Automatic Differentiation of Algorithms

Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson and Laurence Dixon

Numerical Optimisation Centre, University of Hertfordshire

Hatfield, England, Europe

July 17, 2000

Abstract

We introduce the basic notions of Automatic Differentiation, describe some extensions which are of interest in the context of nonlinear optimization and give some illustrative examples.

Keywords: Adjoint Programming, Algorithm, Automatic Differentiation, Checkpoints, Error Analysis, Function Approximation, Implicit Equations, Interval Analysis, Non-linear Optimization, Optimal Control, Parallelism, Penalty Functions, Program Transformation, Variable Momentum.

1 Introduction

Automatic Differentiation (AD) is a set of techniques for transforming a program that calculates numerical values of a function, into a program which calculates numerical values for derivatives of that function with about the same accuracy and efficiency as the function values themselves.

The derivatives sought may be first order (the gradient of a target function, or the Jacobian of a set of constraints), higher order (Hessian times direction vector or a truncated Taylor series), or nested (calculating $\nabla_x\ F(x, f(x), f'(x))$ for given f and F).

Many non-linear optimization techniques exploit gradient and curvature information about the target and constraint functions being calculated. Derivatives also play a key role in sensitivity analysis (model validation), inverse problems (data assimilation) and simulation (design parameter choice).

These derivatives can be estimated using divided differences, but such estimates are prone to truncation error when the differencing intervals are numerically large, and to round-off error when they are small. In addition, the run-time requirements of a divided difference approach are often unacceptably high, particularly for problems with a large number (thousands) of independent variables.

The manual development of code for evaluating analytic derivatives of a function is a tedious and error-prone activity. Of course, symbol manipulation programs can differentiate individual equations, but the code for evaluating a function of interest typically has a non-trivial control flow, involving conditional statements, loops, and subroutine calls, as well as data structures which may be updated many times during the evaluation process. Particularly if the underlying program is subject to continual structural change, it is generally desirable to automate, at least in part, the process of transforming it into a program that calculates derivative values, and this was the initial motivation for the development of AD.

The basic process of AD is to take the text of a program (called the underlying program) which calculates a numerical value, and to transform it into the text of a program (called the transformed program) which calculates the desired derivative values. The transformed program carries out these derivative calculations by repeated use of the chain rule from elementary calculus, but applied to floating point numerical values rather than to symbolic expressions.

The transformation process may be carried out by a compiler-like tool, or by operator overloading. Tools using the latter approach are simpler to build, but produce code which is less efficient to run.

The compiler-like transformation of a pre-existing program is not the whole story of AD however. The efficient transformation of programs which include the solution of complicated sub-problems often benefits from user insight into the problem structure, and conversely the conceptual framework imposed by AD often gives users insight into more efficient ways of coding the underlying program. Consequently the term AD has stretched to cover the user-driven transformation of abstract algorithms, as well as the automatic transformation of concrete programs.

In this paper we give a rapid review of the basic techniques of AD, followed by a quick tour of a few extensions and examples with which we have been personally involved and which we consider interesting from the standpoint of non-linear optimization. This paper does not attempt to give a history of AD (see [20]), nor does it give a complete account of the foundations of AD (a full account from a mathematical point of view is given in the excellent book by Griewank [17]). Neither do we attempt to make a systematic survey of prior or current work in the field (such as that in the blue and green books [19], [3]), nor of the many tools which are available[1]. A great deal of work which we regard as central to the discipline is not mentioned at all in this paper, for reasons of space. Nevertheless, we hope to impart a flavour of AD, to give the reader some idea of what goes on inside an AD tool, and to develop an initial insight into the effect which certain lines of research in AD may eventually have upon what

[1]See for example `www.mcs.anl.gov/Projects/autodiff/AD_Tools`

such tools can accomplish for optimization.

The rest of this paper is organized as follows. In the first part of the paper we develop the two basic building blocks of AD, the forward and reverse accumulation modes. The forward mode is set out in the next section, and the reverse mode in section 4, following the introduction of the ancillary notion of a Wengert list in Section 3. The reverse mode can be implemented directly by overloading, but the more efficient program transformation approach requires the adjoint program construction techniques set out in Section 5. Section 5 also introduces the important concepts of checkpointing and preaccumulation.

The second part of the paper outlines some extensions of the basic techniques of AD. Section 6 introduces some of the issues raised for AD by function approximation techniques such as discretization and iterative solution of subproblems. The case of implicit equation solution is considered in more detail in Section 7. Section 8 deals with the use of the reverse mode to obtain automatic error estimates, and Section 9 considers the extension of AD to second and higher derivatives.

The final part of the paper begins in Section 10 with a discussion of the differences between the overloading and code translation approaches to AD implementation. This is followed by two examples, which are used to illustrate the earlier theory and to provide a concrete setting for some of the discussion: a discrete-time optimal control problem in Section 11 and a constrained nonlinear optimization with exact penalty function in Section 12. Some reflections upon the future impact of AD-related research are set out in the final Section.

2 Forward Accumulation

Suppose that we have an underlying program (or a subroutine) f, which takes n independent variables x_i as inputs, and produces m dependent variables y_i as outputs, and that we wish to obtain numerical values for the Jacobian $J = f' = [\partial y_i / \partial x_j]$ given particular values for the x_i.

The forward accumulation technique associates with each floating point program variable v a vector $\dot{v}$ of floating point derivative values. Conceptually the simplest, *Cartesian*, case is when each dot-vector $\dot{v}$ contains one component for each independent variable x_i and component i contains the corresponding derivative $\partial v / \partial x_i$ so that

$$\dot{v} = \nabla_x \, v.$$

More generally, the number r of vector components may differ from the number of independent variables, and component i may contain an arbitrary directional derivative, or tangent vector, of the form $p_i \cdot \nabla_x \, v$ corresponding to the tangent direction given by the $n-$vector p_i.

In the cartesian case, we initialize the dot-vector $\dot{x}_i$ corresponding to the independent variable x_i by setting $\dot{x}_i := e_i$, the i-th cartesian unit vector. We write this loosely as $[\dot{x}] := I_n$. More generally, we initialize $\dot{x}_i$ to the $i-$th row of the tangent direction bundle $P = [p]_{n\times r}$.

Each operation which assigns a value to a floating point variable must be augmented by an operation to assign correct floating point values to the corresponding dot-vector: for example the operation

$$v_3 := v_1 * \sin(v_2)$$

must be augmented by the assignment

$$\dot{v}_3 := v_1 * \cos(v_2) * \dot{v}_2 + \dot{v}_1 * \sin(v_2).$$

It is straightforward to see how to modify the underlying program so that it calculates the dot-vector values directly itself. We can use an operator-overloading approach, or we can systematically transform the source code. The source translation approach requires a greater initial investment in development, but has certain advantages from the viewpoint of efficiency, which we discuss further in §10 below.

In an overloading approach, the pair $(v, \dot{v})$ can be combined into a new user-defined data type called a *doublet*. Appropriate overloaded operations corresponding to the usual floating point operations can be defined to manipulate the dot-values in accordance with the chain rule. All active floating point program variables[2] can be re-declared to be of this doublet type. The derivative operations and storage management will automatically occur even though the text of the evaluation program is unchanged.

In a source-translation approach the lines of code which declare and manipulate storage space and values for active program variables v can be augmented by code to declare and manipulate storage space and values for $\dot{v}$ in tandem[3].

If suitable processors are available, the components of $\dot{v}$ can be calculated in parallel. If the structure of the problem is such that the $\dot{v}$ are sparse, then they can be implemented as sparse vectors. If the number of non-zero components is large, then it will usually be more efficient to evaluate them in batches[4], with the underlying function evaluation repeated for each batch.

[2]A program variable is *active* if it both depends upon an independent variable, and influences the value of a dependent variable, for some possible control flow of the program.

[3]It is prudent to place the assignment to $\dot{v}$ before that for v in the transformed code, since the variable v on the left-hand side of an assignment statement may also appear on the right, and the old value of v rather than the new is required to evaluate $\dot{v}$. In most modern computer languages parameter passing mechanisms, array index calculations, and pointer manipulation make it difficult to determine at compile time whether two variables are the same.

[4]The batch size is chosen so that the overhead of repeating the function evaluation, amortized over the size of the batch, just balances the thrashing caused by the growth of the working set with the batch size.

3 Wengert Lists

In order to describe the reverse accumulation technique, we need to untangle the relationship between a mathematical variable and a program variable. In this section we describe for this purpose an abstraction called a *Wengert list* [24]. We can think of a Wengert list as a trace of a particular run of a program, with specific values for the inputs. The only statements which occur in the Wengert list are assignment statements to non-overwritable variables called Wengert variables. The Wengert list abstracts away from all control-flow considerations: all loops are unrolled, all procedure calls are inlined, and all conditional statements are replaced by the taken branch. Consequently, the Wengert list may be many times longer than the text of the program to which it corresponds.

The Wengert list also abstracts away from all considerations of storage management. Each assignment statement in the Wengert list has a different variable on the left-hand side. Thus a single program variable may correspond to many different Wengert variables, one Wengert variable for each occasion upon which a value is assigned to the program variable. The Wengert list can be considered as a straight-line program for evaluating y from x without overwriting any variable after it has been initialized. Alternatively, a Wengert list can be viewed as an unordered set of mathematical equations expressing functional dependencies between Wengert variables and which could be differentiated symbolically[5]. The length of the Wengert list, and hence the number of Wengert variables for which storage is required, is proportional to the run-time of the underlying program.

In general, a Wengert list has the following form:

```
for $i$ from $1$ upto $n$ do
    $v_i := x_i$
enddo
for $i$ from $n+1$ upto $N$ do
    $v_i := f_i(v_{\tau_i 1}, \ldots, v_{\tau_i n_i})$
enddo
for $i$ from $N+1$ upto $N+m$ do
    $y_{i-N} := v_{i-m}$
enddo
$\{(y_1, \ldots, y_m) = f(x_1, \ldots, x_n)\}$
```

where for each $i > n$, n_i is the arity of f_i and τ_i is a map from $\{1, \ldots, n_i\}$ into $\{1, \ldots, i-1\}$.

In this formulation, we allow the functions f_i to be arbitrary differentiable scalar-valued functions. However we could, by introducing additional Wengert variables, ensure that the functions f_i were all of a certain simple form: for example we could allow only unary operations (operations on single arguments)

[5]The Wengert list can also be viewed as a linearization of the computational graph.

together with binary addition[6]. Alternatively, we could allow more general vector- or matrix-valued functions for f_i.

In what follows, we frequently write down derivative expressions such as $\partial y_j/\partial v_i$. This is a slight abuse of notation, since each intermediate variable v_i depends functionally upon the input variables x_i. Purists who wish to avoid any ambiguity about whether a variable is dependent or independent can replace the assignment $v_i := f_i(v_{\tau_i 1}, \ldots, v_{\tau_i n_i})$ by the identity $v_i = f_i(v_{\tau_i 1}, \ldots, v_{\tau_i n_i}) + u_i$ where the u_i are additional independent variables with value zero, and consider $\partial y_j/\partial u_i$ when we write $\partial y_j/\partial v_i$.

4 Reverse Accumulation

The reverse accumulation technique associates with each floating point program variable v a vector $\bar{v}$ of floating point derivative values[7].

Conceptually, the simplest case is when each of these bar-vectors contains one component for each dependent variable, and component i contains the corresponding derivative $\partial y_i/\partial v$, so that

$$\bar{v} = D_v \ y.$$

More generally, the number s of vector components may differ from the number of dependent variables, and component i may contain an arbitrary adjoint derivative, or co-tangent vector, of the form $q_i \cdot D_v \ y$. corresponding to the co-tangent direction given by the $m-$vector q_i.

Each operation which assigns a value to a floating point variable must be augmented by an operation to assign correct floating point values to the corresponding bar-vectors, according to the chain rule: for example the operation

$$v_3 := v_1 * \sin(v_2)$$

corresponds to the assignments

$$\bar{v}_1 := \bar{v}_3 * \sin(v_2); \qquad \bar{v}_2 := \bar{v}_3 * v_1 * \cos(v_2).$$

In contrast with the forward case, the bar-vectors $\bar{v}_i$ cannot be calculated in the same sequence as the variable values v_i, but must be evaluated in the opposite (or reverse) order.

In the simplest case, we initialize the bar-vector $\bar{y}_i$ corresponding to the dependent variable y_i by setting $\bar{y}_i := e_i^T$, the i-th cartesian unit vector. We write this loosely as $[\bar{y}] := I_m$. More generally, we initialize $\bar{y}_i$ to the $i-$th column of the co-tangent direction bundle $Q = [q]_{s \times m}$.

[6] Multiplication by a constant and squaring are unary operations, and binary multiplication can be defined by $a * b = 2^{-2} * [(a+b)^2 - (a-b)^2]$. To avoid cancellation error, the operands can be dynamically scaled by opposite powers of two, which cancel in the derivative formulae.

[7] Formally, $\dot{v}$ is a column vector and $\bar{v}$ is a row vector.

In this section we explain how to reverse accumulate the adjoint variables $\bar{v}$ for programs expressed in the form of a Wengert list. In §5 which follows, we extend these techniques to more general programs with variable assignment and control flow. Examination of the Wengert list yields the following algorithm for computing the adjoint variables.

```
for $i$ from 1 upto $n$ do
    $v_i := x_i$
    $\bar{v}_i := 0.0$
enddo
for $i$ from $n+1$ upto $N$ do
    $v_i := f_i(v_{\tau_i 1}, \ldots, v_{\tau_i n_i})$
    $\bar{v}_i := 0.0$
enddo
for $i$ from $N+1$ upto $N+m$ do
    $y_{i-N} := v_{i-m}$
    $\bar{v}_{i-m} := \bar{y}_{i-N}$
enddo
for $i$ from $N$ downto $n+1$ do
    for $j$ from 1 to $n_i$ do
        $\bar{v}_{\tau_i j} := \bar{v}_{\tau_i j} + \bar{v}_i * (D_j f_i)(v_{\tau_i 1}, \ldots, v_{\tau_i n_i})$
    enddo
enddo
for $i$ from $n$ downto 1 do
    $\bar{x}_i := \bar{v}_i$
enddo
$\{(\bar{x}_1^T, \ldots, \bar{x}_n^T) = Qf'(x_1, \ldots, x_n)\}$
```

The adjoint variables are incremented rather than simply assigned because although a Wengert variable can be written only once, it can be read several times. At each point at which it enters the subsequent calculation it can affect the dependent variables, and the relevant adjoint value is the sum of all such effects.

Of particular interest is the case $m = 1$ where there is only one dependent variable. Programs which calculate a single scalar-valued objective or *target* function arise in unconstrained problems or when constraints are incorporated using penalty or barrier functions. In this case the $\bar{v}_i$ are scalars, and it is clear that reverse mode AD allows the entire gradient vector to be extracted, to the same level of precision as the function, for about the cost of three function evaluations, *regardless of the number of independent variables.* This fact, which deserves to be more widely known than it appears to be, follows from the consideration that the computational cost of evaluating Df_i for elementary f_i is generally no greater than that of evaluating f_i itself.

In the case where overloading is used, it is a relatively simple matter to modify the underlying program so that it builds its own Wengert list of elementary floating point operations, with each overloaded operation appending the next list item to a data structure as a side-effect. The reverse pass over this list can then be invoked by calling a separate routine. The Jacobian values of $D_j f_i$ can be saved on a stack on the way forward, and used in reverse order on the way back. Alternatively, the values of the program variables can be saved whenever they are overwritten, and the restored values used to calculate the $D_j f_i$ on the way back.

The high storage requirement of such a naive approach to reverse mode AD is prohibitive for large problems. However, for many small to medium size problems the relatively cheap cost of secondary storage, the efficiency of virtual memory, and the fact that access to the Wengert list can be made essentially serial, means that the naive implementation approach is viable.

However it is also possible (and more efficient in both run-time and storage space, see §10 below) to implement the reverse method by transforming the underlying program into an adjoint program with the "opposite" control flow, and we consider how to do this in the next section. This transformation enables the more subtle analysis of the trade-offs between *storing* results that will be needed later and *recomputing* them, which is required by larger problems. The judicious use of recomputation usually allows reverse mode AD to be done with a storage requirement that is only a small factor larger than that required by the underlying program. Furthermore, the recomputations can generally be done in parallel in such a way that the overall runtime is not increased. We consider this issue further in the next section, and give an example in §11.

5 Adjoint Program Construction

In this section we sketch how to transform code so as to enable the calculation of adjoint values. We have no space here to describe the informatics involved, so we simply set out the transformation process as if it were being done by hand. The initial task of AD is to automate this process of program transformation, by the development of compiler-like tools and appropriate operating system interfaces. We assume that the underlying program has been augmented to save partial derivative values or overwritten variable values on the way forward, and consider the structure of the program, called the *adjoint program*, required to carry out the reverse pass.

Variables and assignment statements. The adjoint program declares and manipulates adjoint program variables, which may be vectors or scalars. Exactly one adjoint program variable $\bar{v}$ is required for each program variable v in the underlying program. This follows from the observation that, if two Wengert variables correspond to successive values of the same program variable on the way forward, then their adjoints can share the same storage on the way back: a program variable value which has been overwritten can no longer influence the dependent variable values and so has adjoint value zero, while a program variable value which has not yet been assigned corresponds to an adjoint value which will not be used again and so can be discarded.

Hence, to the program assignment statement: "$v_i := f_i(v_{\tau_i 1}, \ldots, v_{\tau_i n_i})$" corresponds the adjoint code:

```
$\bar{t} := \bar{v}_i$
$\bar{v}_i := 0.0$
for $j$ from 1 to $n_i$ do
    $\bar{v}_{\tau_i j} := \bar{v}_{\tau_i j} + \bar{t} * (D_j f_i)(v_{\tau_i 1}, \ldots, v_{\tau_i n_i})$
enddo
```

Here $\bar{t}$ is a "temporary" adjoint variable, introduced to allow for the fact that in the underlying program, in contrast to the Wengert list, the variable v_i on the left-hand side of an assignment statement may also appear on the right[8].

An alternative to saving partial derivative values on the way forward is to calculate them on the reverse pass[9]. This requires some of the overwritten values of program variables to be saved on the way forward so that they can be restored at the corresponding point on the reverse pass. Specifically, if the overwritten value appears as an argument to a non-affine function f_i then it must be saved. Sometimes we can avoid the need to store and restore floating point program variable values by inverting the calculation which produced them [22] but roundoff makes this difficult in general[10]. Alternatively, we can re-calculate the overwritten program variable values from checkpoints as described below in §5.5.

Sequence of statements. The statements in the adjoint program consist of the adjoints of the statements in the underlying program, but in reverse order, so that the adjoint of "$S_1; S_2; S_3$" is "$\bar{S}_3; \bar{S}_2; \bar{S}_1$" We have already seen how to adjoin assignment statements. We indicate below how to adjoin statements affecting control flow.

[8]It is prudent to do this in all cases since, as mentioned before, parameter passing mechanisms, array index calculations, and pointer manipulation make it difficult to determine at compile time whether two variables are the same.

[9]Whichever alternative is adopted, access to the archived values is serial and predictable, so high latency secondary storage can be used provided the burst bandwidth is sufficiently high.

[10]Although it is worth noting that if $w_{i+1} = f_i(w_i)$ for all i where w is the state vector, and if g_i is an approximate inverse of f_{i-1}, then the transformation $\tilde{f}_i$ defined by $w_{i+1} = f_i(w_i) + w_{i-1} - g_i(w_i)$ approximates f_i to the same degree and $\tilde{f}_{i-1}$ has exact inverse $\tilde{g}_i$ given by $w_{i-1} = g_i(w_i) + w_{i+1} - f_i(w_i)$.

Procedure and function calls The adjoint of a procedure call is a call to the adjoint procedure. The adjoint procedure $\bar{P}$ contains the adjoints of the statements in the underlying procedure P, in the reverse order. Out parameters become in parameters and in parameters become in-out. Functions can be treated as procedures with an additional out-parameter.

When there is a need to trade storage space against recomputation, procedure boundaries provide a natural point at which to do so. According to the orthodox view, in a well-designed program the number of times variables are updated across procedure-call boundaries, either as global variables or as parameters, is low relative to the number of program variable updates which occur within the procedure. This allows space to be saved using *pre-accumulation* or *checkpointing*.

Preaccumulation. Pre-accumulation involves treating the entire procedure as a (possibly vector-valued) elementary operation f_i and storing the partial derivative f_i' on the stack instead of storing partial derivative values for the complete set of internal operations. This Jacobian f_i' can be evaluated at the time when f_i is called, by a recursive application of forward or reverse mode AD to the procedure[11]. This leads to a substantial space saving when the space occupied by f_i' is small relative to the number of internal operations of the procedure f_i.

Although extra multiplications are required to incorporate f_i' on the outer reverse pass, the total operation count may be actually reduced, depending upon the number of procedure inputs and outputs n_i and m_i relative to n and m [9]. Similar considerations apply to the exploitation of structural sparsity. In a parallel processing environment, preaccumulation can shorten the elapsed time of a calculation even when $m = 1$, because the pre-accumulation of f_i' can be done in parallel and hence moved off the critical path of the calculation [5].

Checkpointing. A checkpoint is a complete record of the program state at a particular point of execution[12]. *Incremental* checkpointing across a procedure boundary is a matter of noting what changes are made to the environment by the procedure via parameters and global variables, in such a way that these changes can be quickly undone and reapplied (toggled) to a previously recorded

[11]From the linear algebra viewpoint, the Wengert list expresses the Jacobian as a product of large, sparse matrices, one for each f_i. Forward and reverse accumulation correspond to multiplying these sparse matrices from left to right, or from right to left. There is a huge body of recent interesting work on the optimal order in which to interlock forward and reverse accumulation steps to optimize the operation count, which we do not have space to touch on here. A good conceptual overview of the issues is given by Griewank and Reese [18]. See also [11].

[12]A checkpoint includes the program counter and a snapshot of the procedure calling history (runtime stack), as well as the program variable values.

checkpoint[13]. When a checkpoint has been taken at the entry point of a procedure call, then the complete internal record of variable values overwritten by the procedure can be discarded and the storage saved, since these values can now be recomputed from the checkpoint. According to orthodoxy, in a well-designed program an incremental checkpoint across a procedure call boundary should require only a small proportion of the space occupied by the entire program state. On a reverse pass, the adjoint procedure $\bar{P}$ begins by toggling the program state from the exit state to the entry state using the incremental checkpoint, then calls the augmented version of the underlying procedure P to re-create the internal record before proceeding with reverse accumulation. In the parallel processing case this re-creation process for P can be moved off the critical path by allowing it to be started sufficiently early to be ready when required. Finally, in the case of nested procedure calls, subroutines P_1, P_2, etc called by P need not be re-evaluated when P is re-evaluated: provided the incremental checkpoint for P_i is available, evaluation of P_i can be replaced by a state toggle from the entry to the exit state.

These two techniques of preaccumulation and checkpointing can be combined. For further details see [23].

Conditional statements and loops. The adjoint of the conditional statement "if c then S_1 else S_2 endif" is the statement "if c then $\bar{S}_1$ else $\bar{S}_2$ endif". If either of the statements $S1$ or $S2$ could affect the value of the condition c, then the value of c can be pushed on a stack on the way forward, and popped on the way back, just like the partial derivative values or overwritten program variable values.

The adjoint of a loop is also a loop. In case of a for loop, the adjoint is a for loop in reverse order. In case of a while loop, the adjoint loop performs the adjoint of the loop body the same number of times as on the way forward. We can can either determine a precondition to identify the first iteration of the forward loop, which is the last iteration of the backward loop [15, Chapter 21], or we can store the number of iterations that was actually performed, analogous to the if statement[14].

Where loop iterations are independent, they can be done in the same order as on the way forward and array subscripts can be calculated in the same way as on the way forward. Otherwise the array index calculations must be reversed: sometimes this is possible, since roundoff is not an issue, but in the worst case the index values have to be stored in sequence on the way forward and restored on the reverse pass, just like overwritten floating point variables.

[13]For example a "fork" can be used to take an incremental checkpoint if the operating system uses a lazy copy-on-write scheme for the virtual memory pages in the process runtime stack.

[14]Often the sequence of values to be stored exhibits a regular pattern, in which case standard data compression techniques such as Huffman encoding can be applied to reduce the space required.

Loop iterations also form good boundaries at which to consider checkpointing and preaccumulation. Loops which perform temporal evolution or some other form of in-place state space update (such as ODE evolution or Optimal Control) are particularly good candidates for checkpointing (for example see §11 below).

Loops to perform array operations can be regarded as single steps and replaced by the corresponding adjoint step. For example the matrix operation $X := Y * Z$ corresponds to the adjoint operations $\bar{Y} := \bar{Y} + Z * \bar{X}; \bar{Z} := \bar{Z} + \bar{X} * Y$, where we adopt the convention that adjoint matrix components are of transpose shape relative to the underlying matrix.

Loops which perform equation solving are of particular interest, since in general we do not need to record the process by which the solution was found (see §7 below.)

Input and output. For reads and writes to a sequential file, called say "foo", the adjoint operations are straightforward, and similar to those for variable assignment. The adjoint to "read (v, foo)" is "write ($\bar{v}$, foobar); $\bar{v} := 0.0$" and the adjoint to "write (v, foo)" is "read ($\bar{t}$, foobar); $\bar{v} := \bar{v} + \bar{t}$ ". For random access files the situation is a little more fraught, see [13] for a good account of what is involved.

In the parallel processing case similar considerations apply to Inter-Processor Communication: sends can be regarded as writes and receives as reads[15].

6 Approximating Differentiable Functions

A question which we often need to consider explicitly is "if we calculate an approximation f_n to a function f_*, when do we want the derivative of f_n and when do we need an approximation to the derivative of f_*?" This is an important question, because the fact that f_n approximates f_* does not imply that f'_n approximates f'_* to the same order, or even in the limit. This is particularly apparent when piecewise-defined functions are glued together using if-statements[16]. For example the code

```
if $x = 0.0$ then $y := 0$ else $y := (1 - \cos(x))/x$ endif
```

will give the derivative value $\partial y / \partial x = 0.0$ for $x = 0.0$ instead of the presumably intended value of 0.5. For forward or reverse mode AD to work correctly in this case the programmer could have written:

```
if $x = 0.0$ then $y := x/2$ else $y := (1 - \cos(x))/x$ endif
```

[15]Actually there is an interesting dualism between trying to find the optimal decomposition of a program into parallel parts to minimize runtime and IPC, and the optimal checkpointing schedule to minimize the overwrite stack and incremental checkpoint sizes.

[16]In contrast with differencing, an AD tool can produce a warning when an intermediate variable v is too close to a cut value, by looking at $\dot{v}$ in the light of the given tolerance for x or at $\bar{v}$ in the light of the required tolerance for y, cf §8.

A similar problem occurs when using a while loop: different numbers of iterations give different branches of a piecewise function. Differentiate an iterative approximation $v := \phi(x, v)$ and the derivatives $\dot{v}$ may not converge, or may lag behind the convergence of v. For example, suppose the starting value for v is exactly right: then the while loop is skipped and we have $\dot{v} = 0$. The situation with reverse accumulation is even more problematic if we take the naive approach of differentiating the approximation function which we coded without having considered at the time when we coded it the requirement that it also approximate the derivative [14].

Clearly the derivatives must be incorporated into the stopping criterion in some way. A lot is now known about how to do this, but in many cases it is better to construct an iterative approximation to the derivative of the function to which the underlying iterative approximation is converging, rather than to differentiate the underlying approximation function directly. We consider this issue further in the next section, but point out that methods suitable for an interval-valued approach appear to have some potential to reconcile these two agendas[17].

Another source of inaccuracy is introduced by discretizing a continuous problem. In this case, it is usually best to differentiate the discretization used, since verification of descent criteria (eg Wolfe conditions) and the introduction of devices to enforce global convergence of Newton-like methods should be applied to the numerical values actually being calculated[18]. However this policy requires the discretization to be suitable for derivatives as well as for function values, which is a non-trivial additional constraint upon the modelling process.

7 Iteration and Equation Solving

Many computations $y := f(x)$ include as a subproblem the solution of implicit equations, of the form $\psi(u, v) = 0$ where u, v are $p-$ and $q-$vectors of knowns and unknowns respectively and ψ is a well-behaved $q-$vector valued map. In the linear case, these subproblems take the form of solving $Av = b$ for v where A and b are functions of x.

The underlying program contains code for solving these implicit equations, and it would be possible to treat this solver code as a black-box, and to apply AD to it mechanistically. In some cases, as we saw in the previous section, this will not produce the derivative values which are required: in other cases it will produce correct, but very inefficient, derivative code. It is usually advantageous for the AD translation process to identify explicitly the equations being

[17]Consider the properties of an algorithm which produces a joint enclosure for the true and approximation function values.

[18]Convergence under dynamic refinement of the discretization typically relies upon an unstated compactness result. Again, consideration of enclosure properties suggests that interval methods have some potential here in the context of AD.

solved, and to provide or invoke a solution code for the corresponding derivative equations which exploits shared values between the two equation solutions.

For example, if $Av = b$ then $A\dot{v} = \dot{b} - \dot{A}v$ for each tangent direction. Similarly, the adjoint operations corresponding to solving $Av = b$ for v are $\bar{b} := \bar{b} + z; \bar{A} := \bar{A} - vz$ where z is the solution to $zA = \bar{v}$ for the corresponding co-tangent direction[19]. If the underlying program forms an LU-decomposition of A in order to solve the original equations, then this can be exploited to obtain $\dot{v}$ from v, or $\bar{A}, \bar{b}$ from $\bar{v}$, at a much lower cost than simply applying AD to the equation solver: typically the operation count becomes of order q^2 rather than q^3, which in many cases means that the derivatives effectively become free [10].

For the non-linear case of solving $\psi(u, v) = 0$ for v, an iterative scheme $v := \phi(u, v)$ will generally be used. Now $\dot{v}$ must satisfy [1] the linear equations $\psi'_v \dot{v} = -\psi'_u \dot{u}$. Similarly [7] the adjoint operation corresponding to solving $\psi(u, v) = 0$ is $\bar{u} := \bar{u} - z\psi'_u$ where z is the solution to the linear equations $z\psi'_v = \bar{v}$. We could use AD to form the matrices ψ'_u, ψ'_v explicitly but if, for example, Newton's method is used as the iterative scheme ϕ for solving the underlying nonlinear equations, then the relevant matrices will already have been formed and factorized. Conversely, explicit formation of the derivatives produces information that can be used to improve the solution of the underlying equations, possibly at the next trial point of the function under evaluation. Similar remarks apply to preconditioning.

8 Automatic Error Analysis

It is useful to know when a function value has converged as accurately as rounding error will allow. Consider again the Wengert list of §3, with the items in the form $v_i := f_i(v_{\tau_i 1}, \ldots, v_{\tau_i n_i}) + u_i$. Suppose that the f_i instead of being floating point operations are actually smooth operations on infinite precision real numbers, and that the u_i rather than being zero are the errors introduced by round-off and normalization. Then the difference $y - \hat{y}$ between the calculated value of y and the true value $\hat{y}$ is to first order equal to $\sum_{i=n+1}^{N} u_i \bar{v}_i$. If the errors u_i are statistically independent and from symmetric distributions, and we have *a priori* or *a posteriori* error bounds $|u_i| < \Delta_i$ then the Euclidean norm $\|y - \hat{y}\|_2$ is almost certainly bounded by $4\sqrt{\sum_{i=n+1}^{N} \Delta_i^2 \|\bar{v}_i\|_2^2}$, see [20, §12]. Similarly, the use of interval analysis and the L_1-norm gives a validated error bound which is asymptotically tight [*op cit*].

Optimization algorithms almost always evaluate target functions more than once in regions where the exact target value is critical. Where an iterative

[19]We follow the convention that the elements of $\bar{A}$ have the transpose form to A. If pairs of floating point real variables are being interpreted as floating point complex numbers, then the adjoint values are conjugated as well as transposed: $y = f(v)$ and $\bar{y} = 1.0 + i0.0$ implies $\bar{v} = f'(v)^*$ since $\bar{v}_{re} + i\bar{v}_{im} = \partial y_{re}/\partial v_{re} + i\partial y_{re}/\partial v_{im} = \partial y_{re}/\partial v_{re} - i\partial y_{im}/\partial v_{re} = \partial y^*/\partial v$ by the Cauchy-Riemann equations.

solution is being used for a subproblem, therefore, it is natural to ask: when is the solution accurate enough to enable the routine evaluating the outer function to make a correct decision, and conversely how should the solution from the previous outer evaluation be used to initialize the subproblem solution, and how accurate will the resulting derivatives be?

Reverse accumulation provides some assistance with questions of this type [7]. Suppose that v is an approximate solution to $\psi(u, v) = 0$ and the exact solution is $\hat{v}$, and let the corresponding values for the dependent variables be $y, \hat{y}$. Set $w := \psi(u, v)$, then $\hat{y} = y - zw + O(\|w\|^2)$ provided z is chosen to satisfy $\|\bar{v} - z\psi_v\| < \|w\|$, and in this case $\bar{u}$ is accurate to order $\|w\|$. In the linear case $Av = b$, $w = Av - b$, giving $\hat{y} = y - zw$ to order $\|w\|^2$ provided z satisfies $\|zA - \bar{v}\| \leq \|w\|$.

9 Higher Derivatives.

We can apply first order forward or reverse mode AD repeatedly, to obtain higher order derivative values[20]. For example, applying the forward mode twice gives matrices $\ddot{v}$ with $[\ddot{y}] = P^T f'' P$. In the case of a single independent variable, we can generalize this to calculate truncated Taylor series in a particular direction. These are potentially very useful when performing line-searches. When $n > 1$ we can interpolate Taylor series to obtain derivatives of arbitrary order [4]: for example

$$\frac{\partial^2}{\partial x_1 \partial x_2} = \frac{1}{4}\left[\left(\frac{\partial}{\partial x_1} + \frac{\partial}{\partial x_2}\right)^2 - \left(\frac{\partial}{\partial x_1} - \frac{\partial}{\partial x_2}\right)^2\right].$$

We can also obtain second derivative information by combining the forward and reverse modes. In outline, we take the program $y := f(x)$, transform it using reverse mode to give the adjoint program $\bar{x} := \bar{y} f'(x)$, and then transform this using the forward mode to give the program $\dot{\bar{x}} := \dot{\bar{y}} f'(x) + \bar{y} f''(x)\dot{x}$. If we set $\bar{y} = I_m, \dot{x} = I_n, \dot{\bar{y}} = 0_{nm}$ then this gives $\dot{\bar{x}} = f''(x)$. However, sometimes it is useful to set other initial values for quantities such as $\dot{\bar{y}}$, for example if a projected Hessian is required, or as in the example of §11 below.

This approach of applying forward to reverse is particularly efficient in the case $m = 1$ of a single target variable, in which case we obtain a complete Hessian $H = f''$ at a cost of about $6n$ evaluations of f, or a projected Hessian at even lower cost. If we are using a Truncated Newton or Conjugate Gradient algorithm, or some form of gradient descent algorithm with a variable momentum term, then it is very useful to be able to evaluate terms like Hp at a computational cost which is independent of n.

[20]We can regard initialized tangent or co-tangent components in differentiated code as being additional independent variables in their own right. Subsequent code differentiation is simplified by use of identities such as $\partial v_j / \partial v_i = \partial \dot{v}_j / \partial \dot{v}_i = \partial \bar{v}_i / \partial \bar{v}_j ; \partial \bar{v}_j / \partial v_i = \partial \bar{v}_i / \partial v_j$; etc.

Applying reverse mode to forward differentiated code produces the same calculation, and hence the same result, as applying forward to reverse. All that happens is that the dots change places on the barred variables[21], so that $\bar{\bar{v}}$ corresponds to $\bar{v}$ and $\bar{v}$ corresponds to $\dot{\bar{v}}$.

Reverse differentiation of reverse differentiated code can always be replaced by forward differentiation of the original forward code. There is therefore never any need to adjoin adjoint code. For example, suppose we want to differentiate the scalar function $y := F(g(v))$ where $g = f'$ and v is a function of x. Evaluate $y := f(v)$, set $\bar{y} := 1.0$ and reverse gives $\bar{v} = f'(v)$. Set $w := \bar{v}$ and evaluation of $y := F(w)$ is straightforward. But how do we obtain $\partial y/\partial x$?

Setting $\bar{y} := 1.0$ and reversing F gives $\bar{w} := F'(w)$. Instead of adjoining $w := \bar{v}$ by setting $\bar{\bar{v}} := \bar{w}$, which would require us to adjoin the adjoint code for g to get the value for $\bar{x}$, we set $\dot{v} := \bar{w}$ and then forward and reverse through f gives $\bar{v} := \dot{\bar{v}}$ from which we can obtain $\bar{x}$ as usual. This is the numerically correct assignment, since $\dot{\bar{v}} = f''(v)\dot{v} = \bar{w}f''(v) = F'(f'(v))f''(v)$.

We can also fix tangent or co-tangent directions to be derivatives of other functions: for example if $y := f(x)$ then setting $\dot{x} := \bar{x}$ and repeating the evaluation of y and $\bar{x}$ gives the quantity $\dot{\bar{x}} = Hg$ where $H = f''(x), g = f'(x)$. Accurate quantities of this type are useful in many gradient descent algorithms, including Truncated Newton.

10 Overloading and Program Transformation

The overloading approach is quick to implement, but suffers from a number of disadvantages. Most compilers implement expressions containing overloaded operators exactly as they are written, without performing any compile-time optimization on the expression. For example, the assignment

$$y := a * \sin(a * x * *2 + b * x + c) + b * \cos(a * x * *2 + b * x + c)$$

contains the shared subexpression $a * x * *2 + b * x + c$, which need only be evaluated once, and which would be more efficiently evaluated as $(a * x + b) * x + c$. Consequently an overloaded doublet implementation will be considerably less efficient than the optimized underlying floating point implementation, even before the costs of the extra floating point operations are taken into account[22]

[21] Conceptually different sets of dots and bars are used, corresponding to different tangent and co-tangent variables. Strictly we should use a tensor derivative notation for repeated differentiation.

[22] There are good reasons for this literal-minded compilation. Overloaded operators may have complex side-effects involving global state, and in any case cannot generally be assumed to have the same semantics as their built-in counterparts. For example, matrix multiplication is not commutative, octonian multiplication is not associative, intervals do not satisfy the distributive law, and common subexpressions involving random oracles must be recomputed for each occurrence. Most overloaded operator languages give the user no way to tell the compiler which optimizing transformations are safe.

Nevertheless, there is no better way to understand AD than to implement a baby AD tool using operator overloading and for many small to medium size problems such a tool is adequate.

Transforming the underlying program to a new source program, rather than augmenting it using overloaded operators, allows the compiler to perform optimization on the derivative calculations as well as upon the underlying calculations. For example, when adjoining the assignment to y, the derivatives of sin and cos are already available, and the derivative of the argument can be obtained by adding the two available quantities $a * x$ and $a * x + b$.

With a language translation approach, a great deal more can also be done to automate the dependency analysis required to determine which variables are active, although when array indices or pointers are manipulated in a complex way at run time, the translator must make a conservative assumption, or rely upon user-inserted directives. Deferring choices until run time almost inevitably produces code which runs more slowly than when the decision can be made at compile time.

The output from the translator is input to an optimizing compiler, so there is generally no need for the code to be particularly efficient; rather, the translator must produce code which it is easy for the compiler to analyse and optimize. This requirement is certainly compatible with making the transformed code intelligible to humans, and users have become accustomed to being able to write source code in a form that is intelligible to them, and to rely upon the compiler to re-arrange it into a form which is efficient before producing object code.

11 Pantoja's Algorithm and Checkpointing

In this section, we show how Automatic Differentiation can be combined with Pantoja's algorithm and a checkpointing technique in such a way as to allow accurate evaluation of the Newton direction for a discrete time optimal control problem at an extremely low computational cost [8]. The purpose of this example is to show the combined use of forward and reverse mode AD to produce Hessian information, and to illustrate how checkpointing can be combined with parallel processing to reduce the run-time storage requirement to something feasible.

Consider the following discrete-time optimal control problem: choose independent control variables $x_i \in R^p$ so as to minimize the scalar target function

$$y = F(v_N) \quad \text{where } v_{i+1} = f_i(x_i, v_i) \quad \text{for } 0 \leq i < N$$

and v_0 is some fixed constant. Each f_i is a smooth map from $R^p \times R^q \to R^q$ and F is a smooth map from the state space R^q to R: the states v_i may include running totals of cost functions which are composed into y by F.

Starting with stored values for $x_i : 0 \leq i < N$, we seek the Newton direction, ie vectors $t_i \in R^p$ such that

$$\sum_{j=0}^{N-1} \left[\frac{\partial^2 y}{\partial x_i \partial x_j} \right] t_j + \frac{\partial y}{\partial x_i} = 0 \quad \text{for } 0 \leq i < N$$

Pantoja [21] gives an algorithm for calculating the Newton direction exactly. However his algorithm involves the solution of linear equations with coefficients given by recursive identities such as:

$$\begin{aligned} A_i &= \left[f'_{v,i}\right]^T D_{i+1} \left[f'_{v,i}\right] + \bar{v}_{i+1} \left[f''_{vv,i}\right] \\ B_i &= \left[f'_{x,i}\right]^T D_{i+1} \left[f'_{v,i}\right] + \bar{v}_{i+1} \left[f''_{xv,i}\right] \\ C_i &= \left[f'_{x,i}\right]^T D_{i+1} \left[f'_{x,i}\right] + \bar{v}_{i+1} \left[f''_{xx,i}\right] \\ D_i &= A_i - B_i^T C_i^{-1} B_i \qquad \bar{v}_i = \bar{v}_{i+1} \left[f'_{x,i}\right] \end{aligned}$$

which in turn requires the accurate evaluation of terms containing second derivatives of f_i. Fortunately AD can be applied to the original code for evaluating F in such a way that the values $\dot{\bar{x}} = \dot{\bar{y}} f'(x) + \bar{y} f''(x) \dot{x}$ are exactly the quantities required [8]. A primary benefit of AD here is the elimination of the labour of forming and differentiating adjoint equations by hand, however the total flop-cost of the AD-form of the algorithm is of the same order as $6(p+q)$ evaluations of the target function y, regardless of the number of timesteps N.

Algorithm (Pantoja with AD)

(1) For i from 1 upto N, calculate and store v_i.

(2) Evaluate $a_N = \bar{v}_N = F'(v_N), D_N = [F''(v_N)]$ as described in §9 above.

(3) For each i from $N-1$ down to 0 calculate $q-$vectors $\bar{v}_i, a_i$ and a $q \times q$ matrix D_i as follows.

(3.1) Define dot-vectors of length $p+q$ by

$$\begin{bmatrix} \dot{x}_i \\ \dot{v}_i \end{bmatrix} = \begin{bmatrix} I_p & O \\ O & I_q \end{bmatrix}$$

(3.2) Evaluate $v_{i+1} = f_i(x_i, v_i)$ using forward mode AD, so that

$$[\dot{v}_{i+1}] = [f'_{x,i} \; f'_{v,i}].$$

(3.3) Set $\bar{v}_{i+1}$ to the value supplied by the previous iteration and set

$$[\dot{\bar{v}}_{i+1}] := [D_{i+1} f'_{x,i} \; D_{i+1} f'_{v,i}].$$

(3.4) Apply the forward mode of AD to the forward calculation $v_{i+1} := f_i(x_i, v_i)$ and then to the adjoint calculation $[\bar{x}_i\ \bar{v}_i] := \bar{v}_{i+1} f_i'(x_i, v_i)$, giving the matrix

$$\begin{bmatrix} \dot{\bar{x}}_i \\ \dot{\bar{v}}_i \end{bmatrix} = \begin{bmatrix} C_i & B_i \\ B_i^T & A_i \end{bmatrix}$$

(3.5) Row reduce this to obtain

$$\begin{bmatrix} I & C_i^{-1} B_i \\ O & A_i - B_i^T C_i^{-1} B_i \end{bmatrix} = \begin{bmatrix} I & E_i \\ O & D_i \end{bmatrix}$$

and at the same time calculate the vectors

$$a_i = a_{i+1} \left([f'_{v,i}] - [f'_{x,i}] E_i \right), \qquad c_i^T = -a_{i+1} [f'_{x,i}] C_i^{-1}.$$

Now $\bar{v}_i, a_i, D_i$ are available for the next iteration.

(3.6) Store the values $\dot{v}_{i+1}, \bar{x}_i, E_i, c_i$.

(4) For each i from 0 up to $N-1$ calculate $t_i \in R^p, s_{i+1} \in R^q$ by

$$s_0 = 0, \quad t_i = c_i - E_i s_i, \quad s_{i+1} = \left[f'_{x,i}\right] t_i + \left[f'_{v,i}\right] s_i$$

Now t_i is the Newton direction.

STOP

Many other solution techniques which use state-control feedback can be implemented as simple modifications of this algorithm. For example differential dynamic programming (DDP) replaces the vectors a_i by $\bar{v}_i$ in the calculation for c_i. AD in principle allows algorithms of this form, combined with the techniques for differentiating implicit equation solutions, to be applied to Differential Equations.

Reducing the storage requirement. By using the state values v_i as checkpoints, we can reduce the storage requirement of the reverse mode to that required for a single timestep f_i together with one checkpoint per timestep. Each checkpoint requires storage for the state vector v_i together with the values $\dot{v}_{i+1}, \bar{x}_i, E_i, c_i$.

However, a much more efficient use of checkpoint storage than this is possible. For example, suppose that N is a million. If we store values for $x_i, \bar{x}_i, D_i$ whenever i is a multiple of a thousand, then we can re-compute the values of E_i, c_i etc when we need them, in groups of a thousand at a time. This doubles the total computational effort required but reduces the storage requirement from a million full checkpoints to a thousand primary plus a thousand additional checkpoints.

This line of argument can be developed further: with a third level of checkpoint we require three times the computational cost, but storage for only three hundred checkpoints. With six levels these numbers are 6 and 60, and with

$20 = \log_2 N$ levels of checkpoint we require just $\log_2 N$ times the computational effort together with storage for $\log_2 N$ checkpoints. For this example the storage requirement for reverse accumulation is therefore less than the storage already required to hold the values of the control variables.

In fact, by spacing the checkpoints irregularly we can halve these requirements [16]. If we have several processors available, we can use them to recalculate the various levels of checkpoint in parallel with the main algorithm so that the required values are ready just in time. It is instructive to work out in detail the schedule for doing this in such a way that the overall runtime does not increase as the storage requirement reduces [2].

12 Fletcher's Ideal Penalty Function

.

In this section we show how AD can be used to evaluate and differentiate a parameter-free form of a penalty function introduced by Fletcher [12]. The purpose of this example is to illustrate the differentiation of functions which combine nested subproblem solution with the calculation of gradients of other functions.

Consider the constrained optimization problem: optimize $f(x)$ subject to $k(x) = 0$ where f, k are smooth maps $R^n \to R$ and $R^n \to R^q$ respectively. Set $g = f', N = k'$ to be the function gradient and constraint normals, and define $\lambda(x), \mu(x) \in R^q$ by the equations

$$NN^T\lambda = Ng, \qquad NN^T\mu = k.$$

Now define $\nu(x) \in R^n$ by $\nu = N^T\mu$ and $F : R^n \to R$ by

$$F(x) = f(x - \nu(x)) + \sum_{i=1}^{q} \lambda_i(x) k_i(x - \nu(x)) + \frac{1}{2}\sum_{i=1}^{q} \nu_i^2(x))$$

Under mild conditions we have [6] that (i) if x^* is a constrained local minimum of f subject to $k = 0$ then x^* is an unconstrained local minimum of F and conversely (ii) if x^* is an unconstrained local minimum of F satisfying $k = 0$ then x^* is a constrained local minimum of f. It follows that if x^* is a constrained minimum of f subject to $k = 0$ then there is a neighbourhood of x^* in which x^* is the only unconstrained local minimum of F, and minimizing F in this neighbourhood will find x^*.

The penalty function F also has the desirable property that near a minimum point the penalty function has the same curvature as the Lagrangian of the target function in directions tangent to the constraint manifold, and unit positive curvature in directions normal to the constraint manifold. Thus F has numerical conditioning similar to that of the target function f and constraints k from which F is constructed.

We can evaluate F as follows. Solve the equation $NN^T\mu = k$ for μ using AD to evaluate NN^T. For example, we could set $y := k(x); \bar{y} := [I_q]$ and reverse to get $\bar{x} = N^T$. Then set $\nu := N^T\mu$. Similarly λ is the solution of $NN^T\lambda = Ng$, where reverse accumulation gives g. Now it is a simple matter to compute the value of F.

We can use AD to obtain the gradient and directional Hessians of F, and these can be used by optimization software to find a local minimum point x^* of F which corresponds to the solution of the original constrained problem. For example, the adjoint of the step "solve $NN^T\mu = k$ for μ" is "solve $\xi NN^T = \bar{\mu}$ for ξ then set $\bar{k} := \bar{k} + \xi, \bar{N} := \bar{N} - \nu\xi - (\mu\xi N)^T$", and the adjoint of the step $N := \bar{x}^T$ is to set $\dot{x} := \bar{N}$ then go forward and reverse through the calculation of k and set $\bar{x} := \bar{x} + \dot{\bar{x}}$.

If q is large we may prefer an iterative method of solving the linear equations for n and λ such as Conjugate Gradient, which in turn requires evaluation of vectors such as NN^Tp. Reverse accumulation also allows automatic error estimates to be made for the effect of truncating a subproblem solution upon the calculated function value as described in §8 above. This allows us to solve the equations for λ and μ with just sufficient accuracy to ensure that the calculated value of $F(x)$ is correct to the required accuracy (specified in advance) at each iteration step of the optimization algorithm. We can even apply AD to the implicit equations defining x^* so as to perform an automatic error analysis or to determine sensitivities of the solution.

13 Future Directions

Several themes for future developments emerge from this. AD has largely achieved its initial agenda of producing fast, accurate derivative code without the costly and error-prone intervention of well-intentioned humans. An analogy can be drawn with the experiences gained by automating the process of translating computer programs from high-level language descriptions into machine code, and from this perspective the future of AD is increasing bound up with the process of compiler-writing and language translation generally. More and more scientific compilers will contain AD algorithms, or at least hooks to allow AD algorithms to be invoked during the compilation process. A great deal of research still remains to be done in this area, particularly in the case of parallelizing compilers, but increasingly the task of AD in this context is to formulate the program transformation problem in terms which enable it to be solved by existing and emerging compiler-generator tools.

Although a great number of AD users are content simply to apply AD to their existing code, this is not the end of the story. At the opposite extreme from the legacy-code user are those doing research into non-linear optimization algorithms. Taking (for brevity of exposition) a somewhat combative stance, we could assert that many optimization algorithms were initially designed upon the implicit assumption that gradient information was, by its nature, expensive and inaccurate relative to the function evaluation. Second derivative information was likely to be even worse, and any algorithm which required third or higher order derivative information was not viable. The current state of AD implies that even quite mild forms of this position are no longer tenable.

While many traditional algorithms work extremely well even in very large dimensions when given accurate derivatives[23], the contribution of AD to algorithm design remains open. Certainly the ability of reverse accumulation to give complete, accurate gradient and directional Hessian vectors at a cost of a few function evaluations, regardless of the problem dimension, influences the choice of algorithm and the globalization strategy for problems in very large dimension, and we identify this as one context in which AD is likely to develop further from a theoretical point of view. The interaction between AD and Interval Analysis is another interesting arena for future development.

Many by-products produced during reverse accumulation are of a type which could naturally be exploited during the optimization process by an algorithm with knowledge of the target function's structure, and conversely explicit representation of such structure would in many cases allow an AD tool to operate more effectively. In particular, when equation solution is a sub-problem, there is a benefit to coding the equations being solved as well as the code to solve them, even if the solution code never evaluates the equations, in order to allow the residuals and their derivatives to be used by the AD tool. Likewise there is a benefit to signaling explicitly to an AD tool the accuracy to which derivatives are required, and the use to which they will subsequently be put.

Perhaps the most ambitious way forward for the next few years is the development of AD as a conceptual tool to allow users to capture and express their insights into the nature and structure of the algorithms which their programs instantiate, and to develop new ways of representing these algorithms beyond those offered by current programming languages, in such a way that these insights can be automatically exploited by the environment in which their programs run.

References

[1] Michael Bartholomew-Biggs, 1998, Using Forward Accumulation for Automatic Differentiation of Implicitly Defined Functions, Computational Optimization and Applications, **9**, 65–84.

[2] Jochen Benary, 1996, Parallelism in the Reverse Mode, pp 137–147 *in* [3]

[23]In fairness, it should be mentioned that some traditional algorithms rely upon the inaccuracy of supplied derivatives in order to avoid saddle points, which tend to proliferate in large dimensions.

[3] Martin Berz et al (Editors), 1996, Computational Differentiation : Techniques, Applications and Tools, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.

[4] Christian Bischof *et al*, 1993, Structured Second- and Higher-Order Derivatives through Univariate Taylor Series, Optimization Methods and Software, **2**, 211–232.

[5] Steven Brown and Bruce Christianson, 1997, Automatic Differentiation of Computer Programs in a Parallel Computing Environment, pp 169–178 *in* H. Power *and* J. Casares Long (Editors), *Applications of High Performance Computing in Engineering vol V*, Computational Mechanics Publications, Southampton, UK.

[6] Bruce Christianson, 1993, A Geometric Approach to Fletcher's Ideal Penalty Function, Journal of Optimization Theory and Applications, **84**(2), 433–441.

[7] Bruce Christianson, 1998, Reverse Accumulation and Implicit Functions, Optimization Methods and Software, **9**(4), 307–322.

[8] Bruce Christianson, 1999, Cheap Newton Steps for Optimal Control Problems : Automatic Differentiation and Pantoja's Algorithm, Optimization Methods and Software, **10**(5), 729–743.

[9] Bruce Christianson, Laurence Dixon *and* Steven Brown, 1996, Sharing Storage using Dirty Vectors, pp 107–115 *in* [3].

[10] Bruce Christianson *et al*, 1997, Giving Reverse Differentiation a Helping Hand, Optimization Methods and Software, **8**(1), 53–67.

[11] Laurence Dixon, 1991, Use of Automatic Differentiation for Calculating Hessians and Newton Steps, pp 114–125 *in* [19]

[12] Roger Fletcher, 1970, A Class of Methods for Nonlinear Programming with Termination and Convergence Properties, pp 157–175 *in* Integer and Nonlinear Programming, Edited by J. Abadie, North Holland, Amsterdam, Holland.

[13] Ralf Geiring *and* Thomas Kaminski, 1998, Recipes for Adjoint Code Construction, ACM Transactions on Mathematical Software, **24**(4), 437–474.

[14] Jean Charles Gilbert, 1992, Automatic Differentiation and Iterative Processes, Optimization Methods and Software, **1**, 13–21.

[15] David Gries, 1981, The Science of Programming, Springer Verlag.

[16] Andreas Griewank, 1992, Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation, Optimization Methods and Software **1**(1) 35–54.

[17] Andreas Griewank, 2000, Evaluating Derivatives : Principles and Techniques of Algorithmic Differentiation, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.

[18] Andreas Griewank *and* Shawn Reese, 1991, On the Calculation of Jacobian Matrices by the Markowitz Rule, pp 126–135 *in* [19]

[19] Andreas Griewank *and* George Corliss (Editors), 1991, Automatic Differentiation of Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.

[20] Masao Iri, 1991, History of Automatic Differentiation and Rounding Error Estimation, pp 3–16 *in* [19]

[21] J.F.A.De O. Pantoja, 1988, Differential Dynamic Programming and Newton's Method, Int J Control, **47**(5), 1539–1553.

[22] Dimitri Shiraev, 1993, Fast Automatic Differentiation for Vector Processors and Reduction of the Spatial Complexity in a Source Translation Environment, PhD Dissertation, Karlsruhe, Europe.

[23] Yu. M. Volin *and* G.M. Ostrovskii, 1985, Automatic Computation of Derivatives with the use of the Multilevel Differentiation Technique, Computers and Mathematics with Applications, **11**(11), 1099–1114.

[24] R.E. Wengert, 1964, A Simple Automatic Derivative Evaluation Program, Communications of the ACM, **7**, 463–464.