difference scheme (9) gives the most accurate answers for $h \approx 10^{-5}$ while (10) performs best with $h \approx 10^{-4}$.

5.4 Advantages

The most important advantage of the finite difference approach is that it does not require access to the source code for the function evaluation component and therefore, from this point of view, it is an ideal method for library software.

Second derivative estimates can also be formed but usually with less accuracy compared to a central finite difference scheme for first derivatives.

5.5 Disadvantages

The method provides only approximate estimates. In ideal circumstances using central differences we still expect to loose one third of the available figures. For poorly scaled or ill-conditioned problems, the accuracy can be much worse. For metrology applications with data accurate to 1 part in 10^6, there is a significant risk that finite differences implemented in IEEE arithmetic could introduce errors comparable or larger than the measurement uncertainty.

There are technical difficulties in balancing truncation errors with cancellation errors in the choice of step size h.

For problems involving a large number of parameters, the method is inefficient. If there are n parameters, then between $n + 1$ and $2n + 1$ function evaluations are required to calculate all n partial derivatives. No advantage of multiple occurrence of common expressions can be made. However, if it is known from the structure of the problem that a significant number of derivatives are zero, the finite difference scheme can be modified to calculate only the nonzero derivatives.

6 The complex step method

6.1 Description

The complex-step method is similar to finite differences but uses complex arithmetic to provide accurate estimates. Current interest (e.g. [22, 23]) in the approach was generated by Squire and Trapp [31] who revisited earlier work by Lyness and Moler [20]. We recall that a complex number z is of the form $z = x + iy$ where x and y are real and $i = \sqrt{-1}$. All the arithmetic operations for real numbers also apply for complex numbers. Most real-valued functions $f(x)$ occuring in science also have complex-valued counterparts, and can be witten in the form

$$f(z) = \Re f(z) + i\Im f(z),$$

where the real-valued functions $\Re f$ and $\Im f$ are known as the *real* and *imaginary parts*. The concept of derivative is defined by the complex version of (1) and from this we can also derive the Taylor expansion for a complex function in the form

$$f(z+w) = f(z) + wf'(z) + \frac{w^2}{2}f''(z) + \frac{w^3}{3!}f'''(z) + \frac{w^4}{4!}f''''(z) + \ldots, \quad (11)$$

where z and w are complex numbers. (The Taylor expansion holds only for what are termed analytical functions which have continous derivatives of all orders, but these include almost all the function of interest to science.) Taking $z = x$ and $w = ih$ where x is real and h is real and small, we have

$$f(x+ih) = f(x) + ihf'(x) - \frac{h^2}{2}f''(x) - i\frac{h^3}{3!}f'''(x) + \frac{h^4}{4!}f''''(x) + \ldots. \quad (12)$$

Taking real and imaginary parts, we have

$$\Re f(x+ih) = f(x) - \frac{h^2}{2}f''(x) + \frac{h^4}{4!}f''''(x) + \ldots. \quad (13)$$

and

$$\Im f(x+ih) = hf'(x) - \frac{h^3}{3!}f'''(x) + \frac{h^5}{5!}f'''''(x) + \ldots.$$

In this last equation, we see that for h small,

$$f'(x) \doteq \dot{f}_z = \frac{\Im f(x+ih)}{h} \quad (14)$$

with a truncation error of order h^2. Unlike the use of a finite-difference formula, h can be chosen to be *very* small with no concern about the loss of significant figures through subtractive cancellation since no subtraction

is involved. The only practical restriction is that h must not be chosen so small that it underflows, i.e., is replaced by zero in floating-point arithmetic. In our software, we use $h = 10^{-100}$, which should be suitable for all but pathologically-scaled problems.

The success of this approach, i.e., the use of the approximation (14), depends on the availability of complex arithmetic and on the integrity of the inbuilt complex-valued functions. Some of these aspects are not entirely straightforward [23]. For example, if the sine function is evaluated for complex arguments as

$$\sin z = \frac{e^{iz} - e^{-iz}}{2i},$$

then if $z = x + ih$ with h very small,

$$\Im \sin(x + ih) = \cos(x)\frac{e^h - e^{-h}}{2}$$

and there is damaging subtractive cancellation in the computation of $e^h - e^{-h}$ (precisely the behaviour we are trying to avoid).

The complex step method is particularly suitable for implementation in Matlab since the default data type is complex, and all the main intrinsic functions can be evaluated for complex arguments. For example, the following Matlab script will calculate accurate derivatives of the function $f(x) = x^3$ at the points $x = 2^{-10}, 2^{-9}, \ldots, 2^{10}$.

```
h = 1.0e-100
n = [-10:10]'

x = (2*ones(21,1)).^n % x = 2^n, n= -10,-9,...,10.

df = imag((x+i*h).^3)/h    % Derivative using complex
                           % step method.

% Difference between estimated and analytical derivatives.

df - 3*x.*x
```

Matlab evaluates the differences to be identically zero.

Care must be taken to ensure that the intrinsic functions used in the function evaluation component behave in the appropriate way in complex arithmetic. For example, if **x** is an n-vector (real or complex), the Matlab `norm` function to calculate the norm of the vector always returns a real-valued number, the square root of the sum of the squares of the real and imaginary components. Hence, the imaginary part will always be zero. If we instead calculate `normx = sqrt(x'*x)` we will still obtain an incorrect derivative since, in

Matlab, `z'` determines the complex-conjugate transpose of a vector (where the signs of the imaginary parts are reversed). However, if we use `normx = sqrt(x.'*x)`, we obtain the correct result (since the operator `.'` calculates the transpose of `x` without taking complex conjugates).

We recommend that software components to calculate derivatives using the complex step method be validated using finite differences, for example (section 5.2).

In languages such as Fortran 90 [25] which support complex arithmetic, the complex step method is also easy to implement. The program below to differentiate $f(x) = x^3$ is equivalent to the Matlab script above. It also evaluates the differences between the analytical and estimated derivatives as exactly zero (0.0D0).

```
PROGRAM differentiate_cube
implicit none
double precision h,x(21),df(21)
integer          n(21)
double complex   z(21),f(21)

n = (/-10:10/)

x = 2.0d0

x = x**n

h = 1.0d-100

z = dcmplx(x,h)

f = z**3

df = imag(f)/h

print *, df-3*x*x

END PROGRAM differentiate_cube
```

More useful is the component FGRAD, below, which calculates the gradient vector of a function of n variables specified by a user-supplied component FUSER with complex-valued input and output:

```
      SUBROUTINE fgrad(n,x,y,g,fuser)
      implicit none
      integer,           intent(in)  ::  n
      double precision,  intent(in)  ::  x(n)
      double precision,  intent(out) ::  y,g(n)
      INTERFACE
        SUBROUTINE fuser(n,x,y)
        integer,    intent(in)   :: n
        double complex, intent(in)   :: x(n)
        double complex, intent(out)  :: y
        END SUBROUTINE fuser
      END INTERFACE

!     Local variables
      integer i
      double precision hv(n)
      double complex xi(n),yi
      double precision, parameters :: h = 1.0d-100

      do i=1,n

         hv = 0.0d0
         hv(i) = h             ! Step by h in the ith element.

         xi = dcmplx(x,hv)     ! The ith element of xi is x(i)+i*h

         call fuser(n,xi,yi)   ! Evaluate f at xi.

         g(i) = imag(yi)/h     ! Complex step estimate of ith derivative.


      enddo

      y = real(yi)             ! Real component of yi stores y.

      END SUBROUTINE fgrad
```

We can use FGRAD to calculate the gradient of the function

$$y = f(x_1, x_2) = x_1 + x_2 \cos x_1 x_2,$$

for example. Usually, we would evaluate y in a component similar to:

```
      SUBROUTINE feval(n,x,y)

      implicit none
      integer,          intent(in)  :: n
      double precision, intent(in)  :: x(n)
      double precision, intent(out) :: y

      double precision x1,x2,z,w  ! local variables

!     Calculate y
!     -----------
      x1 = x(1)
      x2 = x(2)

      z  = x1*x2
      w  = cos(z)

      y  = x1 + x2*w
!     --------------

      END SUBROUTINE feval
```

In order to use FGRAD, we encode it as

```
      SUBROUTINE feval_cs(n,x,y)

      implicit none
      integer,        intent(in)  :: n
      double complex, intent(in)  :: x(n)
      double complex, intent(out) :: y

      double complex x1,x2,z,w  ! local variables

!     Calculate y
!     -----------
      x1 = x(1)
      x2 = x(2)

      z  = x1*x2
      w  = cos(z)

      y  = x1 + x2*w
!     --------------

      END SUBROUTINE feval_cs
```

FEVAL_CS is identical to FEVAL except for the variable declarations. The following program compares the use of FGRAD against analytical derivatives:

```
      PROGRAM test_fgrad

      implicit none
      integer          n
      double precision x(2),y,g(2)
      external feval_cs

      n = 2
      x(1) = 1.0d0
      x(2) = 2.0d0

      call fgrad(n,x,y,g,feval_cs)

!     Compare calculated and analytical derivatives.

      print*, g(1) - 1.0d0 +(x(2)**2)*sin(x(1)*x(2))
      print*, g(2) - cos(x(1)*x(2)) +(x(1)*x(2))*sin(x(1)*x(2))

      END PROGRAM test_fgrad
```

The computed derivatives differ from the analytical derivatives by no more than η, the machine precision.

6.2 Second derivatives

The formula (13) suggests that second derivatives can be approximated by

$$\ddot{f} = 2\frac{f(x) - \Re f(x + ih)}{h^2}.$$

This formula *does* involve substraction so we are not able to set h to a very small number. In fact, this approximation is very similar to that provided by (10) and its performance is likewise similar. The same value of $h = \eta^{1/4}$ is recommended for well-scaled problems.

The values of second and higher derivatives can be defined mathematically by the *Cauchy Integral Formula* [29], e.g,

$$f''(z) = \frac{2}{2\pi i}\int_\gamma \frac{f(w)}{(w-z)^3}dw,$$

where γ denotes a contour around z in the complex plane. This integral can be approximated using quadrature techniques without subtractive cancellation errors and so can give an accurate estimate provided enough function evaluations are made around the contour. This method is unlikely to be competitive with the techniques described in section 7.

6.3 Advantages

The complex step method can provide derivative information accurate to full machine precision.

The method is easy to implement. In a language that supports complex arithmetic, the user can write a function evaluation component as normal, only declaring the relevant variables as complex, but care must be exercised with some intrinsic functions.

There is no requirement to have access to source code, so long as the correct variable types have been used.

6.4 Disadvantages

As with standard finite differences, for problems involving a large number of variables, the method is inefficient. If there are n variables, then n function evaluations are required to calculate all n partial derivatives.

Existing software written in real arithmetic has to be re-engineered to work with complex arithmetic.

Subtractive cancellation can occur in special circumstances.

6.5 Example application: nonlinear regression

We have implemented, in Matlab, a number of components to calculate derivatives using the complex step method. One has the specification:

```
  function [ J ] = fdiffe( a, f_model, p1, p2, p3, p4);
% ------------------------------------------------------------------
% Compute an approximation to the Jacobian matrix for
% m functions of n variables calculated by f_model.
% ------------------------------------------------------------------
% Input
%    a         n x 1    Array of variables.
%
%    f_model   Name of a module for evaluating the function.
%
% Optional input parameters
%    p1,...    Parameters passed to f_model.
%
% Output
%    J         m x n    Jacobian matrix.
% ------------------------------------------------------------------
```

We have used this component in a solver to perform nonlinear regression. Suppose we wish to find the best fit curve $y = \phi(x, \mathbf{a})$ to a set of data $X = \{(x_i, y_i)\}_{i=1}^{m}$. Estimates of the parameters $\mathbf{a}$ can be determined by solving

$$\min_{\mathbf{a}} \sum_{i=1}^{m} f_i^2(\mathbf{a}),$$

where $f_i(\mathbf{a}) = w_i(y_i - \phi(x_i, \mathbf{a}))$ and w_i are suitably chosen weights [10]. The solver has the specification:

```
  function [a,phi,f,J] = nllsf(ai,tol,x,y,w,f_model,p1,p2,p3,p4);
% ------------------------------------------------------------------
% Determine the weighted least squares best fit curve y =  phi(x,a)
% to data (x,y) where f_model calculates phi and has specification
%
%      [ f ] = f_model(a,x,p1,p2...)
% ------------------------------------------------------------------
% Input
%    ai        n x 1
%              Initial estimate of the parameters.
%
%   tol        2 x 1
%              Convergence tolerances.
%
%     x        m x 1
%     y        Data point coordinates.
%
%     w        m x 1
%              Weights.
%
%    f_model   Name of a function for evaluating the
%              function phi.
%
% Optional input parameters
%    p1,...    Optional parameters passed to f_model.
%
% Output
%    a         n x 1
%              Solution estimate of the parameters.
%
%    phi       m x 1
%              Model function phi evaluated at a and x.
%
%    f         m x 1
%              Residuals evaluated at a.
%
%    J         m x n
%              Jacobian matrix of partial derivatives of
%              f with respect to a.
% ------------------------------------------------------------------
```

and implements a version of the Gauss-Newton algorithm (section 2.1). The solver has been used on a wide range of regression problems. Its behaviour has been observed to be identical to a similar solver for which the user has to supply the gradients $\partial\phi/\partial a_j$.

The component FDIFFE.M can be used to calculate the Jacobian matrix associated with any nonlinear function specified by the user in the component F_MODEL.M. It has also been used to determine sensitivity coefficients, for example (section 2.2).

The Fortran 90 component FGRAD.F90 or similar can be used in exactly the same way. In particular, with optimisation software that uses harness components [9], we can provide solvers that require only function values but solve the optimisation problem using an optimisation component that also requires gradients. For example, the MINPACK nonlinear least squares component LMDER [14] can be adapted to operate in this way.

7 Program differentiation techniques

Automatic differentiation (AD) is a set of techniques aimed at 'differentiating the program' that computes a function value [11]. AD is an accurate method in the sense that AD applies the rules of calculus in a repetitive way to an algorithmic specification of a function and, in exact arithmetic, will produce the exact answer. Like symbolic algebra, AD relies on the fact that any program, no matter how complex it is, can be broken down into a finite combination of elementary operators such as arithmetic operations (e.g. $+$, $-$) or elementary functions (e.g. $\sin x$, $\cos x$, e^x) [4, 18].

For example, consider the function,

$$f(x_1, x_2) = x_1 + x_2 \cos(x_1 x_2). \tag{15}$$

In order to construct an algorithm to evaluate the value of the function $f(x_1, x_2)$ in (15), we start from x_1 and x_2. The evaluation can be performed in the following order:

$$\left.\begin{array}{rcl} t_1 &=& x_1, \\ t_2 &=& x_2, \\ t_3 &=& t_1 t_2, \\ t_4 &=& \cos t_3, \\ t_5 &=& t_2 t_4, \\ t_6 &=& t_1 + t_5. \end{array}\right\} \tag{16}$$

There are two different programming approaches to differentiating program code: *operator overloading* and *source-to-source transformation*. In the operator overloading approach, the basic arithmetic operations and intrinsic functions are assigned routines that calculate the derivatives of the operators outputs in addition to the calculation of the function value. The source code of the function is progressively differentiated by calling these routines at the same time as each operation is performed in the evaluation of the program. There are two approaches to program differentiation: *forward automatic differentiation* (FAD) and *reverse automatic differentiation* (RAD). These are described below (sections 8 and 9). Operator overloading is only allowed by a limited number of programing languages such as Fortran 90, Ada, C++ and Matlab. Examples of software packages that use this approach are ADOL-F [30] and ADOL-C [19].

The source-to-source approach defines a new source code for calculating the derivatives explicitly from the program function evaluation source code. However, the implementation of this method requires considerable programming effort. Examples of software packages that use this approach are ADIFOR [6] and ODYSSEE [12]. See section 10.

8 Forward Automatic Differentiation

8.1 Description

Forward Automatic Differentiation (FAD) was the first to be introduced when it was implemented in the early 1960's by, for example, Wengert [32]. The process is called "forward" since it computes the derivative information at the same time as it computes the function value. The process builds up the gradient values, step by step, as it goes through the basic operations and elementary functions of the program code. For example, suppose we want to calculate the gradient of the function $y = f(x_1, \cdots, x_n)$, where the number of steps to find y is m. We introduce the variables $d_1, \ldots, d_m$ to record the intermediate quantities calculated along the way. At the jth operation, d_j is calculated as a function Ψ_j of at most two terms already computed

$$d_j = \Psi_j(d_k, d_l), \quad 1 \leq k, l \leq j-1,$$

where Ψ_j is a unary, binary operation or elementary function. The evaluation algorithm to compute y can therefore be described as

I Initialise $d_i = x_i$, $i = 1, \ldots n$.

II For $j = n+1, \ldots, m$,
$$d_j = \Psi_j(d_k, d_l).$$

III Set $y = d_m$.

Now let $\mathbf{u}$ and $\mathbf{d'}_i$ be n-vectors with $\mathbf{u} = (1, \ldots, 1)^{\mathrm{T}}$. The FAD algorithm for calculating $\mathbf{y'} = (\partial y/\partial x_1, \ldots, \partial y \partial x_n)^{\mathrm{T}}$ is:

I Initialise $\mathbf{d'}_i = \mathbf{u}$, $i = 1, \ldots n$.

II For $j = n+1, \ldots, m$

$$\mathbf{d'}_j = \frac{\partial \Psi_j}{\partial d_k}\mathbf{d'}_k + \frac{\partial \Psi_j}{\partial d_l}\mathbf{d'}_l. \tag{17}$$

III Set $\mathbf{y'} = \mathbf{d'}_m$.

The $\partial \Psi_j / \partial d_{l,k}$ are computed using the basic rules of differential calculus. The computation of both the function value y and gradient $\mathbf{y'}$ are done simultaneously. If we define a variable $D = \{d, \mathbf{d'}\}$, called *doublet*, then the whole process can be defined by the following algorithm:

Forward automatic differentiation

I Initialise $D_i = \{x_i, \mathbf{u}\}$, $i = 1, \ldots n$.

II For $j = n+1, \ldots, m$
$$D_j = \Psi_j(D_k, D_l).$$

III Set $\{y, \mathbf{y}'\} = D_m$.

At the end of the process, the doublet D_m holds both the function value y and its gradient $\mathbf{y}'$. Here the functions Ψ_j are interpreted as applying to doublet variables. For example if Ψ is the multiplication operator and $D = \{d, \mathbf{d}'\}$ and $E = \{e, \mathbf{e}'\}$ are two doublets, then

$$\Psi(D, E) = \{de, d\mathbf{e}' + e\mathbf{d}'\}.$$

The memory requirement for computing the gradient of a function of n variables using FAD is between n and $4n$ times the memory requirement of computing the function itself.

We have implemented the FAD approach in Fortran 90, a language that supports operator overloading [5]. Its main features are:

- The definition of the type DOUBLET.
- The definition of INTERFACEs to overload the standard operations with those for DOUBLETs.
- The implementation of the operations on DOUBLETs as FUNCTIONs.
- The development of generic type transformations from REAL to DOUBLET and vice versa.

All these components are contained in a MODULE called FAD_OPERATORS.

The declaration of the type doublet is straightforward:

```
TYPE doublet
real*8 :: d
real*8, allocatable, dimension(:) :: gradd
END TYPE doublet
```

The interface blocks have the form typified by:

```
INTERFACE OPERATOR (*)
  MODULE PROCEDURE multiply, multiply_constant,constant_multiply,&
      ...
END INTERFACE
```

which relates the '*' operator to the FUNCTIONs MULTIPLY, MULTIPLY_CONSTANT, etc., for multiplication with doublet-doublets, doublet-reals, etc. The FUNCTION MULTIPLY, for example, is:

```
      FUNCTION multiply(u,v)
      TYPE(doublet), INTENT(IN):: u,v
      TYPE(doublet):: multiply

!     Function values

      multiply%d     = (u%d)*(v%d)

!     Gradients

      multiply%gradd = (u%d)*(v%gradd) + (u%gradd)*(v%d)

      END FUNCTION multiply
```

We illustrate the use of FAD on a simple component to calculate $y = f(x_1, x_2) = x_1 + x_2 \cos x_1 x_2$: The following component uses FAD to calculate the y and its gradient $\mathbf{g}$ with respect to x_1 and x_2:

```
      SUBROUTINE fgeval_fad(xi,yo,g)

      USE fad_operators
      double precision, intent(in)  ::  xi(2)      ! xi is the input x
      double precision, intent(out) ::  yo,g(2)    ! yo is the output y

      type(doublet) :: x(2),y,x1,x2,z,w  ! local variables

      do i=1,size(x)
         call init_doublet(xi(i),i,x(i))
         ! Initialize doublets corresponding
         ! to the independent variables
         ! x(1) = [xi(1),(1,0)] ; x(2) = [xi(2),(0,1)]
      end do

!     Calculate y
!     -----------
      x1 = x(1)
      x2 = x(2)

      z  = x1*x2
      w  = cos(z)

      y  = x1 + x2*w
!     --------------

      call doublet2real8(y,yo,g) ! Extract yo and g from doublet y.

      END SUBROUTINE fgeval_fad
```

Apart from the variable declarations and the call to the type conversion components, the main computionals steps in FGEVAL_FAD is identical to FEVAL, section 6.1. As far as calling the component FGEVAL_FAD, its signature is simply:

```
subroutine fgeval_fad(x,y,g)

double precision, intent(in)  ::  x(2)
double precision, intent(out) ::  y, g(2)
```

There is no need to know anything about how **g** is calculated, only that in order to access the FAD routines, the software has to link to the MODULE FAD_OPERATORS.

8.2 Second derivatives

In order to obtain second derivative information using FAD, an extra term representing the Hessian of second partial derivative matrix needs to be added. If f is a function of n variables, we extend the doublet variable to a variable called a *triplet* $\{d, \mathbf{d}', d''\}$ where d'' is an $n \times n$ matrix. The same scheme for doublets is applied to triplets with the matrices d''_i initialised to $\mathbf{0}$ at the beginning of the algorithm and subsequent triplets calculated according to the rules of calculus and the computational scheme for y. For example, if Ψ is the multiplication operator and $D = \{d, \mathbf{d}', d''\}$ and $E = \{e, \mathbf{e}', ee''\}$ are two triplets, then

$$\Psi(D, E) = \{de, d\mathbf{e}' + e\mathbf{d}', de'' + ed'' + \mathbf{d}'\mathbf{e}'^{\mathrm{T}} + \mathbf{e}'\mathbf{d}'^{\mathrm{T}}\},$$

where the ijth element of the $n \times n$ matrix $\mathbf{d}'\mathbf{e}'^{\mathrm{T}}$ is $\mathbf{d}'(i) \times \mathbf{e}'(j)$, etc.

8.3 Advantages

The FAD approach provides mathematically accurate derivative information.

FAD is to some extent an easy algorithm to implement in that its approach only involves the application of the basic rules of calculus.

Second and higher derivatives can be calculated accurately (in the sense that the computational steps are mathematically correct).

FAD is efficient for problems with a small number of independent variables. The cost of differentiating a function f with respect to n independent variables is at least n and at most $4n$ the cost of evaluating the function f.

8.4 Disadvantages

The FAD algorithm requires the source code for computing the function to be available to the user in order to modify it. However, the modification only involves changing the declarative part of the source code while the rest of the code remains substantially the same.

The algorithm requires the declaration of a new type of variable and the overloading of operations that are associated with this new type. Not all programming languages allow such procedures to be implemented.

9 Reverse Automatic Differentiation

9.1 Description

Reverse Automatic Differentiation (RAD) is mathematicaly related to *adjoint sensitivity analysis* for differential equations, whose use also dates back to the 1960's. The method was then widely used in nuclear engineering [7], and weather prediction [26], for example. The RAD algorithm works in terms of the computational graph associated with the evaluation of a function. Like FAD, to implement the RAD algorithm we require the declaration of a new scalar variable known as the *adjoint* variable $\bar{t}$ for each node of the computational graph.

The algorithm has two phases. The first, the *forward sweep* is essentially the same as the function evaluation algorithm for FAD:

I Initialise $t_i = x_i$, $i = 1, \ldots n$.

II For $j = n+1, \ldots, m$,
$$t_j = \Psi_j(d_k, d_l).$$

III Set $y = t_m$.

The second, the *reverse sweep*, initialises the adjoint variables $\bar{t}_i$ and then applies adjoint algebra, a set of predefined routines corresponding to each operation or elementary function, to find the gradient of the computed function. This requires stepping in the reverse direction to that of the function generation and using the information saved from the forward sweep to calculate the gradient. The reverse sweep is summarised as:

I Initialise $\bar{t}_i = 0$, $i = 1, \ldots m-1$, $\bar{t}_m = 1$.

II For $j = m, m-1, m-2, \ldots, n+1$, if $t_j = \Psi_j(t_k, t_l)$, then
$$\bar{t}_k = \bar{t}_k + \frac{\partial \Psi}{\partial t_k}\bar{t}_j, \quad \bar{t}_l = \bar{t}_l + \frac{\partial \Psi}{\partial t_l}\bar{t}_j. \tag{18}$$

III For $i = 1, \ldots, n$, $\partial y/\partial x_i = \bar{t}_i$.

For example, the computational graph associated with the evaluation of $y = x_1 + x_2 \cos x_1 x_2$ has six nodes as in (16). The calculation of the gradient involves five updates to the 6 adjoints $\bar{t}_1, \ldots, \bar{t}_6$ shown as columns of the array below. First column shows steps in the forward sweep progressing from top to bottom. The remaining six columns shows the values of the

adjoint variables at each stage of the reverse sweep, progressing from bottom to top. These updates are generated according to rule (18).

	$\bar{t}_1$	$\bar{t}_2$	$\bar{t}_3$	$\bar{t}_4$	$\bar{t}_5$	$\bar{t}_6$
$t_1 = x_1$	$1 - t_2^2 \sin t_3$	$t_4 - t_1 t_2 \sin t_3$	$-t_2 \sin t_3$	t_2	1	1
$t_2 = x_2$	1	$t_4 - t_1 t_2 \sin t_3$	$-t_2 \sin t_3$	t_2	1	1
$t_3 = t_1 t_2$	1	t_4	$-t_2 \sin t_3$	t_2	1	1
$t_4 = \cos t_3$	1	t_4	0	t_2	1	1
$t_5 = t_2 t_4$	1	0	0	0	1	1
$t_6 = t_1 + t_5$	0	0	0	0	0	1

We note that

$$\begin{aligned} \bar{t}_1 &= 1 - t_2^2 \sin t_3 = 1 - x_2^2 \sin x_1 x_2 = \partial y / \partial x_1, \\ \bar{t}_2 &= t_4 - t_1 t_2 \sin t_3 = \cos x_1 x_2 - x_1 x_2 \sin x_1 x_2 = \partial y / \partial x_1, \end{aligned}$$

as required.

The cost of computing the gradient of a function of n variables is between 2 and 6 times the cost of computing the function value itself. We note that this is independent of the number of variables n.

We have also implemented RAD in Fortran 90 [5]. The implementation is similar to that for FAD but also requires the implementation of a component to execute the reverse sweep from the information computed by the forward sweep. This component uses pointers and linked lists to store and manipulate the information and represents a significant increase in programming complexity compared to the FAD approach.

The following component illustrates the use of RAD to calculate the derivatives of $y = f(x_1, x_2) = x_1 + x_2 \cos(x_1 x_2)$:

```
      SUBROUTINE fgeval_rad(xi,yo,g)

      USE rad_operators
      double precision, intent(in)  ::  xi(2)      ! xi is the input x
      double precision, intent(out) ::  yo,g(2)    ! yo is the output y

      type(adjoint) :: x(2),y,x1,x2,z,w  ! local variables

      do i=1,size(x)
         call init_adjoint(xi(i),i,x(i)) ! initialize adjoints corresponding
                                         ! to the independent variables
      end do

!     Calculate y
!     -----------
      x1 = x(1)
```

```
      x2 = x(2)

      z  = x1*x2
      w  = cos(z)

      y  = x1 + x2*w
!     --------------

      call differentiate()

      call adjoint2real8(y,yo,g)        ! Extract yo and g from the adjoint
                                        ! variable.

      call delete_graph                 ! Memory management, deleting
                                        ! pointers, etc.

      END SUBROUTINE fgeval_rad
```

As with the FAD version, FGEVAL_RAD has the same main computational steps as FEVAL, section 6.1. However, the RAD version has additional CALLs to perform the reverse sweep and deleting pointers, etc., that store information specifying the computational graph.

9.2 Second derivatives

The computation of second derivative information is done through a second forward sweep and backward sweep through the computational graph. It can be viewed as applying RAD to the evaluation of the partial derivatives $\partial y/\partial x_i$ calculated using FAD. The information generated during these sweeps is stored in two new sets of parameters $\{s_i\}$ and $\{\bar{s}_i\}$.

The forward sweep is as follows:

For $i = 1, \ldots, n$:

I Initialise $s_q = 0$, $q = 1, \ldots n$, and then set $s_i = 1$.

II For $j = n+1, \ldots, m$, if $t_j = \Psi(t_k, t_l)$, then (*cf.* (17))

$$s_j = s_k \frac{\partial \Psi}{\partial t_k} + s_j \frac{\partial \Psi}{\partial t_l}.$$

The reverse sweep is:

For $i = 1, \ldots, n$:

I Initialise $\bar{s}_q = 0$, $q = 1, \ldots m$.

II For $j = m, \ldots, n+1$, if $t_j = \Psi_j(t_k, t_l)$, then

$$\begin{aligned} \bar{s}_k &= \bar{s}_k + \bar{s}_j \frac{\partial \Psi_j}{\partial t_j} + \bar{t}_j s_k \frac{\partial^2 \Psi_j}{\partial t_k^2} + \bar{t}_j s_l \frac{\partial^2 \Psi_j}{\partial t_k \partial t_l}, \\ \bar{s}_l &= \bar{s}_l + \bar{s}_j \frac{\partial \Psi_j}{\partial t_l} + \bar{t}_j s_l \frac{\partial^2 \Psi_j}{\partial t_l^2} + \bar{t}_j s_k \frac{\partial^2 \Psi_j}{\partial t_k \partial t_l} \end{aligned}$$

This scheme calculates the matrix of second partial derivatives, row by row.

9.3 Advantages

As FAD, RAD provides mathematically accurate derivatives.

The RAD algorithm is efficient in terms of the number of computational steps. This makes RAD particularly suitable for problems with large number of independent variables.

The RAD method is also efficient with respect to memory requirements. In particular, the memory requirement is independent of the number of variables.

9.4 Disadvantages

The RAD algorithm requires the modification of the source code by the user. Similarly to FAD, the modification only involves changing the declarative part of the source code while the rest of the code remains substantially the same.

To implement RAD, the user has also to declare a new type of variable and the overload operators associated with this new type.

The RAD algorithm implementations involve linked lists, binary trees, pointers and careful memory management. It requires a language that supports operator overloading such as Fortran 90.

9.5 Example application: maximum likelihood estimation

Weighted nonlinear least squares problems such as those considered in section 2.3 arise when the uncertainties σ_i in the input data have been adequately characterised. If we have only approximate estimates s_i for σ_i, it may be appropriate to regard the σ_i as unknowns but for which we have some statistical model. For example, suppose the model is

$$y_i \in N(\phi(x_i, \mathbf{a}), \sigma_i^2), \quad \log s_i^2 \in N(\log \sigma_i^2, \rho_i^2)$$

where ρ_i is known. The maximum likelihood estimate of $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_m)^{\mathrm{T}}$ and $\mathbf{a}$ is determined by minimising a log-likelihood function of the form

$$\begin{aligned} F(\sigma, \mathbf{a}) &= \sum_{i=1}^{m} \log \sigma_i^2 + \sum_{i=1}^{m} \frac{1}{2} \left(\frac{y_i - \phi_i(x_i, \mathbf{a})}{\sigma_i} \right)^2 \\ &+ \sum_{i=1}^{m} \frac{1}{2} \left(\frac{\log s_i^2 - \log \sigma_i^2}{\rho_i} \right)^2 . \end{aligned}$$

Optimisation software such as component E04LBF in the NAG library [27] requires us to calculate the gradient vector of first derivatives and the Hessian matrix of second derivatives of F with respect to $\boldsymbol{\sigma}$ and $\mathbf{a}$. We have calculated these using the RAD algorithm implemented in Fortran 90.

10 Source to source

10.1 Description

A source to source AD method is a code translator that takes a file containing source code for a function evaluation component and outputs a file containing source code for computing the derivatives. In this way it is similar to the symbolic algebra approach.

The software package ADIFOR 2.0 is probably the best example of a source to source method [6] and applies to software written in Fortran 77 [1]. In the code canonicalization phase, the source Fortran 77 code is rewritten into a standard form. Function or subroutine calls are transformed into assignments to new temporary variables. Statement functions are expanded into inline code. Long right-hand sides of assignment statements are broken down into smaller pieces, and rewritten so that all variables appearing on the righthand side of an assignment statement are of the same type. The software does require some prior information about the variables with respect to which derivatives are required. This is done by initialising these variables to *active* variables (i.e., the independent variables) and initilising the rest to *passive variables* (i.e., dependent variables).

The ADIFOR processor performs an interprocedural analysis on the 'canonicalized' code to determine which variables in the code are active and then builds the computational graph corresponding to the code. Next, in a procedure similar to a search routine, it identifies all possible program paths through which an independent variable can affect the dependent variables. From this, the processor augments the code with gradient calculations corresponding to each assignment statement containing an active variable.

As with operator-overloading, source to source (SS) approaches can work either in forward or reverse modes (ADIFOR uses a hybrid approach). We can think of SS and operator-overloading as applying the same principles but with SS working at the source-code level while operator-overloading is more at the compiled code level.

10.2 Advantages

SS can differentiate large blocks of code accurately and with a minimum modification to the actual program.

SS produces source code that can then be compiled separately and/or integrated with other components.

SS can be made moderately efficient.

SS can be applied to languages that do not support overloading.

10.3 Disadvantages

SS requires a major amount of effort to implement.

Not all features of the language may be supported by SS.

SS requires access to the source code.

11 Numerical examples

11.1 Example 1: $\sin 1/x$

In order to illustrate the differences in the accuracy of derivatives obtained using the algorithms described above, consider the following analytic function,

$$f(x) = \sin 1/x, \tag{19}$$

plotted in Figure 1. As x tends toward zero the curve oscillates rapidly between 1 and -1. The analytical gradient is given by

$$f'(x) = -\frac{1}{x^2}\cos 1/x,$$

giving rise to large gradients near $x = 0$ (Figure 2). We calculate a measure of the relative error e in a computed estimate $\dot{f}$ of f' as

$$e = \frac{\dot{f} - f'}{\sqrt{[1 + (f')^2]}}. \tag{20}$$

Figure 3 illustrates the error in computing the derivative of $\sin 1/x$ with respect to x using RAD, the complex-step method, and finite differences. The graph plots $-\log_{10}(|e| + \eta)$ where $\eta \approx 2.2 \times 10^{-16}$ is the machine precision. The vertical scale can be interpreted as number of correct digits in the estimate. The finite differences were calculated using the component FDJAC1 from MINPACK [14] which implements a forward difference approach (8). The graph shows that the FAD, RAD and the complex step methods give answers correct to approximately full precision while the finite difference approach gives poor results with the relative error as large as 10 or more. Figure 4 illustrates the excellent accuracy properties of FAD, RAD and the complex step methods. The figure shows that all the methods are providing a relative accuracy of a few multiples of the machine precision η. The central band corresponds to estimates that are judged to be exact (zero error) to full precision.

11.2 Example 2: $x^{-1}\cos x^{-1}$

We perform a similar analysis for the function,

$$f(x) = x^{-1}\cos x^{-1}, \tag{21}$$

see Figure 5, whose derivative is given by

$$f'(x) = \frac{x^{-1}\sin x^{-1} - \cos x^{-1}}{x^2},$$

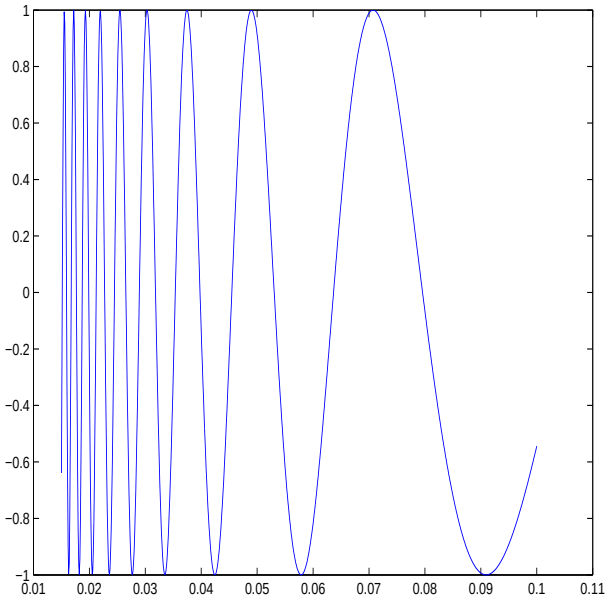

Figure 1: Graph of $f(x) = \sin 1/x, \quad x \in [0.015, 0.1]$

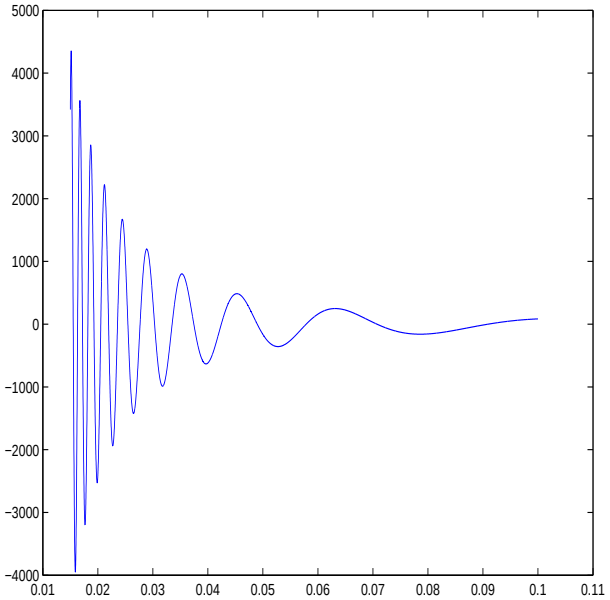

Figure 2: Graph of $f'(x) = -\frac{1}{x^2}\cos 1/x, \quad x \in [0.015, 0.1]$

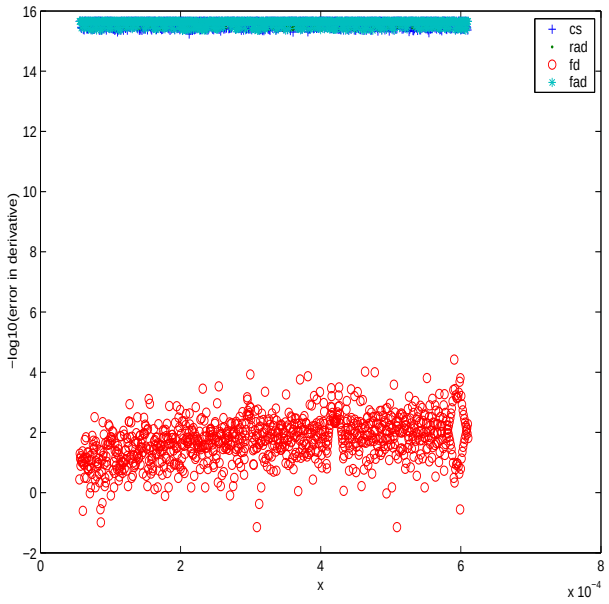

Figure 3: Graph of $-\log_{10}(|e| + \eta)$ where e is the relative error (equation (20)) in computing the derivative of $\sin 1/x, \quad x \in [0.6, 6.0] \times 10^{-4}$, using FAD, RAD, complex step (CS) and finite difference (FD) methods.

(see Figure 6). As in the first example, Figure 7 illustrates the relative error in computing the derivative using FAD, RAD, the complex-step method, and finite differences. A more detailed comparison between FAD, RAD and the complex step method is provided in Figure 8.

11.3 Gradients associated with maximum likelihood estimation

We have used RAD in the implementation of a maximum likelihood estimation problem considered in section 9.5 to compute the gradient $\mathbf{g}$ of F with respect to σ and $\mathbf{a}$. The difference in the analytical gradient and the automatic differentiation gradients is illustrated in Figure 9. The figure shows the error γ between the AD gradient and the analytical gradient for a thousand Monte Carlo simulations, where the estimate γ of the error is computed as

$$\gamma = \|\mathbf{g}_{(AD)} - \mathbf{g}_{(analytical)}\|_2.$$

We can see that the largest error is 4.1×10^{-15} which indicates that the AD is providing values for the gradient with a high numerical accuracy.

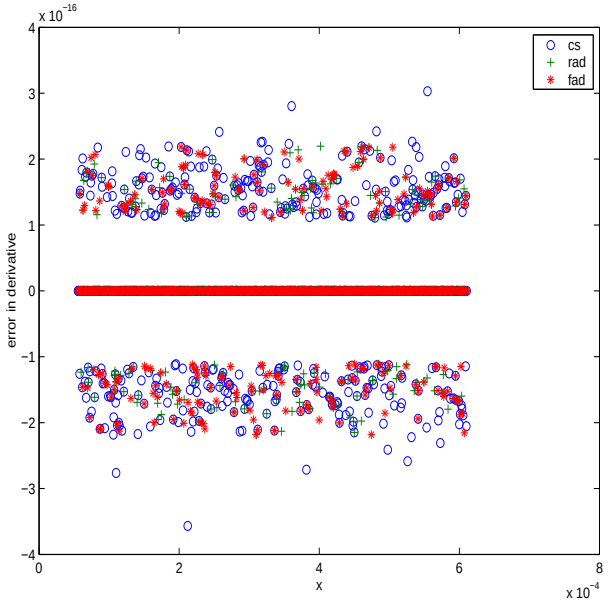

Figure 4: Graph of the relative error e (equation (20)) in computing the derivative of $\sin 1/x, \quad x \in [0.6, 6.0] \times 10^{-4}$, for FAD, RAD and complex step (CS) methods.

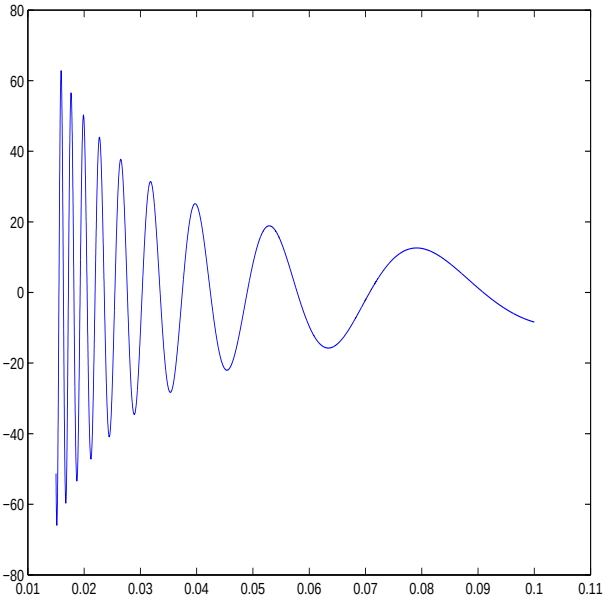

Figure 5: Graph of $f(x) = x^{-1} \cos x^{-1}, \quad x \in [0.015, 0.1]$.

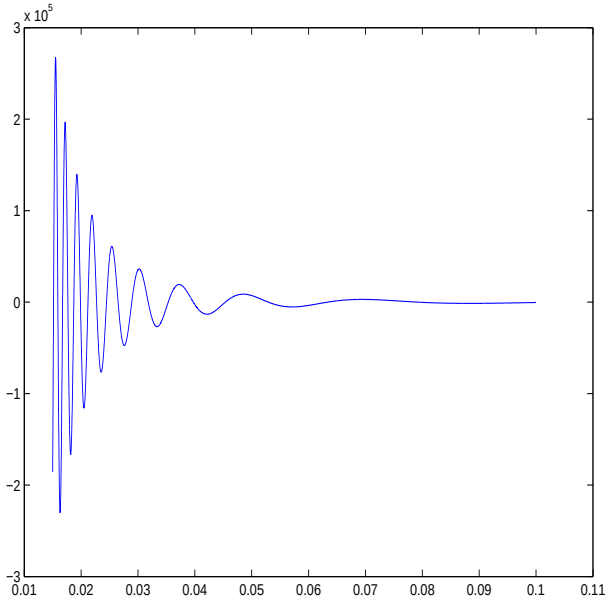

Figure 6: Graph of $f'(x) = \frac{x^{-1}\sin x^{-1} - \cos x^{-1}}{x^2}$, $x \in [0.015, 0.1]$.

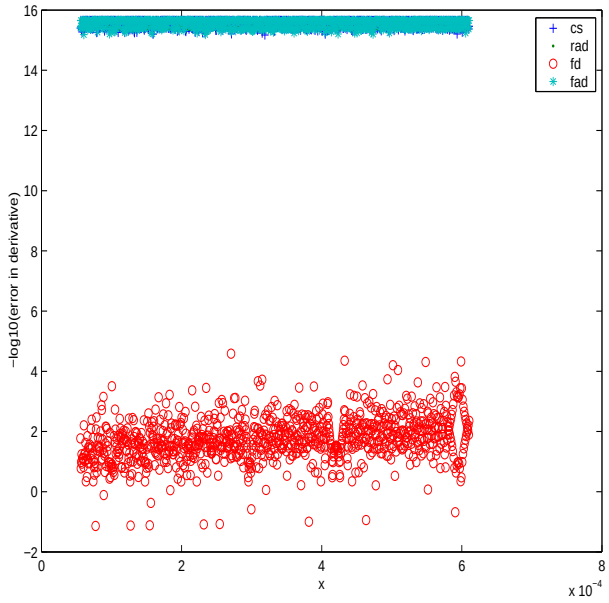

Figure 7: Graph of $-\log_{10}(|e|+\eta)$ where e is the relative error e in computing the derivative of $f(x) = x^{-1}\cos x^{-1}$, $x \in [0.6, 6.0] \times 10^{-4}]$ using FAD, RAD, complex step (CS) and finite difference (FD) methods.

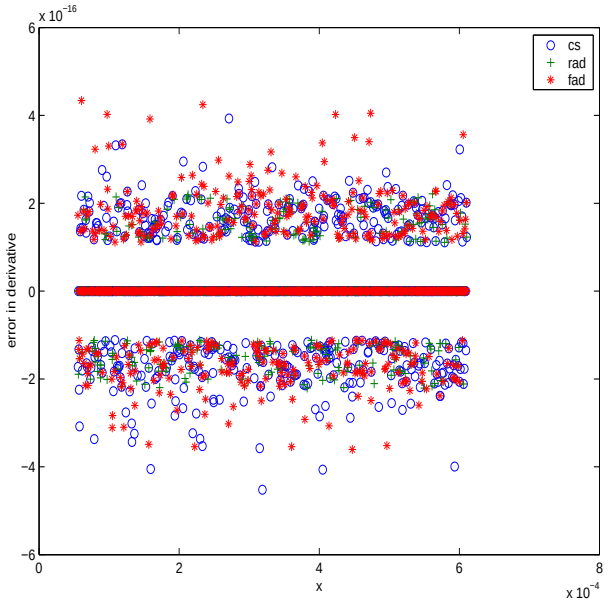

Figure 8: Graph of the relative error e in computing the derivative of $f(x) = x^{-1}\cos x^{-1}, \quad x \in [0.6, 6.0] \times 10^{-4}$ using FAD, RAD and complex step (CS) methods.

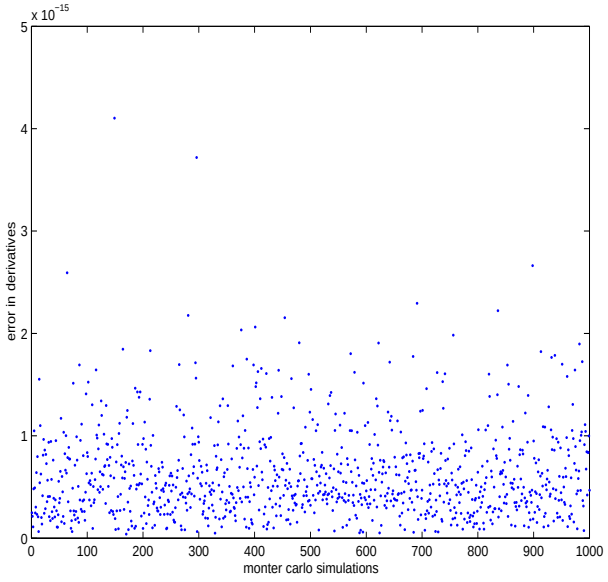

Figure 9: Graph of error γ against number of Monte Carlo simulation trials.

Method	Accuracy f'	Accuracy f''	Implement'n	Use	Efficiency
FD	5	4	1	1	4
CS	1	4	2	2	4
FAD	1	1	3	3	2
RAD	1	1	4	4	1
SS	1	1	5	5	2

Table 3: Ranking of five automatic differentiation methods according to five criteria where 1 is best and 5 is worst.

12 Conclusions and recommendations

In this report we have been concerned with methods for calculating the derivatives of functions. In particular, we have been interested in finite differences (FD), the complex step method (CS), forward automatic differentiation (FAD) and reverse automatic differentiation (RAD) and source to source translation (SS). We have analysed these method in terms of their accuracy, their ease of implementation, their ability to calculate second (and higher) derivatives and their efficiency with respect to speed and memory. We have found:

- FD is the easiest to implement and use but is inaccurate, losing as much as half the available figures. FD is also inefficient for problems involving a large number of variables.
- CS is relatively easy to implement and use, especially in Matlab, accurate but can be inefficient for large problems. The method does not produce accurate second derivatives.
- FAD produces accurate first and second derivatives, is straightforward to implement and use but can be inefficient for large problems.
- RAD produces accurate first and second derivatives, is efficient in both speed and memory. It is quite straightforward to use but its implementation requires significant effort.
- SS produces accurate first and second derivatives. It is moderately efficient, depending on the the degree of sophistication of the implementation. Its implementation requires major effort.

This information can be summarised by the rankings in Table 3, with 1 best and 5 worst.

For modest sized problems not requiring second derivatives then the complex step method has much to recommend it. We have used it extensively

in Matlab in recent years and have found it to meet practically all our requirements for these types of problems. Anyone able to write function evaluation software will be able to implement it. The method is not absolutely guaranteed to produce accurate derivatives since in some circumstances subtractive cancellation can occur. Further work is required to identify when this is likely. More generally, it would be of significant benefit to the scientific community for the complex step method to be further developed and integrated into solvers, etc. If first derivatives have been calculated analytically and implemented in software then CS can be used to calculate accurate second derivatives.

For large problems or problems requiring second derivatives, FAD or RAD need to be considered. Although FAD is easier to implement, it does not provide the efficiency of RAD. It should be borne in mind that for both AD approaches, the development of the FUNCTIONs implementing the overloading, the reverse sweep for RAD, etc., are a *one time only* investment. Once the infrastructure is in place, the use of FAD or RAD involves modest overhead, in terms of programming, above that required to evaluate the function in the first place. Since we are not usually interested in using AD for problems that are entirely simple, the efficiency gains they offer (both computationally and in programming) will usually be significant.

Acknowledgements. This work has been supported by the Department of Trade and Industry's National Measurement System Software Support for Metrology Programme (2001 – 2004).

References

[1] A. Balfour and D. H. Marwick. *Programming in Standard Fortran 77*. Heinemann, London, 1979.

[2] BIPM, IEC, IFCC, ISO, IUPAC, IUPAP, and OIML. *Guide to the Expression of Uncertainty in Measurement*. Geneva, second edition, 1995.

[3] C. Bischof, A. Carle, G. Corliss, A. Griewank, and Hovland P. ADIFOR – generating derivative codes from Fortran programs. *Scientific Programming*, 1:1–29, 1992.

[4] C. Bischof, L. Roh, and A. Mauer. *ADIC – An extensible automatic differentiation tool for ANSI-C*. Argonne, IL. ANL/MCS-P626-1196.

[5] R. Boudjemaa. *Automatic Differentiation in Shape Optimisation*. PhD thesis, Department of Applied Mathematics, Leeds University, 2002.

[6] P. Hovland P. Khademi C. Bischof, A. Carle and A. Mauer. *ADIFOR 2.0 Users' Guide*. Argonne National Laboratory, Argonne, IL, 1998.

[7] D. G. Cacuci. Sensitivity theory for nonlinear systems I: nonlinear functional analysis approach. *Journal of Mathematical Physics*, 22(12):2794–2802, 1981.

[8] M. G. Cox, M. P. Dainton, and P. M. Harris. Software Support for Metrology Best Practice Guide No. 6: Uncertainty and Statistical Modelling. Technical report, National Physical Laboratory, Teddington, 2001.

[9] M. G. Cox, A. B. Forbes, P. M. Fossati, P. M. Harris, and I. M. Smith. Techniques for the efficient solution of large scale calibration problems. Technical Report CMSC 25/03, National Physical Laboratory, Teddington, May 2003.

[10] M. G. Cox, A. B. Forbes, and P. M. Harris. Software Support for Metrology Best Practice Guide 4: Modelling Discrete Data. Technical report, National Physical Laboratory, Teddington, 2000.

[11] C. Faure. Automatic differentiation for adjoint code generation: An introduction to automatic adjoint code generation. Technical Report 3555, INRIA, 1998.

[12] C. Faure and Y. Papegay. Odyssée Version 1.6: User's Reference Manual. Technical report, INRIA, 1997.

[13] A. B. Forbes. Efficient estimators in data fusion. In P. Ciarlini, M. G. Cox, E. Filipe, F. Pavese, and D. Richter, editors, *Advanced Mathematical and Computational Tools in Metrology V*, pages 164–169, Singapore, 2001. World Scientific.

[14] B. S. Garbow, K. E. Hillstrom, and J. J. Moré. User's guide for MINPACK-1. Technical Report ANL-80-74, Argonne National Laboratory, Argonne, IL, 1980.

[15] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, London, 1981.

[16] G. H. Golub and C. F. Van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore, third edition, 1996.

[17] A. Griewank. On automatic differentiation. Technical report, Argonne National Laboratory, Argonne, IL, 1988.

[18] A. Griewank and G. F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation and Applications*, Philadelphia, 1991. Society for Industrial and Applied Mathematics.

[19] 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, 1996.

[20] J. N. Lyness and C. B. Moler. Numerical differentiation of analytic functions. *SIAM J. Numer. Anal.*, 4:202–210, 1967.

[21] Maplesoft, 57 Erb Street West, Waterloo, Ontario, Canada, N2L 6C2. *http://www.maplesoft.com*.

[22] J. R. R. A. Martins, I. M. Kroo, and J. J. Alonso. An automated method for sensitivity analysis using complex variables. In *38th Aerospace Sciences Meeting and Exhibit*, pages 10–13. American Institute of Aeronautics and Astronautics, Jan 2000.

[23] J. R. R. A. Martins, P. Sturdza, and J. J. Alonso. The connection between the complex-step derivative approximation and algorithmic differentiation. In *39th Aerospace Sciences Meeting and Exhibit*. American Institute of Aeronautics and Astronautics, 2001.

[24] MathWorks, Inc., Natick, Mass. *Using Matlab*, 2002. *http://www.mathworks.com*.

[25] M. Metcalf and J. Reid. *Fortran 90/95 Explained*. Oxford University Press, 1996.

[26] I. M. Navon and U. Muller. FESW - a finite-element Fortran IV program for solving the shallow water equation. *Advances in Engineering Software*, 1:77–84, 1970.

[27] The Numerical Algorithms Group Limited, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR. *The NAG Fortran Library, Mark 20, Introductory Guide*, 2002. *http://www.nag.co.uk/*.

[28] W. Press and S. A. Teukolsky. Numerical calculation of derivatives. *Computers in Physics*, 5, 1991.

[29] W. Rudin. *Real and Complex Analysis*. McGraw-Hill Science, 1986.

[30] D. Shiriaev and A. Griewank. *Computational Differentiation: Techniques, Applications, and Tools*, chapter ADOL-F: Automatic differentiation of Fortran codes. SIAM, 1996.

[31] W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Rev.*, 40:110–112, 1998.

[32] R. E. Wengert. A simple automatic derivation evaluation program. *Comm. ACM*, 7:463–464, 1964.

[33] Wolfram Research, Inc., 100 Trade Center Drive, Champaign, IL 61820-7237, USA. *http://www.wolfram.com/mathematica/*.