RAL-TR-1998-057

AD01, a Fortran 90 code for automatic differentiation*

by

J. D. Pryce[1] and J. K. Reid

Abstract

We describe the design of a Fortran 90 code called AD01 for automatic differentiation. By changing the type of each independent variable and of each variable whose value depends on the independent variables and making a small number of other changes, the user can alter an existing code into one that calculates derivatives. Both the forward and backward methods are supported. With the forward method, any derivative may be found. With the backward method, first and second derivatives are available.

Computing and Information Systems Department,
Rutherford Appleton Laboratory,
Chilton, Didcot,
Oxfordshire OX11 0QX, England.

*Also available as Royal Military College of Science report **DoIS/04/98** and by anonymous ftp from `matisa.cc.rl.ac.uk` in the directory `pub/reports` and the file `prRAL98057.ps.gz`.

[1]Software Engineering Group, Royal Military College of Science (Cranfield University), Shrivenham, Wilts SN6 8LA, England.

Contents

Glossary

f	Final dependent variable x_m.
$f_i(x_1, x_2, \ldots, x_{i-1})$	Dependence of x_i on previous variables.
$h_1, h_2, \ldots, h_n$	Increment $\mathbf{h}$ in $\mathbf{x}$.
$i_1, i_2, \ldots, i_r$	Multi-index $\mathbf{i}$ of order r.
$\mathbf{i}(i)$	Multi-index in position i of the lexographic order.
r	Derivative order.
R	Highest order of derivative required.
$\mathcal{R}$	Number of distinct derivatives of order $\leq R$.
$x_1, x_2, \ldots, x_n$	Independent variables $\mathbf{x}$.
$x_{n+1}, x_{n+2}, \ldots, x_m$	Dependent variables.
$\mathbf{x}^{\mathbf{i}}$	The product $x_{i_1} x_{i_2} \ldots x_{i_r}$.
$\bar{x}_i$	Adjoint variable $\frac{\partial f}{\partial x_i}$, $i = 1, 2, \ldots, m$.
y_i	Derivatives $\frac{\partial x_i}{\partial x_l}$, $i = 1, 2, \ldots, m$, $1 \leq l \leq n$.
$\bar{y}_i$	Adjoint variables $\frac{\partial}{\partial y_i} \frac{\partial f}{\partial x_l}$, $i = 1, 2, \ldots, m$, $1 \leq l \leq n$.
${}^nC_r$	Binomial coefficient $\frac{n!}{r!(n-r)!}$.
$D_{\mathbf{i}}$	Derivative operator associated with $\mathbf{i}$.
$T_{\mathbf{i}}$	Taylor coefficient operator $\frac{\sigma(\mathbf{i})}{r!} D_{\mathbf{i}}$.
$\sigma(\mathbf{i})$	Number of permutations of $\mathbf{i}$.

Chapter 1

Introduction

Automatic differentiation is a means of computing the derivatives of a variable for a given set of values of the independent variables. It does not seek to construct an analytic formula (symbolic algebra), but works more directly from the computer code that expresses the value of the variable. We assume that the independent variables are $x_1, x_2, \ldots, x_n$ and write the calculation as the sequence of operations

$$x_i \;=\; f_i(x_1, x_2, \ldots, x_{i-1}),\; i = n{+}1, n{+}2, \ldots m. \tag{1.1}$$

We take these as the unary or binary operations into which the compiler divides the expressions in the code (details in Section 3.1), which means that some of the variables are temporary variables. The notation may seem clumsy since each f_i is a function of only one or two variables x_k, but it covers both the unary and binary cases and allows for generalization.

The dependent variables are $x_{n+1}, x_{n+2}, \ldots, x_m$.[1] For now, we suppose that derivatives are required only of the last computed result $f = x_m$. There are two basic methods.

In the *forward* method, for each variable x_i, $i = 1, 2, \ldots, m$, the computer holds a value and a representation of all the derivatives up to the desired degree. For each operation (1.1), the derivatives of the result are calculated from those of the operands by the chain rule

$$\frac{\partial x_i}{\partial x_j} \;=\; \sum_{k=1}^{i-1} \frac{\partial f_i}{\partial x_k}\frac{\partial x_k}{\partial x_j},\;\; j = 1, 2, \ldots, n. \tag{1.2}$$

Note that, since f_i is unary or binary, the sum in equation (1.2) consists of a single term or a sum of two terms.

In the *backward* method, a *computational graph* (Rall 1981) is constructed to represent the whole computation with a node i for each variable, x_i, $i = 1$,

[1] If the code re-uses a variable, we treat each re-use as a fresh dependent variable in this mathematical description.

2, ..., m, and links to nodes for the operands of the operation. Associated with each node is storage for the value and also for the derivative of f with respect to the node variable. The values x_i, $i = n+1$, $n+2$, ..., m, are calculated in a forward pass. Our code, AD01, does this when it constructs the graph. Initially, all the variables are regarded as independent so the derivatives are all zero except at the node for f where the derivative value is 1. Backwards, one by one, the variables are changed to be dependent and the derivatives updated by the chain rule. For example, at the node corresponding to equation (1.1), we find

$$\frac{\partial f}{\partial x_k}^{new} = \frac{\partial f}{\partial x_k}^{old} + \frac{\partial f}{\partial x_i}\frac{\partial f_i}{\partial x_k}, \quad k = 1, 2, \ldots, i-1. \tag{1.3}$$

At a unary node, $\frac{\partial f_i}{\partial x_k}$ is zero for all but one value of k, so only one such update is needed. Similarly, only two are needed at a binary node. It can be seen that the work at a node during the backward pass is independent of the number of nonzero derivatives, in contrast to the forward method, where the amount of work involved in the computation (1.2) is proportional to the number of nonzero derivatives of the x_i.

The derivatives of f are also known as *adjoints* $\bar{x}_k$. We find it convenient sometimes to discard the superscripts *new* and *old* and use the assignment notation

$$\bar{x}_k := \bar{x}_k + \bar{x}_i\frac{\partial f_i}{\partial x_k}, \quad k = 1, 2, \ldots, i-1. \tag{1.4}$$

Whether the forward or backward method is better depends on many factors, including the number of derivatives required, the degree, the number of independent variables, and the sparsity of the derivatives. We have therefore decided to provide both the forward and backward method, with an easy mechanism for changing from one to the other.

Our work is based on the code DAPRE of Stephens and Pryce (1991) (see also Davis, Pryce and Stephens 1990). Stephens worked in Fortran 77 and used a pre-processor to convert user code that calculates variables to code that calculates both variables and derivatives. The output code makes extensive use of explicit procedure calls and is hard to read. This approach is similar to that of ADIFOR (Bischof, Carle, Corliss, Griewank and Hovland 1992), except that ADIFOR generates in-line code to calculate first derivatives. Fortran 90 supports derived types and operator overloading, which allows the procedure calls to be represented by the same symbols as for the original calculation of variables.

We have therefore avoided having a pre-processor and instead require the user to make some easy changes to the source code. All the independent variables, and appropriate variables that depend on them (see Section 3.4), must be declared to be of the derived type `ad01_real`. We refer to all such variables as *active* variables.

We collect the type definitions, the associated procedures, and everything else needed into a module. There are four versions, for forward or backward differentiation in single or double precision arithmetic. We have employed the same set of names in each, to permit the user to switch from one to another very easily. In the code, only the `use` statement needs alteration. For guidance and examples of the code conversion process, turn to Section 3.4.

Most of the early work of converting DAPRE into AD01 was done by David Cowey, under an EPSRC CASE PhD studentship supervised by the authors.

The code itself is included in Release 12 of the Harwell Subroutine Library (HSL 1995) and is available from AEA Technology, Harwell. The contact is Mrs Maria Woodbridge, AEA Technology, Building 477, Harwell, Didcot, Oxfordshire OX11 ORA, England; email: `maria.woodbridge@aeat.co.uk`; telephone: (44) 1235 432345; Fax: (44) 1235 432023.

The aim of this paper is to explain the mathematical foundations of this work and the reasoning behind the design, and to show some examples of its use. The forward method permits derivatives of any order to be calculated, subject to limits on time and storage. Other packages, such as ADOL-C (Griewank, Juedes and Utke 1996), COSY-INFINITY (Berz 1991*b*, Berz 1991*a*) and Neidinger's APL implementation (Neidinger 1992), also have this capability. The backward method computes derivatives up to order two, see Section 2.2. The approach for this is similar to that of Christianson (1991) but was developed independently by Stephens.

The paper is organized as follows. We explain the mathematics of our approach in Chapter 2, with the forward and backward methods in Section 2.1 and Section 2.2, respectively. In Chapter 3, Section 3.1 describes the subset of Fortran that is supported by the package, and Sections 3.2 and 3.3 describe support for 'subsidiary' differentiated calculations. Section 3.4 gives guidance on the code conversion process. In Chapter 4, Section 4.1 describes the data structures used in the implementation. Section 4.2 explains how the computational graph for the backward method is constructed and used. Section 4.3 explains how the forward method computes derivatives of arbitrary order. Section 4.4 describes the exception-handling features. Chapter 5 describes the tests of correctness and performance that have been made. Chapter 6 contains our conclusions and plans for further development, as well as thanks and acknowledgements.

Chapter 2

Mathematical foundations

In this chapter we review the mathematics underlying the forward method (Section 2.1) and the backward method (Section 2.2).

2.1 The forward method

Any floating-point computation can be broken down into a sequence of unary operations such as $-a$ or $\sin(b)$ and binary operations such as $a + b$ or $\max(a, b)$. For each such operation that we support (details in Section 3.1), we provide a procedure that implements the operation for one or two `ad01_real` objects. For each such object, the module holds in its data structure (details in Subsections 4.1.1 and 4.1.3) all the required derivatives. The procedure uses these derivatives to calculate the result of the operation and all its required derivatives. We begin by considering the case when only first derivatives are wanted.

2.1.1 First derivatives

In the forward method for first derivatives, the gradient ∇x_i of each variable with respect to the n independent variables is stored after being calculated by equation (1.2). We remind the reader that the sum in equation (1.2) has only one term for a unary operation and two for a binary operation.

We may take account of sparsity in the derivatives by holding, with each variable, a list of the independent variables upon which it depends. We refer to this list as its *sparsity pattern*. Initially, we have the independent variables x_i, $i = 1, 2, \ldots, n$, each of which depends on no other independent variables so has only one nonzero derivative, namely $\nabla x_i = e_i$, the ith unit n-vector. Each unary operation preserves the sparsity since it introduces no additional dependencies. However, the sparsity patterns merge when a binary operation is applied since the result is dependent on any variable that either of the operands depended upon. Exact cancellation is a possibility, leading to a zero derivative value being held explicitly, but is unlikely to occur sufficiently frequently to warrant testing for its occurrence. This merging of patterns means that we start with very sparse derivatives but must expect them to become fuller as the calculation progresses. There comes a point when it is more efficient to ignore the sparsity, so our codes automatically switch the storage mode when a threshold is reached. The threshold itself is discussed in Subsection 4.1.1.

For each unary and binary operation supported (see Section 3.1), we provide a suitable procedure. Each calculates the derivatives of the result. In the case of a binary operation, a merge of sparsity patterns may be needed, too.

2.1.2 Higher-order derivatives

When working with higher-order derivatives, it is convenient to use vector subscripts and superscripts, which we call *multi-indices*. If a variable a depends on the independent variables x_{i_k}, $k = 1, 2, \ldots, r$, a multi-index is an integer vector

$$\mathbf{i} \;=\; (i_1, i_2, \ldots, i_r), \tag{2.1}$$

with each index in the range $1 \le i_k \le n$, $k = 1, 2, \ldots, r$. This allows us to use the notation

$$D_{\mathbf{i}}a \;=\; \frac{\partial^{\,r} a}{\partial x_{i_1} \partial x_{i_2} ... \partial x_{i_r}} \tag{2.2}$$

for a derivative of order r and the notation

$$\mathbf{x}^{\mathbf{i}} \;=\; x_{i_1} x_{i_2} \ldots x_{i_r} \tag{2.3}$$

for a monomial of degree r. Since permutations of the indices in a multi-index have no effect on the corresponding derivative or monomial, it suffices to store only the derivatives corresponding to the multi-indices satisfying the inequalities

$$i_1 \ge i_2 \ge \ldots \ge i_r. \tag{2.4}$$

We use the notation

$$\sigma(\mathbf{i}) \tag{2.5}$$

for the number of permutations of $\mathbf{i}$ (which correspond to identical derivatives or monomials). For example, (2,1,1) has permutations (1,2,1) and (1,1,2), so $\sigma((2,1,1))$ has the value 3. If $\mathbf{i}$ has l groups of identical indices and the number of indices in the groups are $n_1, n_2, \ldots, n_l$, we find the relation

$$\sigma(\mathbf{i}) \;=\; \frac{r!}{n_1! n_2! \ldots n_l!}. \tag{2.6}$$

The number of multi-indices of order r (distinct derivatives of order r) is

$$^{n+r-1}C_r = \frac{(n+r-1)!}{r!\,(n-1)!} \tag{2.7}$$

Note that, for large n, this behaves as $O(n^r/r!)$, so the storage gain over holding all derivatives is by a factor of about $r!$.

The total number of multi-indices of orders up to R is

$$\mathcal{R} \;=\; \sum_{r=0}^{R} {}^{n+r-1}C_r \;=\; {}^{n+R}C_R. \tag{2.8}$$

It is convenient to order them by increasing degree and according to a lexographic ordering of multi-indices within each degree. For example, if $n = 3$, this corresponds to the following multi-index ordering

$$\begin{array}{l}(),(1),(2),(3),(1,1),(2,1),(2,2),(3,1),(3,2),(3,3),\\ (1,1,1),(2,1,1),(2,2,1),(2,2,2),\ldots\end{array} \tag{2.9}$$

The position of $\mathbf{i} = (i_1, i_2, \ldots, i_r)$ in this sequence is

$$^{n+r-1}C_{r-1} + {}^{i_1+r-2}C_r + {}^{i_2+r-3}C_{r-1} + \ldots + {}^{i_r-1}C_1 + 1. \tag{2.10}$$

It is not often necessary to compute these positions directly, but to speed up the calculation when they are, an array of binomial coefficients is precomputed when a computation is initialized. We use the notation

$$\mathbf{i}(i) \tag{2.11}$$

for the i-th multi-index in this sequence; for example, if $n = 3$, $\mathbf{i}(5)$ is the multi-index (1,1).

We use this order for the derivatives, which we actually hold as Taylor coefficients

$$T_{\mathbf{i}}a \;=\; \frac{\sigma(\mathbf{i})}{r!} D_{\mathbf{i}}a \tag{2.12}$$

because this makes the Taylor expansion take the form

$$a(\mathbf{x}+\mathbf{h}) = \sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)}a\, \mathbf{h}^{\mathbf{i}(i)}, \tag{2.13}$$

which may also be written as

$$a(\mathbf{x}+\mathbf{h}) = a(\mathbf{x}) + \sum_{i=2}^{\mathcal{R}} T_{\mathbf{i}(i)} a\, \mathbf{h}^{\mathbf{i}(i)} \tag{2.14}$$

since $\mathbf{i}(1) = ()$.

Sparsity among the derivatives is even more helpful here than when only first derivatives are required (Subsection 2.1.1). If a variable depends on $x_i, i = \lambda_1, \lambda_2, \ldots, \lambda_s$, we hold the list $(\lambda_1, \lambda_2, \ldots, \lambda_s)$ and each index j in a multi-index will refer to x_{λ_j} rather than x_j.

Alternative ways to compute high-order multivariate derivatives, by clever use of univariate Taylor series, are studied by Griewank and Utke (1996), and by Bischof, Corliss and Griewank (1993).

2.2 The backward method

The backward method for first derivatives was explained in Chapter 1. It may be applied to any dependent variable. That is, f is not restricted to the final variable. In general, we need to set to zero the adjoints (defined in Chapter 1) of all variables that precede f in the computational sequence, set the adjoint of f to unity, and proceed backwards from the node for f updating the adjoints using the assignment (1.4). Thus, we can provide a procedure to calculate the derivatives of any variable, just as for the forward method. However, here real work has to be done, as opposed to the simple look up needed for the forward method. The number of derivatives required therefore affects the balance between the effectiveness of the two methods.

In Subsection 2.2.1, we give an alternative derivation of the backward method for first derivatives, similar to that of Rall (1981). In Subsection 2.2.2, we explain how the backward method may be extended to second derivatives.

2.2.1 A matrix view of the backward method

In the forward method at node i, $n{+}1 \le i \le m$, we have the calculation of equation (1.2). This may be represented as forward substitution on the matrix equation

$$L\, d_j \;=\; e_j, \quad j = 1, 2, \ldots, n, \tag{2.15}$$

where

d_j is the m-vector of derivatives $\partial x_k / \partial x_j$, $k = 1, 2, \ldots, m$;

L is the m by m unit lower-triangular matrix whose only nonzero off-diagonal entries are $l(i,k) = -\partial f_i/\partial x_k$, $i = n{+}1, n{+}2, \ldots, m$; $k = 1, 2, \ldots, i{-}1$; and

e_j is the jth unit vector of order m.

Only one or two of the off-diagonal entries in a row are nonzero. L may be written as

$$L = L_{n+1}L_{n+2}\ldots L_m \tag{2.16}$$

where L_i is the unit lower triangular matrix whose row i is the same as row i of L and which has no off-diagonal entries in other rows. L_i^{-1} is unit lower triangular with the same structure as L_i, but the signs of the off-diagonal entries are reversed. The backsubstitution (2.15) can therefore be written as

$$d_j = L_m^{-1}L_{m-1}^{-1}\ldots L_{n+1}^{-1}e_j. \tag{2.17}$$

In the forward method, we actually calculate all the derivatives:

$$D = L_m^{-1}L_{m-1}^{-1}\ldots L_{n+1}^{-1}\begin{pmatrix} I \\ 0 \end{pmatrix} \tag{2.18}$$

where D is the matrix whose j-th column is d_j and I has order n. For any particular variable x_k, the derivatives are available as row k of D. Alternatively, they may be calculated directly as

$$e_k^T L_k^{-1}L_{k-1}^{-1}\ldots L_{n+1}^{-1}\begin{pmatrix} I \\ 0 \end{pmatrix} \tag{2.19}$$

The backward method consists of computing (2.19) from the left:

$$b_k^T = e_k^T \tag{2.20}$$
$$b_i^T = b_{i-1}^T L_i^{-1},\ i = k, k{-}1, \ldots, n{+}1. \tag{2.21}$$

It is clear that the number of arithmetic operations is proportional to the number of nonzero local derivatives $\partial f_i/\partial x_k$.

2.2.2 The backward method for second derivatives

To find second derivatives, we treat the calculation of the first derivative $\partial f/\partial x_l$, $1 \le l \le n$, as a primary calculation to which we apply the backward method. The approach was devised by Stephens (Stephens and Pryce 1991) and is similar to that used by Christianson (1991) and Dixon (1991).

It is simplest to use the forward method as the starting point. Using the notation

$$y_i = \frac{\partial x_i}{\partial x_l}, \quad i = 1, 2, \ldots, m, \tag{2.22}$$

the forward calculation is

$$\begin{aligned} x_i &= f_i && (2.23)\\ y_i &= \sum_{k=1}^{i-1} y_k \frac{\partial f_i}{\partial x_k}, && (2.24)\end{aligned}$$

for $i = n{+}1, n{+}2, \ldots, m$. For the backward calculation, we have the adjoint variables

$$\bar{x}_i = \frac{\partial}{\partial x_i}\frac{\partial f}{\partial x_l}, \quad \bar{y}_i = \frac{\partial}{\partial y_i}\frac{\partial f}{\partial x_l}, \quad i = 1, 2, \ldots, m. \tag{2.25}$$

The backward calculations corresponding to equation (2.23) are

$$\bar{x}_j := \bar{x}_j + \bar{x}_i \frac{\partial f_i}{\partial x_j}, \quad j = 1, 2, \ldots, i{-}1 \tag{2.26}$$

and those corresponding to equation (2.24) are

$$\begin{aligned} \bar{y}_k &:= \bar{y}_k + \bar{y}_i \frac{\partial f_i}{\partial x_k}, \quad k = 1, 2, \ldots, i{-}1 && (2.27)\\ \bar{x}_j &:= \bar{x}_j + \bar{y}_i y_k \frac{\partial^2 f_i}{\partial x_k \partial x_j}, \quad j, k = 1, 2, \ldots, i{-}1. && (2.28)\end{aligned}$$

for $j = m, m{-}1, \ldots, n{+}1$. Of course, for a unary f_i only one of each of the calculations (2.26) to (2.28) is non-trivial. For a binary f_i, only two of each of the calculations (2.26) and (2.27) and four of (2.28) are non-trivial.

The calculation starts with $\bar{y}_m = 1$, $\bar{y}_i = 0$, $i = 1, 2, \ldots, m{-}1$ and $\bar{x}_i = 0$, $i = 1, 2, \ldots, m$. The calculation (2.27) is independent of l, so need only be performed once. Actually, it is the backward method for the first derivatives, so the $\bar{y}_i$ hold $\frac{\partial f}{\partial x_i}$ at the end.

For each l, $1 \le l \le n$, we therefore perform the forward calculation (2.24) starting with $y_l = 1$, $y_i = 0, i = 1, 2, \ldots, n$, $i \ne l$, then set $\bar{\mathbf{x}} = \mathbf{0}$ and perform the calculations (2.26) and (2.28) backwards ($j = m, m{-}1, \ldots, n{+}1$).

Since $\bar{\mathbf{y}}$ is known, an alternative is to perform the calculation (2.28) forwards during the calculation of $\mathbf{y}$, which is what we have chosen to do in AD01.

The same calculation may be regarded as the backward method applied to the backward method for first derivatives. Now the roles of $\mathbf{y}$ and $\bar{\mathbf{y}}$ are interchanged, but the calculations are identical.

Chapter 3

The AD01 Interface

3.1 Language supported

Our aim is to support most Fortran 77 programs. Perhaps the biggest exclusion is of complex arithmetic, which was forced upon us by the many additional procedures that this would have involved. We can also support many Fortran 90 programs. Derived types and modules present no difficulties, but we have not had the resources to support the array features or the new intrinsic procedures.

We provide separate modules for single and double precision arithmetic. To avoid complicated data structures in the double precision module, we have chosen to store all its real values in double precision, but we do support binary operations and assignments between single-precison inactive variables and active variables (of type `ad01_real`). The user of a mixed precision program may therefore use the double precision module by changing the type of any active variable (single or double) to `ad01_real`.

Each module provides overloaded procedures for scalar assignments and intrinsic operations and procedures where they involve one or two active variables. More details are given in the rest of this section. Note that the operands may be array elements or structure components.[1]

Active scalar variables are given values by assignments from expressions involving active scalar variables whose values have been previously found, and inactive scalars. The inactive objects may be of type `real` (single or double in the double precision module) or `integer`. They are treated as invariants for the purpose of differentiation, but they may be Fortran variables.

Each module supports the binary operators `+`, `-`, `*`, `/`, `**` for two scalars

[1]However, the current definition of Fortran specifies an unexpected behaviour for intrinsic assignment of a structure with a component of type `ad01_real`. Intrinsic assignment will be applied when derived assignment needed. This is expected to be altered in a corrigendum, but in the meantime the problem may be averted by the user defining assignment for the type with a component of type `ad01_real`.

where one is of type `ad01_real` and the other is of type `ad01_real`, `real`, or `integer`. This means that for each operation, five procedures are needed in the single precison module and seven procedures are needed in the double precison module. However, most of them are very short subprograms that make simple tranformations of the arguments, such as interchanging them or converting the type of an inactive one, then calling another subprogram. `ad01` also supports the unary operator `-` for a scalar of type `ad01_real`. In all these cases, the result is a scalar of type `ad01_real`.

Each module supports the binary operators `==`, `/=`, `>`, `>=`, `<`, `<=` for two scalars where one is of type `ad01_real` and the other is of type `real`, or `integer`. The result is always a scalar of type `ad01_logical`.

Assignment to a scalar of type `ad01_real` from a scalar of type `ad01_real`, `real`, or `integer` is accepted, as is assignment to a scalar of type default `integer` from a scalar of type `ad01_real`. However, assignment to `real` is not provided since such an assignment is likely to be erroneous; if the user really means this, the subroutine `ad01_value` is available to return the `real` value of an active variable.

These are all defined assignments, even for assigning one `ad01_real` scalar to another. This is because an `ad01_real` is essentially an integer pointer to where the data is held in a structure. Intrinsic assignment would copy the pointer, which would be erroneous, instead of copying the data. For this reason and because AD01 does not yet support whole-array operations, we have defined assignment to an `ad01_real` array from an `ad01_real` array or scalar, since otherwise such an assignment will be interpreted by the compiler as intrinsic assignment. The defined assignment raises an exception (see Section 4.4) and sets each array element to an undefined value.

The functions `abs`, `acos`, `asin`, `atan`, `cos`, `cosh`, `exp`, `log`, `log10`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh` are provided for a scalar of type `ad01_real`. The result is of type `ad01_real`. The functions `aint` and `anint` for a scalar of type `ad01_real` are provided. The result is of type `real`.

The functions `atan2`, `max`, `min`, and `sign` are provided for two scalars of type `ad01_real` or one scalar of type `ad01_real` and one scalar of type `real`. The result is of type `ad01_real`. We have chosen not to go beyond two arguments for `max` and `min` because of the large number of explicit subprograms that would be needed and because the two-argument case is by far the most common. Also, users can make nested invocations such as `min(a,min(b,c))`.

The functions `int` and `nint` are provided for a scalar of type `ad01_real`. The result is of type default `integer`.

As well as overloading these generic names of intrinsics, we also overload explicit names such as `dlog` in the double precision version and `alog` in the single precision version. This leads to a total of 122 functions in the double precision module and 90 functions in the single precision module.

3.2 Additional unary functions

We permit the user to code additional unary functions $u(x)$. Such a function must have an `ad01_real` argument `x` and return an `ad01_real` result `u`. It must find the value of the argument as an ordinary real x, calculate $u(x)$ and its derivatives $u'(x)$, $u''(x)$, ...up to the required order and place them in a real array `derv`, and finally make a call of the form

```
call ad01_user(derv,x,u)
```

In the backward method, this sets up a new node of the computational graph and stores the derivatives for it. For the forward method, the derivatives are calculated (see Subsection 4.3.4 for further details). The function can then be used exactly like the intrinsics `exp`, `sin`, etc.

If such a function is used frequently, this results in considerable savings of space and time. As an example, consider the evaluation of a polynomial by Horner's rule:

$$p_n = a_n,\ p_{i-1} = xp_i + a_{i-1},\quad i = n, n-1, \ldots, 1. \tag{3.1}$$

With forward differentiation, storage is needed for all the variables p_i and their derivatives and also for all the intermediate variables xp_i and their derivatives. This may be very substantial if there are many independent variables and many derivatives are required. The work is proportional to the storage. With backward differentiation, the overheads are not so great, but all these variables must be recorded in the computational graph. Alternatively, the user may calculate the polynomial and its local derivatives, as shown in Figure 3.2. Now all the intermediate calculations are performed with real variables. With forward differentiation, storage is needed only for the result and its derivatives. With backward differentiation, only one node need be recorded in the computational graph.

Another application is where the program uses special functions such as Bessel functions. Here it is usually *essential* to use the above technique, for the following reason. The code to compute the special function $u(x)$ probably uses methods such as economized polynomials, that produce an approximation $p(x)$ such that $|u(x) - p(x)|$ is guaranteed to be small, but whose derivatives may bear no relation to those of $u(x)$. The user should not attempt to differentiate the source code even if it is available. Instead, a recurrence based on the defining differential equation, or similar, should be used to calculate the derivatives.

3.3 Subsidiary calculations

It was simple enough to hand code the derivative calculations shown in Figure 3.2, but in a more complicated case it may be very advantageous to

```
function u(x)
! A function in a module that access the required ad01 module
!   and provides the integers degree and n and the real array
!   coeff.
    type(ad01_real) :: u
    type(ad01_real), intent(in) :: x
    real :: derv(0:degree),rx
! Find the value as a real.
    call ad01_value (x,rx)
! Calculate the derivative values
    derv(0) = coeff(n)
    derv(1:degree) = 0.0
      do i = n, 1, -1
        do l =  degree, 1, -1
           derv(l) = l*derv(l-1) + derv(l)*rx
        end do
        derv(0) = derv(0)*rx + coeff(i-1)
     end do
! Pass the derivative values to the ad01 module
     call ad01_user(derv, x, u)
end function u
```

Figure 3.1: Evaluating a polynomial and its derivatives

use automatic differentiation. We therefore provide a facility for suspending the calculation for a subsidiary computation. At a given moment, there is an active calculation, whose data are held in the module, and any number of suspended calculations for each of which the data is held in a variable of a special type called `ad01_data`. A calculation is suspended by a call of `ad01_store`, which stores all the module data in an `ad01_data` variable. This allows the user to start a new calculation with an `ad01_initialize` call or resume a previous calculation by using an `ad01_restore` call to restore the module data from an `ad01_data` variable. Since pointer assignments are used, these save and restore operations are not hopelessly expensive.

3.4 Code conversion examples

The full interface specification and more examples are given in the user documentation (Harwell Subroutine Library 1996), see the Appendix. Our aim here is to give some guidance on the process of converting code to use AD01.

Normally the requirement is to 'differentiate' a section of code in the middle of a larger program—it may be purely inline, or may call one or more subprograms. As said in Chapter 1, the independent variables and variables that depend on them must be made *active*, that is declared to be of type `ad01_real`. As a consequence, each operation involving them is performed by an `ad01` procedure. Variables whose values do not depend on the independent variables should remain as reals. They may be made active, but this is wasteful since their derivatives are always zero.

Unless there is only one independent variable, AD01 requires the independent variables to be collected into a rank-one array. This keeps the interface simple and does not represent a serious loss of generality since the programmer can follow the call to `ad01_initialize` by assignments to variables of his or her choice, for example,

```
:
call ad01_initialize(level,x,xval)
a = x(1)
b = x(2)
c = x(3)
:
```

A normal run consists of a call to `ad01_initialize` to initialize the module data, specify the order of the highest derivative required, specify the independent variables, and provide values for the independent variables. There follows the main computation in which the values of the dependent variables are calculated. Finally, procedure calls are made to obtain derivative values. Derivatives of any active scalar variable are available (we can temporarily regard the computation as having been terminated when the final assignment to that variable is made). For cases where few of the independent variables affect a dependent variable, there are facilities to return the derivatives in packed form. If derivatives are required for another set of values of the independent variables, the process must be repeated, including the call to `ad01_initialize`.

3.4.1 A simple approach to conversion

In this subsection, we explain how a code may be converted to use AD01. For simplicity, we assume here that no components of structures need to be made active. We discuss that case in Subsection 3.4.2.

The basic rule for making a section of code active is: if the value of any part of a named variable may depend on the independent variables, via assignment or via argument association in a procedure call (including function results), then that named variable must be made active. The active section may call procedures, which may themselves call further procedures. This means that, although it is easy in principle, the conversion process needs some care. It can be described as follows.

Algorithm 3.4.1 (Making an active section)

```
1.  Make the independent variables active. If there is more than one and
    they are not already collected into a rank-one array, declare such an
    array and add assignments from this to the independent variables.
2.  Starting in the scope where active variables first appear,
    and recursively through all procedures encountered, do steps 3 to 5.
    3.  If the right-hand side of an assignment statement is active, ensure that
        the left-hand-side variable is active,
    4.  If a procedure call has an actual argument that is active, ensure that
        the corresponding dummy argument is active.
    5.  If a procedure called has a dummy argument that is active, ensure that
        the corresponding actual argument is active.
```

The compiler can assist in the conversion process, provided the interface of each procedure involved is explicit (good programming practice anyway), that is, either the procedure is a module or internal procedure, or it has an interface block. Then:

- Since assignment `real = ad01_real` is not supported by AD01, if a variable on the left-hand side of an assignment needs to be activated, this will be flagged by an 'unsupported assignment' compiler error. This includes the case where a function returns an active result.
- Dummy or actual arguments that need to be activated will be flagged by an 'argument type mismatch' error.

This gives the following method.

Algorithm 3.4.2 (Conversion with compiler help)

```
1.  Start by declaring the independent variables as active. If necessary,
    declare an active rank-one array for the independent variables and add
    assignments from it.
2.  Declare as active any variable that causes an unsupported assignment or
    or argument type mismatch compilation error.
3.  Continue until there are no errors.
```

There is a simple way to view the above process in graph terms. In order to find out which named variables need to be made active, define a *static dependency graph* G. The nodes of G are the named variables used in the code and the actual arguments that are expressions other than variables. An array is treated as a single named variable and each dummy variable has its own node. The arcs are defined as follows. If any part of variable y appears on the left-hand side of an assignment and any part of variable x appears on the right-hand side in such a way as to cause the whole right-hand-side expression to be active, draw an arc $x \rightarrow y$ in G. If any part of variable x

appears on in an actual argument expression in such a way as to cause the whole expression to be active, draw an arc from x to the expression's node. The dependency graphs of different scoping units form disjoint subgraphs of the whole graph G. Each call from scoping unit P to scoping unit Q defines a *two-way* dependency link between each actual argument that is a variable or an expression and the corresponding dummy argument. A function result is regarded as one of the variables in the expression where it appears.

Then, the above rules are equivalent to the rule: a named variable y is active if and only if there is a path in G from an independent variable to y.

Example 3.4.1 Figure 3.2 shows a simple example with no subprograms. Interspersed with the original code, the changes needed to incorporate AD01 are shown in comments. `!a` means add a statement; `!r` means replace the previous statement. The program computes Rosenbrock's function (5.1), for a value of n input by the user, at the point $\mathbf{x}$ given by $x_i = 1 + 1/i$, $i = 1, \ldots, n$. To compute the gradient of f with respect to `x` at the given point we declare f, `x` as active and insert calls to initialize `x` before evaluating f, and to extract the required values afterward. The variables `xval, fval, g_f` handle this conversion. `x` is replaced by `xval` in the loop that defines the input values, then the call to `ad01_initialize` sets up the vector `x` of input active variables. At the end, calls to `ad01_value` and `ad01_grad` do the reverse process.

3.4.2 Structures with active components

We now discuss the case where some active variables are components of structures. For each such type, we need to declare a new type for which the real component is replaced by an component of type `ad01_real`.[2] We refer to this as the active type corresponding to the original inactive type. The concept may need to be nested, that is, we may need a new type for a component of derived type that becomes active. With this interpretation of how a named variable of derived type is made active, the rules of the previous section suffice.

3.4.3 Being more selective

There are several reasons for having fewer active variables than the rules of Subsection 3.4.1 prescribe. It may be, as in this subsection, purely for

[2]The current definition of Fortran specifies an unexpected behaviour for intrinsic assignment of structures with a component of derived type. Intrinsic assignment will be applied when derived assignment is needed. This is expected to be altered in a corrigendum, but in the meantime the problem may be averted by the user defining assignment for each such type.

Figure 3.2: Program for Rosenbrock function before and after conversion

```
program rosen
!a use HSL_AD01_FORWARD_DOUBLE
   integer,parameter::wp=kind(0d0)
   integer i,n
   real(wp) f
!r type(ad01_real):: f = ad01_undefined
!a real(wp) fval
   real(wp), allocatable:: x(:)
!r type(ad01_real), allocatable:: x(:)
!a real(wp),allocatable::xval(:),g_f(:)
   print*, 'Give n:'
   read*, n
   allocate(x(1:n))
!a allocate(xval(1:n),x(1:n),g_f(1:n))

! Initialize independent variables
   do i=1,n
     x(i) = 1.0_wp + 1.0_wp/i
!r   x(i) = ad01_undefined
!a   xval(i) = 1.0_wp + 1.0_wp/i
   end do
!a call ad01_initialize(1,x,xval)

!Compute Rosenbrock function:
  f = 0.0_wp
  do i=1,n-1
  f=f+(10._wp*(x(i+1)-x(i)**2))**2+(x(i)-1.0_wp)**2
  end do

! Print results
!a call ad01_value(f, fval)
!a call ad01_grad(f, g_f)
   print*, 'Function value f=', f
!r print*, 'Function value f=', fval
!a print*, 'Gradient g_f=', g_f

end
```

efficiency; or, as in the next, that differentiating along certain dependency paths is not what the mathematics requires and may harm an algorithm's performance.

Here, suppose it is known which variables are to be treated as the dependent (output) variables—there may be others that depend on the inputs but whose derivatives are not of interest. To avoid needlessly declaring variables as active, revise the rule for 'activating' variables as below. In the following algorithm, the function `ad01_val` is a 'function version' of the package subroutine `ad01_value`. It is useful in this situation, and we give the code for it as part of Example 3.4.2 below (but note that this code does not include a check for an error condition). It is not provided as part of AD01 because it would not give the user the ability to recover from error conditions.

Algorithm 3.4.3 (Selective activation)

1. Declare as active any named variable that lies on some path *from* an input variable *to* an output variable in the graph G defined above.
2. For any assignment `y=expr` where `y` is to remain inactive, and expression `expr` is active, replace each active variable `x` by `ad01_val(x)`,

which 'de-activates' it.

Warning! This must be done with care. If you break a path of dependency by wrongly inserting `ad01_val()` *somewhere along it then the derivatives will be calculated wrongly.* Especially if subprogram calls are involved, it is important to understand the structure of the relevant part of the code and its information flow, before performing this process.

If making changes to the code which may alter the dependency structure, you are advised to remove all `ad01_val()` calls and start afresh.

Occurrences of active variables in the condition of an `if` and similar situations need not be replaced by their `ad01_val`s. However if the condition involves an active expression, such as `if (cos(x+y) < 0.0)`, then evaluating it with x and y active involves much unnecessary calculation if the number of independent variables and/or order of derivative is large.

3.4.4 Being more selective: discretizations

When differentiating the discretization of a differential equation, it is generally unnecessary and often harmful to differentiate aspects of the discretization strategy, such as timestep control or adaptive space mesh selection. Abstractly, the solution can be regarded as a function $u = f(p, s)$ where p are the parameters whose effect on u we seek and will be the active variables, while s is the data which defines the discretization. We have $s = s(p)$ since the discretization generally changes when the parameters change. But $\partial u/\partial p$ should be computed with s held constant (with a *frozen strategy*). That is, we want to compute $\partial f/\partial p$ rather than the unnecessarily expensive total derivative $\frac{du}{dp} = \frac{\partial f}{\partial p} + \frac{\partial f}{\partial s}\frac{ds}{dp}$. A theoretical reason for this was given long ago by Ortega and Rheinboldt (1970).

We achieve this by 'deactivating' input into those parts of the code that deal with discretization. The next example shows this for a simple code to solve the initial value problem for an ordinary differential equation (ODE) $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$, where $\mathbf{y}$ is an n-vector function of the scalar independent variable t.

Example 3.4.2 The ODE code in this example uses Euler's method as predictor, and the trapezoid rule as corrector, to take a step of length h from computed point $(t_j, \mathbf{y}_j)$ to new computed point $(t_{j+1}, \mathbf{y}_{j+1})$. It is defined by these formulas, where $\mathbf{y}_j, \mathbf{f}_j$ are n-vectors:

$$\begin{aligned} \mathbf{y}^p_{j+1} &= \mathbf{y}_j + h\mathbf{f}_j \quad \text{where } \mathbf{f}_j = \mathbf{f}(t_j, \mathbf{y}_j) & (3.2) \\ \mathbf{f}^p_{j+1} &= \mathbf{f}(t_{j+1}, \mathbf{y}^p_{j+1}) & (3.3) \\ \mathbf{y}_{j+1} &= \mathbf{y}_j + \frac{h}{2}(\mathbf{f}_j + \mathbf{f}^p_{j+1}) & (3.4) \\ \mathbf{f}_{j+1} &= \mathbf{f}(t_{j+1}, \mathbf{y}_{j+1}) & (3.5) \end{aligned}$$

At each step it takes the difference between predicted $\mathbf{y}$ and corrected $\mathbf{y}$ as error estimate vector $\mathbf{e}$, compares this against a given tolerance, and adjusts the stepsize by multiplying it by the factor ρ where

$$\rho = \left(\frac{0.9 * \mathrm{TOL} * (1 + \max_i |y_i|)}{\max_i |e_i|} \right)^{1/2} ; \ \mathbf{e} = \mathbf{y}^p_{j+1} - \mathbf{y}_{j+1}, \ \mathbf{y} \text{ denotes } \mathbf{y}_{j+1}.$$

A simple program which does this for the system $y_1' = -y_2$, $y_2' = y_1$ is shown in Figure 3.3 and Figure 3.4, with the alterations to incorporate AD01 given in comments as for the previous example. Subroutine `step` attempts to perform one step, rejecting it if the error estimate is too large and accepting it otherwise. The variables `errest`, `hfactor` and `h` are the 'strategy' quantities which should not be differentiated, and have been left inactive using the technique outlined above. The `ad01_initialize` call makes the program differentiate (once only) with respect to the two initial values y_1, y_2. For brevity we omit: declaration of an $n \times n$ array `yjac` to hold the Jacobian; calls of `ad01_grad` to extract its value one row at a time which would be adjacent to the `ad01_value` calls; and any print statements. The program is for example only and omits many aspects of robust ODE solving.

3.4.5 Other cases

Another important situation where it is harmful to differentiate code as it stands is an iteration to convergence which is intended to solve one or more

Figure 3.3: Simple ODE solver before and after conversion

```
program ode
!a  use HSL_AD01_FORWARD_DOUBLE
   integer,parameter :: wp=kind(0d0),n=2
   real(wp),parameter :: pi=3.14159265358979_wp
   real(wp)::h,t,tend,tol
   real(wp),dimension(1:n) :: f,y
!r type(ad01_real) :: f(1:n)=ad01_undefined, y(1:n)=ad01_undefined
!a real(wp) yval(1:n)
   t=0._wp
   tend=2.0_wp*pi
   y = (/1._wp,0._wp/)
!r call ad01_initialize(1,y,(/1._wp,0._wp/))
   call fcn(t,y,f)
   tol=0.001_wp
   h=0.01_wp
!a call ad01_value(y,yval)
   do while(t<tend)
     call step(t,y,f,h,tol)
!a   call ad01_value(y,yval)
   end do

contains

   subroutine step(t,y,f,h,tol)
     real(wp)::h,t,tol
     real(wp)::f(1:n),y(1:n)
!r   type(ad01_real)::f(1:n),y(1:n)
     integer i
     real(wp)::err_actual, err_target,hfactor
     real(wp),dimension(1:n):: errest
     real(wp),dimension(1:n)::f_pred,y_pred,y_trial
!r   type(ad01_real),dimension(1:n)::f_pred=ad01_undefined, &
!a                 y_pred=ad01_undefined,y_trial=ad01_undefined
     do i=1,n
       y_pred(i)=y(i) + h*f(i)
     end do
     call fcn(t,y_pred,f_pred)
```

Figure 3.4: Simple ODE solver before and after conversion, continued

```
     do i=1,n
       y_trial(i)=y(i) + (h*0.5_wp)*(f(i) + f_pred(i))
       errest(i)=y_trial(i)-y_pred(i)
!r     errest(i)=ad01_val(y_trial(i))-ad01_val(y_pred(i))
     end do
     err_target=tol* (maxval(abs(y))+1_wp)
!r   err_target=tol* (maxval(abs(ad01_val1(y)))+1._wp)
     err_actual=maxval(abs(errest))
     if (err_actual <= err_target) then
       t=t+h
       y=y_trial
       call fcn(t,y,f)
     end if
     hfactor=(0.9_wp * err_target/err_actual)**0.5_wp
     h=h*min(2._wp,hfactor)
  end subroutine step
  subroutine fcn(t,y,f)
    real(wp)::t
    real(wp)::y(1:n),f(1:n)
!r  type(ad01_real)::y(1:n),f(1:n)
    f(1)=-y(2)
    f(2)=y(1)
  end subroutine fcn

  function ad01_val(x) !when x is a scalar
    real(wp) ad01_val
    type(ad01_real) x
    call ad01_value(x,ad01_val)
  end function ad01_val

  function ad01_val1(x) !when x is a rank-one array
    type(ad01_real) x(:)
    real(wp) ad01_val1(1:size(x))
    call ad01_value(x,ad01_val1)
  end function ad01_val1
end
```

equations. Typically these may have the form $g(x, y, p) = 0$ to be solved to give $y = f(x, p)$ where the derivative $\partial y/\partial p$ is sought. The iteration should be ignored and the derivative obtained by differentiating the underlying system g followed by solving a linear system. Fuller descriptions for backward and forward methods respectively are given by Christianson (1998), Bartholomew-Biggs (1998).

Chapter 4

Implementation

4.1 Data structures

The backward method requires that a representation of the calculation be stored. We have chosen a data structure that resides in the module and represents the computational graph (see Subsection 4.1.3 for more details). This is built by a sequence of unary and binary overloaded operators. Each must be able to find the nodes of its operand or operands. We have chosen to do this with an integer field in the derived type. The integer field also permits us to locate the node from which the backward method is to be applied when first or second derivatives are wanted.

For the forward method, we could have held the data directly in pointer components of the derived type. We have chosen not to do so for consistency with the backward method and because the extra allocations would be likely to slow the computation and would leave us with little control over the management of storage. We again use an integer to locate the data.

This data structure means that, for both methods, we can discard the module data and reuse the storage whenever a computation is complete. The user indicates this by calling `ad01_initialize`. All the `ad01_real` variables now need to be regarded as undefined. We decided that it was essential to have a mechanism to label variables as undefined, both for safety (in case the user inadvertently uses such a value in an expression) and because in the forward method we need to distinguish between a first assignment when its data is established and a subsequent assignment when its data is overwritten. We therefore hold a second integer in the derived type to indicate the case. If this differs from the case in hand, we regard the variable as undefined. On each call of `ad01_initialize`, the case value is incremented by one, which immediately renders all the existing variables with their smaller case values undefined.

When the data of a subsidiary calculation (see Section 3.3) is restored, the case value is also restored so that its variables are restored to validity. Thus the mechanism also prevents the accidental mixing of data from two problems.

Another use for the case value is to distingish between anonymous *temporary* variables and *permanent* variables. We negate the case value if the variable is temporary. For example, when the statement

```
a = abs(b + c)
```

is executed, a temporary variable is created by the procedure for `+`. In the forward method, execution of the procedure for `abs` then creates a new temporary variable by changing the data for the old temporary variable. In the backward method, the node created for the temporary variable is reused by modifying the values of the local derivatives, see Section 4.2. In the forward method, we are sometimes left with discarded temporary variables that are no longer needed (for example, if a binary operation is applied to two temporary variables). Such discarded data is marked as *dead*, which means that the storage can be reused.

The result of these considerations is that the type `ad01_real` has two integer components, which we have named `p` and `case`. The components are private and accessible only within the module.

4.1.1 Data structures for the forward method

For the forward method, the code DAPRE of Stephens and Pryce (1991) stores all the derivatives for each variable in *sparse mode*. That is, an *index list* of the indices of the independent variables upon which the variable depends is stored, and only derivatives with respect to these variables are held. This avoids the calculation and storage of derivatives that are known to be zero. On the other hand, it requires additional memory to store the index lists and complicates binary operations, which have to merge these lists and expand the data of both operands to fit the merged list.

An alternative is to use the *full mode*, where all derivatives are held explicitly and no index list is stored. The sparse mode is ideal for one of the original (independent) variables, since its only dependency is upon itself. As dependent variables are calculated, their index lists grow steadily longer. Therefore we decided to use both modes. The sparse mode is employed if the index list is shorter than a threshold t. By default, t takes the smallest value for which the full mode doubles the required storage (which depends on the order of derivatives being found), unless $n \leq 5$ in which case the default value of t is 0 (forcing the full mode to be used all the time). The user is also permitted to specify t directly in the `ad01_initialize` call. For example, the choice $t = 0$ forces the full mode to be used all the time and the choice $t \geq n$ forces the sparse mode to be used all the time. Note that the sparse mode ensures that each variable's index list specifies exactly those independent variables upon which the variable depends.

To permit the reuse of storage as temporary variables are no longer needed and to accommodate the varying lengths of the index lists and the associated sets of derivatives, we manage two heaps. The integers are held in the array `iheap` and the reals are held in the array `rheap`. Associated with each variable is the following fixed-length data:

`nvars`	The length of the index list.
`ipont`	The starting position of the index list in `iheap`.
`rleng`	The size of the real data.
`rptr`	The starting position of the real data in `rheap`.

Four integer pointer arrays are used to hold this data, and the entries are accessed through the integer component `p` of a variable of type `ad01_real`. The arrays are initially allocated to have targets of size 1000 (held as the constant `heap_size`). If necessary, the size of each is doubled by allocating a temporary array of twice the current size, copying the data into it, deallocating the old array, and making the new array be the pointer target.

The first n locations in the array `iheap` contain the numbers 1 to n to provide an index list for any variable using full mode storage.

The function and derivative information for each variable is stored as explained in Subsection 2.1.2.

4.1.2 Garbage collection in the forward method

The indices of dead variables are linked in a chain (using `ipont`) so that when a new index is needed, it is taken from the head of the chain whenever the chain is non-empty. Thus no garbage collection is needed for the arrays `nvars`, `ipont`, `rleng`, and `rptr`.

Garbage collection is needed for the heap arrays `rheap` and `iheap`. First, a list of the indices of the active variables in the order of their data in `rheap` is formed (the data is in the same order in `iheap`). Then the data for each variable is moved forward in the arrays `iheap` and `rheap` one at a time in the order of this list, altering the corresponding indices in `ipont` and `rptr` at the same time.

To avoid frequent garbage collections, we check the free space after each collection. If more than 10% of `iheap` is in use, we increase its size to the greater of double its present size and five times the size in use. We do the same for `rheap`.

4.1.3 Data structures for the backward method

In the backward method when only first derivatives are required, the following information is needed at a node:

`rval`	The value of the variable x_j .
`onum`	The number of operands.
`iptrs`	The indices k of the operands.
`rvals`	The local derivatives $\frac{\partial f_j}{\partial x_k}$.

Since most nodes are binary, we have decided to hold this data in the form of fixed-length components of a derived type `node`. Storage for three integers and three reals is required. The whole graph is held as a pointer array `graph` of type `node`. This is initially allocated to have size 10,000 (held as the constant `init_max_nodes`). If necessary, its size is doubled by allocating a temporary array of twice the current size, copying the data into it, deallocating the old array, and making the new array be the pointer target.

If second derivatives are required, we also need to store the local second derivatives $\frac{\partial^2 f_j}{\partial x_k \partial x_i}$. At a unary node, we use the second component of `rvals`, since this is otherwise unused. At a binary node, we again use a fixed-length component of a derived type. The type is `second_deriv` and we hold a pointer array `der2` of this type, keeping its size the same as that of the array `graph`. The type has only one component, `der`, which is an array of size 3.

4.2 Backward method evaluation and graph construction

The computational graph is constructed by the execution of the code. Every operation involving an `ad01_real` variable results in either a new node being created in the graph or an old node being altered.

Binary operations with one `ad01_real` and one `real` or `integer` variable are represented as unary nodes. An assignment of a `real` or `integer` to an `ad01_real` gives rise to a node with no operands, which we call a *nonary* node. The assignment of an `ad01_real` variable to an `ad01_real` variable does not require a new node.

When a unary operation is applied to a temporary variable, a new node is not created, but the derivatives are modified. The method is similar to that used by Griewank and Reese (1991) and called *node-condensing.* Let the old node be l and the new node be j. If node l is unary, the combined function is

$$F_j(x_i) \quad = \quad f_j(f_l(x_i)), \tag{4.1}$$

which has derivatives

$$\begin{aligned} \frac{\partial F_j}{\partial x_i} &= f_j' \frac{\partial f_l}{\partial x_i}, & (4.2) \\ \frac{\partial^2 F_j}{\partial x_i^2} &= f_j'' \left(\frac{\partial f_l}{\partial x_i}\right)^2 + f_j' \frac{\partial^2 f_l}{\partial x_i^2}. & (4.3) \end{aligned}$$

If node l is binary, the combined function is

$$F_j(x_i, x_k) = f_j(f_l(x_i, x_k)). \qquad (4.4)$$

The equations (4.2) and (4.3) still apply, and there are similar equations for the derivatives with respect to x_k. For the cross derivative, we have

$$\frac{\partial^2 F_j}{\partial x_i \partial x_k} = f_j'' \frac{\partial f_l}{\partial x_i} \frac{\partial f_l}{\partial x_k} + f_j' \frac{\partial^2 f_l}{\partial x_i \partial x_k}. \qquad (4.5)$$

For functions such as `3.0*x`, `abs(x)` and `x/2.0`, the second derivative f'' is always zero. For functions such as `x + 2.0` and `x - 2.0`, the first derivative f' is always unity. We treat such functions as special cases, avoiding the unnecessary computations of the general case.

4.3 High derivatives

We now consider the evaluation of high derivatives in the forward method. These are held as Taylor coefficients, for which the associated notation was explained in Subsection 2.1.2.

4.3.1 Addition, subtraction, and multiplication

Given the Taylor coefficients of degree up to R for two variables a and b, the Taylor expansion for $a + b$ is

$$\sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)} a \, \mathbf{h}^{\mathbf{i}(i)} + \sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)} b \, \mathbf{h}^{\mathbf{i}(i)}. \qquad (4.6)$$

Here $\mathbf{i}(i)$, $i = 1, \ldots, \mathcal{R}$ is the sequence of multi-indices, see (2.8), (2.9) and (2.11). Each Taylor coefficient is found by adding the corresponding coefficients for a and b. Similarly, the Taylor coefficients for the difference $a - b$ are found by subtraction.

The Taylor coefficients for the product $a \times b$, may be obtained by performing the multiplication

$$\left(\sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)} a \, \mathbf{h}^{\mathbf{i}(i)}\right) \left(\sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)} b \, \mathbf{h}^{\mathbf{i}(i)}\right). \qquad (4.7)$$

and collecting monomial terms, discarding any of degree greater than R. An outer loop runs over the coefficients of a, and an inner loop runs over the coefficients of b. The corresponding multi-indices must be merged. For example,

$$(1,3,4)+(3,5)=(1,3,3,4,5) \tag{4.8}$$

and corresponds to

$$h_1h_3h_4 \times h_3h_5 = h_1h_3^2h_4h_5. \tag{4.9}$$

Since the degrees of the multi-indices in the sequence $\mathbf{i}(i)$, $i=1,\ldots,\mathcal{R}$ are monotonic, we may terminate the inner loop as soon as the degree exceeds R. Since we expect the cases $R=1$ and $R=2$ to be wanted far more often than $R>2$, we have coded these specially, avoiding a nested loop.

4.3.2 Division

For division, $d=a/b$, we need to solve the equation

$$\left(\sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)}b\,\mathbf{h}^{\mathbf{i}(i)}\right)\left(\sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)}d\,\mathbf{h}^{\mathbf{i}(i)}\right)=\sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)}a\,\mathbf{h}^{\mathbf{i}(i)}. \tag{4.10}$$

Separating out the first term of the first sum and noting that $T_{\mathbf{i}(1)}$ is the identity, we find the equation

$$b\sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)}d\,\mathbf{h}^{\mathbf{i}(i)}=\sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)}a\,\mathbf{h}^{\mathbf{i}(i)}-\left(\sum_{i=2}^{\mathcal{R}} T_{\mathbf{i}(i)}b\,\mathbf{h}^{\mathbf{i}(i)}\right)\left(\sum_{i=1}^{\mathcal{R}} T_{\mathbf{i}(i)}d\,\mathbf{h}^{\mathbf{i}(i)}\right). \tag{4.11}$$

Multiplication by

$$\left(\sum_{i=2}^{\mathcal{R}} T_{\mathbf{i}(i)}b\,\mathbf{h}^{\mathbf{i}(i)}\right) \tag{4.12}$$

raises the degree since there is no constant term. Therefore, once the Taylor coefficients $T_{\mathbf{i}(i)}d$ for degree less than r have been found, equation (4.11) may be used to calculate those of degree r. These equations are multi-dimensional versions of those of Rall (1981). Again, for efficiency, we code the cases $R=1$ and $R=2$ specially.

4.3.3 Other binary operations

The exponentiation `a**b` may be implemented as `exp(log(a)*b)`. The function `atan2(a,b)` differs by a constant from either `atan(b/a)` or `-atan(a/b)`. We therefore copy the derivatives of `atan(b/a)` if `abs(b) < abs(a)` or of `-atan(a/b)` otherwise, and use the Fortran intrinsic itself for the value. The functions `max`, `min`, and `sign` involve straightforward copying of the derivatives of one of the arguments, perhaps with a change of sign, apart from exceptional cases such as $x=y$ in `max(x,y)`. These lead to warnings and are discussed in Section 4.4.

4.3.4 Unary operations

Given the Taylor coefficients of degree up to R for a variable a, we wish to calculate the Taylor coefficients for $v = u(a)$, when the value and local Taylor coefficients

$$u_r(a) = \frac{1}{r!}\frac{d^r u}{da^r}, \quad r = 0, 1, \ldots, R \tag{4.13}$$

are known.

The Taylor expansion of $u(a+\delta)$ is

$$u(a+\delta) = u(a) + u_1(a)\delta + u_2(a)\delta^2 + \ldots + u_R(a)\delta^R. \tag{4.14}$$

Into this we need to substitute the Taylor expansion of equation (2.14):

$$\delta = \sum_{i=2}^{\mathcal{R}} T_{\mathbf{i}(i)}a\,\mathbf{h}^{\mathbf{i}(i)}. \tag{4.15}$$

As for the case of multiplication in Subsection 4.3.1, we collect monomial terms, discarding any of degree greater than R.

The present implementation, which was originally in Fortran 77, is a recursion, with the stack being managed 'by hand' rather than by the system. Each recursion adds in a term t involving $\frac{d^r u}{da^r}$, then calls itself for each term involving $\frac{d^{r+1}u}{da^{r+1}}$ with a multi-index that includes that of t. Informal Fortran 90 coding for this is shown in Figure 4.1.

The Taylor coefficients $T_{\mathbf{i}(i)}v$ of the result, $i = 2, 3, \ldots, \mathcal{R}$, are initialized to zero and calculated by

```
call calc (0, (), 2, 1, 0)
```

Note that this works with the derivatives $\frac{d^r u}{da^r}$, rather that the Taylor coefficients. The terms added in are therefore $r!\,\mathit{fact}\,u_r(a)$. Where all the multi-indices differ, this is because we add in only one representative of $r!$ identical terms. Where there are repetitions among the multi-indices, *fact* will have been suitably scaled during the recursive calls.

In fact, the algorithm we have just described is unnecessarily complicated and a straightforward alternative exists. Namely, we may evaluate the Taylor expansion of equation (4.14) by *nested multiplication* up to the desired order R as

$$u(x+\delta) \;:=\; u_R(x) \tag{4.16}$$
$$u(x+\delta) \;:=\; u(x+\delta)\times\delta + u_{R-r}(x), \; r = 1, 2, \ldots, R. \tag{4.17}$$

The calculation can be made more efficient by noting that, since δ has zero constant term, each multiplication by δ raises the least degree of a multinomial by 1; thus, one may discard terms of order $> r$ from the assignment

Figure 4.1: Pseudo-code for a unary operation

```
recursive subroutine calc($r, \mathbf{m}, l, fact, no\_same$)
    ! $r$ is the recursion depth.
    ! $\mathbf{m}$ is the multi-index $\mathbf{i}(i_1)+\mathbf{i}(i_2)+\ldots\mathbf{i}(i_r)$, with $i_1 \le i_2 \le \ldots i_r$,
    !   where + for multi-indices refers to merging, see equation (4.8),
    !   and $i_k$ is the do index at level $k$ of the recursion, $k = 1, 2, \ldots, r$.
    ! $l$ holds $i_r$
    ! $fact$ holds the factor for the next term
    ! $no\_same$ is the number of repetitions of $i_r$ in $i_r, i_{r-1}, \ldots$
    $T_{\mathbf{m}}v = T_{\mathbf{m}}v + fact \times \frac{d^r u}{da^r}$
    $same = no\_same + 1$
    do $i = l, \mathcal{R}$
        if (degree($\mathbf{m}$)+degree ($\mathbf{i}(i)$) $> R$) exit
        call calc($r+1, \mathbf{m}+\mathbf{i}(i), i, fact \times T_{\mathbf{i}(i)}a/same, same$)
        $same = 1$
    end do
end subroutine calc
```

(4.17). The multiplications and additions may be performed as explained in Subsection 4.3.1.

Preliminary tests of this method showed good performance and it will probably replace the present method in the next version of the package.

4.4 Exceptions

An exception is an event that makes it either impossible to continue the calculation, or likely that invalid results will be obtained. We have decided to distinguish between two kinds of exceptions. A serious exception, such as insufficient storage (failure of an `allocate` statement), is treated as an error. For a less serious exception, we give a warning. Our main criterion for choosing the category is whether it is possible to continue the calculation. Our errors are all organizational: invalid call of `ad01_initialize` or `ad01_restore`, procedure call ahead of first call of `ad01_initialize`, and insufficient storage.

We provide the user with control over whether execution always continues, stops after an error, or stops after either an error or a warning. The default is to stop only after an error. We also provide control over whether a message is printed for each error or warning, only for an error, or not at all. The default is to print a message only for an error.

Execution following an error is needed for robust software that recovers from an exception. For example, the software may use an alternative algorithm that is slower. Since the module data cannot be as intended, the module is placed in a 'motoring' mode, where most procedure calls do nothing. This continues until the module is re-initialized by an invocation of `ad01_initialize` or a previous calculation is resumed by an invocation of `ad01_restore` (see Section 3.3).

It would slow the execution too much to check for the ordinary floating-point exceptions such as overflow and divide by zero, so this is left as the user's responsibility. Our assumption is that we do not need to diagnose situations, such as `sqrt(x)` for negative `x`, that would be invalid in the original Fortran code. Also, we do not attempt to diagnose problems whose root cause is inadequate exponent range. Thus the user who is writing robust software must check for the occurrence of floating-point exceptions during use of the AD01 package.

This philosophy led us to diagnose `x**y` for `ad01_real` variables as an error when `x` is not positive, even if the value of `y` is a positive integer. This is because any perturbation of `y` gives a result that is not real. Similarly, `sqrt(x)` and `x**s` for `s` real with a non-integer value is diagnosed when `x` is zero because any negative perturbation of `x` gives a result that is not real.

Sometimes the function is not continuous but both one-sided derivatives are valid. This happens for `int(x)` and `aint(x)` when `x` is a `ad01_real` variable with an integer value. Since both one-sided derivatives of `aint(x)` are zero, we treat this as if it had a zero derivative. Similar considerations apply to `nint(x)` and `anint(x)` when the value differs from an integer by $\frac{1}{2}$.

When `x` has the value zero, the function `abs(x)` is continuous but has different left and right derivatives. Here, we issue a warning and take the derivative to be a random number between the two derivatives. Such a random choice is required for non-differentiable optimization calculations when bundle methods, see Lemaréchal and Zowe (1994), are in use. Similar considerations apply to `sign(x,y)` when either `x` or `y` is a `ad01_real` variable with the value zero, and to `max(x,y)` and `min(x,y)` when either a `x` or `y` is an `ad01_real` variable and `x` and `y` have the same value.

We provide a warning for any comparison (with a logical operator) between an `ad01_real` variable `x` and an object with the same value. This is because any perturbation of `x` (or any perturbation of a particular sign) may cause a different control flow through the program and hence discontinuities. Of course, here there is no mechanism for taking a random combination.

If any procedure is given an input `ad01_real` variable with an undefined value, we issue a warning and return an undefined result. Finally, a warning is issued if a call of a subroutine that extracts function or derivative values from an `ad01_real` variable is erroneous, for example, if higher-order derivatives are requested than are being calculated.

Chapter 5

Testing

5.1 Correctness testing

We describe the program written for testing the correctness of the forward package. That for the backward package is similar, the main difference being that it lacks the tests of the higher derivatives which are done only for the forward package. The aim has been to test each individual facility, as well as various extended calculations.

Each of the intrinsic unary functions f is put though a battery of tests, which includes

1. Compare f, f', f'' with directly computed values, with various values of the maximum degree R set by `ad01_initialize`.

2. Check correct operation when the input is an array of zero size.

All these are done for both sparse and full storage. The unary functions tested, include intrinsic binary functions where one argument is a constant. A similar battery is applied to each of the arithmetic operations `+`, `-`, `*`, `/`, `**`, as well as intrinsic binary functions `max`, `min`, etc, and the relational binary operations `==`, `/=`, and so on.

The following general tests are done:

- Make each of the 7 error conditions and 26 warning conditions occur.

- Test assignment between `ad01_real` and constant values of types `double`, `real` and `integer`.

- Test a unary function defined using `ad01_user`.

- Test the node-condensing feature, Section 4.2.

- Test the `ad01_store` and `ad01_restore` facility for interrupting and resuming a calculation.

- Test the ability to allocate more memory, and to garbage collect, in larger computations.

More extended calculations include computing Rosenbrock's function

$$f = \sum_{i=1}^{n-1} \left\{ (10(x_{i+1} - x_i^2))^2 + (x_i - 1)^2 \right\} \tag{5.1}$$

for $n = 500$ and checking its gradient against an independent calculation.

5.2 Performance testing

Our performance results are from execution in double precision arithmetic on a 143 MHz Sun Ultra 1 using the Fujitsu Fortran Compiler Version 4.0 of Dec 25 1997 with default optimization.

The MINPACK-2 test problem collection Averick, Carter, Moré and Xue (1992) contains Fortran 77 codes for calculating functions, gradients, and Hessians. We have used several of these to verify that AD01 is working correctly and to compare its performance with hand-coded differentation. For AD01, we removed all the hand-coded differentation and made the appropriate changes of type.

5.2.1 Rosenbrock's function

For our first performance test, we consider computing the value of Rosenbrock's function (5.1) and its gradient. We use the subroutine

```
subroutine dfun(x,f)
   real(kind(0d0))  x(:),f
   real(kind(0d0))  s1,s2
   integer i
   f = 0.0d0
   do i = 1, n-1
     s1 = 10.0d0*(x(i+1)-x(i)**2)
     s2 = 1.0d0 - x(i)
     f = f + s1**2 + s2**2
   end do
end subroutine dfun
```

to calculate f itself. A similar subroutine uses AD01 and has the same executable statements. A third routine has added hand-coded statements for calculating the gradient analytically. We show in Table 5.1 timings for several values of n.

It may be seen that the forward method is particlarly slow in this case. When the second addition of the expression

```
f + s1**2 + s2**2
```

is performed, a temporary variable is set up. When the assignment is performed, the old `f` is discarded and the temporary variable is used for the new `f`. Because separate procedures are used for the addition and assignment, the code has no way of knowing that is should have simply updated `f`. Since `f` has progressively more nonzero derivatives as the code progresses, the copying overheads become severe. We have considered adding new operators such as `.eqplus.`, so that the statement becomes

```
f .eqplus. s1**2 + s2**2
```

but there would need to be several such operators. Great generality is available with a single subroutine that tells AD01 that the variable's value is only going to be used once, so it can be given the same status as a temporary variable used in expression evaluation. The code becomes

```
call ad01_temp(f)
f = f + s1**2 + s2**2
```

and the addition function will not create a new temporary. Note that both syntaxes involve a change to the source code. We show the result of this modification in Table 5.1.

It may be seen that

- the use of `ad01_temp` speeds up the forward mode dramatically on this problem,
- the backward method is superior to the forward method, even if `ad01_temp` is in use,
- the time to calculate the function and derivative by the backward method is proportional to the time to calculate the function alone, and
- automatic differentiation using the backward method is about 33 times slower than analytic differentiation; most of the time (more than 80%) is spent constructing the graph.

5.2.2 MINPACK `ept` - elastic-plastic torsion

Our next test is `ept` from the MINPACK-2 test problem collection. It is an elastic-plastic torsion problem and involves a rectangular grid in which each rectangle is divided into two triangles by a NW to SE diagonal. On each triangle, a non-linear element function is defined, and this depends on the

Table 5.1: Times in μsecs for Rosenbrock's function

n	$t1$	$t2$	$\frac{t3}{t1}$	$\frac{t3}{t2}$	$\frac{t4}{t2}$	$\frac{t5}{t1}$	$\frac{t5}{t2}$	$\frac{t6}{t2}$
100	13	39	2533.	858.	209.	97.	33.	27.
200	19	71	4903.	1343.	233.	123.	35.	28.
400	34	150	8960.	2030.	218.	141.	33.	27.
800	63	326	17201.	3320.	212.	159.	33.	27.
1600	118	609	35308.	6851.	204.	165.	32.	26.

$t1$: function only using reals
$t2$: function and derivative using reals
$t3$: function and derivative using forward AD01
$t4$: function and derivative using `ad01_temp`
$t5$: function and derivative using backward AD01
$t6$: function using backward AD01

grid values at the three vertices. The objective function f is a weighted sum of all these element functions.

In the test results shown in Table 5.2, the grid is n by n. For large n the work of one f evaluation is proportional to n^2. This begins to be visible in the column labelled $t1$. The other columns show ratios which compare the times of handcoded function and first and second derivative with those of AD01. It is notable that the `adtemp` amendment, mentioned above, speeds up the forward mode so much that it beats the backwards mode on some values of n.

Table 5.2: Times in μsecs for MINPACK `ept`

		Reals		Backward			Forward			`ad01_temp`		
n	$t1$	$\frac{t2}{t1}$	$\frac{t3}{t1}$	$\frac{t4}{t1}$	$\frac{t5}{t1}$	$\frac{t6}{t3}$	$\frac{t4}{t1}$	$\frac{t5}{t1}$	$\frac{t6}{t3}$	$\frac{t4}{t1}$	$\frac{t5}{t1}$	$\frac{t6}{t3}$
2	49	1.15	2.	6.	8.	14.	25.	32.	17.	17.	19.	10.
4	67	1.34	7.	11.	14.	17.	53.	329.	54.	32.	183.	31.
8	127	1.65	35.	19.	23.	18.	84.	748.	42.	54.	339.	14.
16	347	1.92	252.	25.	31.	16.	112.	1961.	69.	71.	506.	9.
32	1210	2.14	1868.	29.	36.	11.						

$t1$: time for function evaluation, handcoded
$t2$: time for function and gradient, handcoded
$t3$: time for function, gradient and hessian, handcoded
$t4$: time for function evaluation, AD01
$t5$: time for function and gradient, AD01
$t6$: time for function, gradient and hessian, AD01

5.2.3 CFD calculation - steady supersonic 3D flow

The next performance test comes from a CFD calculation by Shaun Forth at RMCS. The model is of steady supersonic 3D flow in a fairly arbitrary region using a finite volume discretization, and the solution is computed by space-marching in the downwind direction.

There are five variables describing the flow on the face of each volume element, from which the flux can be calculated. Thus at each space-step, a system $F(X) = 0$ of $5N$ nonlinear equations in $5N$ variables has to be solved where N is the number of element faces on the current cross-section across the flow.

The function F is assembled from N evaluations of a local function f — a 'CFD flux function' due to Roe (1981) — which relates the flux of conserved quantities (mass, energy, momentum) at a cell face to the flow variables either side of the face. The Jacobian of F is assembled in a similar way from Jacobians of f.

The code for f is a routine ROE_FLUX of about 200 lines with no loops and a few branches. Values of f and its Jacobian are required millions of times in a typical run and account for a large proportion of the overall runtime of the flow solver. ROE_FLUX is typical of many such flux functions in use worldwide. Tests show that exact Jacobians give much faster convergence of the Newton iterations involved than do finite difference approximations to the Jacobian. Thus there is considerable interest in automatic generation of fast, exact Jacobian code for CFD kernels of this kind.

Table 5.3 gives timings for executing ROE_FLUX in various ways:

1. The raw code, with Jacobian J approximated by differencing.
2. ROE_FLUX converted to use the AD01 forward module.
3. ROE_FLUX converted to use the AD01 backward module.
4. ROE_FLUX converted to use a forward AD package specially written by Forth for this application, in which all arrays of derivatives are of fixed size.
5. ROE_FLUX with ADIFOR. For this, George Corliss in Milwaukee converted ROE_FLUX from Fortran 90 to Fortran 77 and passed it through the ADIFOR processor, which outputs a Fortran routine with added code to compute derivatives. The resulting code uses the Reverse method at statement level and the forward mode overall.

Considering the extra generality of AD01, its timings compare favourably with those of Forth's package, though ADIFOR comes out well on top, showing the benefit of source-text translation approaches to AD.

Method	Time (μsecs)
Raw Code + finite diffs	84
AD01 forward Code	972
AD01 backward Code	679
Forth's AD Code	456
ADIFOR Code	102

Table 5.3: Timings (average over 1000 repetitions in each case) for ROE_FLUX computing function and Jacobian by various methods. Fujitsu 4.0 compiler on a Sun Ultra Sparc 143MHz.

Chapter 6

Conclusions

AD01 has proved robust and reliable on the applications on which we have tested it. Its speed is somewhat disappointing compared with the theoretical estimates of computational complexity, but from our tests, the ratios of (time to evaluate differentiated code):(time to evaluate original code) appear comparable with those of other packages based on operator overloading such as ADOL-C.

Though aimed originally at Fortran 77 programs, AD01 can differentiate Fortran 90 code provided it does not use Fortran 90 array features. There is no problem with architectural aspects such as modules and internal procedures, nor with new control flow features, nor with storage management features such as `allocatable`, pointers etc. We believe it is currently the only AD package for general Fortran use which is so general, as well as offering both forward (with derivatives to arbitrary order) and reverse methods.

Developments planned are aimed primarily at large problems. Among these are: whole array assignments (see the discussion in Section 3.1); providing some other array features, which should greatly reduce the memory requirement for the reverse method; the better algorithm described in Subsection 4.3.4 for unary operations in the forward method; speed improvements by use of sparse-matrix techniques; support for Taylor series generation for differential systems; and a Fortran 95 version that takes advantage of its better data initialization features.

We acknowledge with thanks the work of our CASE PhD student David Cowey, who completed the initial implementation of AD01, evaluated the merits of various alternative data structures, and wrote the suite of correctness tests. We also thank EPSRC for its support of David Cowey under grant 9450007X and we thank George Corliss, Shaun Forth, Nick Gould, and Andreas Griewank for discussions that have led to improvments to AD01 and this report.

Bibliography

B.M. Averick, R.G. Carter, J.J. Moré, and G.L. Xue. The MINPACK-2 test problem collection. Technical Report MCS-P153-0892, Argonne National Laboratory, Mathematics and Computer Science Division, June 1992.

M.C. Bartholomew-Biggs. Using forward accumulation for automatic differentiation of implicitly defined functions. *Computational optimization and applications*, **9**, 65–84, 1998.

Martin Berz. COSY INFINITY version 4 reference manual. Technical Report MSUCL – 771, National Superconducting Cyclotron Lab., Michigan State University, East Lansing, Mich., 1991*a*.

Martin Berz. Forward algorithms for high orders and many variables with application to beam physics. *In* A. Griewank and G. F. Corliss, eds, 'Automatic Differentiation of Algorithms: Theory, Implementation, and Application', pp. 147–156. SIAM, Philadelphia, Penn., 1991*b*.

Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR – Generating derivative codes from Fortran programs. *Scientific Programming*, **1**(1), 11–29, 1992.

Christian H. Bischof, George F. Corliss, and Andreas Griewank. Structured second- and higher-order derivatives through univariate Taylor series. *Optimization Methods and Software*, **2**, 211–232, 1993.

Bruce Christianson. Reverse accumulation and implicit functions. *Optimization Methods and Software*, **9**(2), 307–322, 1998.

D. Bruce Christianson. Automatic Hessians by reverse accumulation in Ada. *IMA J. on Numerical Analysis*, 1991. Presented at SIAM Workshop on Automatic Differentiation of Algorithms, Breckenridge, Colo., January 1991.

Paul H. Davis, John D. Pryce, and Bruce Stephens. Recent developments in automatic differentiation. *In* J. C. Mason and M. G. Cox, eds, 'Scientific Software Systems', pp. 153–165. Chapman and Hall, 11 New Fetter

Lane, London EC4P 4EE, 1990. Also appeared as Technical Report ACM–89–1, Royal Military College of Science at Shrivenham, Shrivenham, U.K.

Lawrence C. W. Dixon. Use of automatic differentiation for calculating Hessians and Newton steps. *In* A. Griewank and G. F. Corliss, eds, 'Automatic Differentiation of Algorithms: Theory, Implementation, and Application', pp. 114–125. SIAM, Philadelphia, Penn., 1991.

Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. *In* A. Griewank and G. F. Corliss, eds, 'Automatic Differentiation of Algorithms: Theory, Implementation, and Application', pp. 126–135. SIAM, Philadelphia, Penn., 1991.

Andreas Griewank and Jean Utke. Evaluating higher derivative tensors by forward propagation of univariate power series. Preprint, Technical University Dresden, 1996.

Andreas Griewank, David Juedes, and Jean Utke. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, **22**, 131–167, 1996.

Harwell Subroutine Library. *A catalogue of subroutines (release 12).* AEA Technology, Harwell Laboratory, Harwell, UK, 1996.

C. Lemaréchal and J. Zowe. A condensed introduction to bundle methods in nonsmooth optimizations. *In* E. Spedicato, ed., 'Algorithms for continuous optimization. The state of the art', pp. 357–382, Kluwer Academic Publishers, Dordrecht, 1994.

Richard D. Neidinger. An efficient method for the numerical evaluation of partial derivatives of arbitrary order. *ACM Trans. Math. Software*, **18**(2), 159–173, June 1992.

James M. Ortega and Werner C. Rheinboldt. *Iterative solution of nonlinear equations in several variables.* Academic Press, New York, 1970.

Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, Vol. 120 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1981.

P.L. Roe. Approximate Riemann solvers, parameter vectors and difference schemes. *J. Computational Physics*, **43**, 357–372, 1981.

Bruce R. Stephens and John D. Pryce. DAPRE: A differentiation arithmetic system for FORTRAN. Technical Report ACM–91–3, Royal Military College of Science, Shrivenham, U.K., 1991.