An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB

SHAUN A. FORTH
Cranfield University (Shrivenham Campus)

The Mad package described here facilitates the evaluation of first derivatives of multi-dimensional functions that are defined by computer codes written in MATLAB. The underlying algorithm is the well-known forward mode of automatic differentiation implemented via operator overloading on variables of the class fmad. The main distinguishing feature of this MATLAB implementation is the separation of the linear combination of derivative vectors into a separate derivative vector class derivvec. This allows for the straightforward performance optimisation of the overall package. Additionally by internally using a matrix (two-dimensional) representation of arbitrary dimension directional derivatives we may utilise MATLAB’s sparse matrix class to propagate sparse directional derivatives for MATLAB code which uses arbitrary dimension arrays. On several examples the package is shown to be more efficient than Verma’s ADMAT package.

1. INTRODUCTION

In the standard reference for the subject, Griewank [2000] states that, Algorithmic, or automatic differentiation (AD) is concerned with the accurate and efficient evaluation of derivatives for functions defined by computer programs. AD uses the systematic application of the chain rule of differentiation applied to the floating point representation of a variable’s value and its derivatives. Unlike the finite-difference approximation, no discretisation or cancellation errors are incurred, and the resulting derivative values are accurate to within floating-point round-off. Since only floating point values are used (unlike differentiation within symbolic algebra packages such as Mathematica or Maple) good efficiency may be obtained. Additionally, AD permits the use of control structures (loops, branches and sub-functions) common to modern computer languages but not easily amenable to symbolic differentiation.
1.1 Forward and Reverse Mode AD

There are two fundamental algorithms or modes of AD for calculating first derivatives – forward and reverse [Griewank 2000, Chap. 3].

**Forward, or Tangent(-Linear), Mode AD** involves enhancing the original function code so that a variable’s directional derivatives are calculated along with its value. For example, if we set the values of scalar variables $x_1$ and $x_2$ and their scalar derivatives $Dx_1$ and $Dx_2$, then a variable defined by the statement $y = x_1 \times x_2$ must have its derivatives $Dy$ calculated via the product rule as $Dy = x_1 \times Dx_2 + Dx_1 \times x_2$.

Clearly, if we initialise $Dx_1 = 1$ and $Dx_2 = 0$, for arbitrary, initialised values of $x_1$ and $x_2$, we obtain the value of the directional derivative of $y$ in the $x_1$ coordinate direction. If we define $Dx_1$ and $Dx_2$ to be vectors of length 2 with $Dx_1 = (1, 0)$ and $Dx_2 = (0, 1)$, and interpret $Dy = x_1 \times Dx_2 + Dx_1 \times x_2$ as a vector statement, the calculated $Dy$ is the gradient of $y$ for the supplied values of $x_1$ and $x_2$. By systematically performing such derivative operations for all the necessary floating point operations in the original function code, following the control flow (through branches, loops, sub-functions) as dictated by the values of variables, then gradients of all variables may be calculated.

**Reverse, or Adjoint, Mode AD** is a two stage process. First the original function code is run, perhaps augmented by statements to store data to enable the code to be run a second time in reverse, propagating the sensitivities of the function’s output to each calculated variable. Such sensitivities are termed adjoints.

Griewank [2000] presents a computational complexity analysis for the run time, $\text{time}(\mathbf{Jf}(x))$, to calculate the Jacobian $\mathbf{Jf}(x)$ of a function $f(x) \in \mathbb{R}^n \rightarrow \mathbb{R}^m$ by both forward and reverse mode AD. For the forward mode, $\text{time}(\mathbf{Jf}(x)) = \omega_{fwd} \times \text{time}(f(x))$, where $\text{time}(f(x))$ is the run-time for the original function. The coefficient $\omega_{fwd} \in [1 + n, 1 + 1.5n]$ is dependent on machine characteristics (e.g., relative time for memory access compared to floating point operation). In a similar manner for reverse mode AD $\text{time}(\mathbf{Jf}(x)) = \omega_{rev} \times \text{time}(f(x))$ with $\omega_{rev} \in [1 + 2m, 1.5 + 2.5m]$ again dependent on machine characteristics. The larger coefficients in the bounds for $\omega_{rev}$ compared to $\omega_{fwd}$ reflect the extra memory operations in the reverse mode’s reverse pass to recover values stored in its forward pass. Such operations are not needed in the single pass forward mode.

If $m \ll n$, i.e. the gradients of a small number of function outputs are required with respect to a large number of function inputs, then reverse mode is to be preferred. Examples of such cases typically arise in large-scale optimisation.

1.2 Implementation of AD

AD is implemented in one of two ways: operator overloading or source transformation [Griewank 2000, Chapter 5].

The **operator overloading** approach takes advantage of the facility to define new classes (or types) within modern computer languages such as Fortran 95, C++ or MATLAB. Objects of the new AD class are defined to have a component which stores their value and components to store derivative information. Arithmetic and intrinsic functions are extended to the AD class making use of operator and function overloading. In typed languages such as Fortran or C++, all that remains is for the user to redefine the classes of all relevant objects within the function and all...
sub-functions to that of the AD class, initialise appropriate values and derivatives, invoke the function, and then extract the values of the derivatives. Representative examples of such implementations are the packages ADO1 [Pryce and Reid 1998] and ADOL-C [Griewank et al. 1996].

The alternative source transformation approach requires the development of sophisticated compiler-type software to read in a computer program, determine which statements require differentiation, and then write a new version of the original program augmented with statements to calculate derivatives. Such sophisticated AD source transformation packages exist for languages such as Fortran [Bischof et al. 1996; FastOpt 2003; Tapenade 2003] and C [Bischof et al. 1997].

1.3 AD in MATLAB
Rich and Hill [1992] provided a limited facility for MATLAB that enabled AD of simple arithmetic expressions defined by a character string. Such strings, together with necessary values of variables were passed to an external routine, written in turbo-C, for differentiation. However, the first significant work was that of Coleman and Verma [1998b; 1998a],[Verma 1998b] who, in a monumental coding effort, produced an operator-overloading AD package named ADMAT that provides facilities for forward and reverse mode AD for both first and second derivatives and runtime Jacobian sparsity detection. These authors also interfaced ADMAT with ADMIT [Coleman and Verma 2000], a package for efficient sparse Jacobian calculation via various colouring algorithms. Recently the ADiMat hybrid source-transformation/operator overloading AD tool [Vehreschild 2001] has been developed, and comparisons [Bischof et al. 2003; Bischof et al. 2002] show its forward mode to be more efficient than that of ADMAT. It would appear that while ADMAT’s operation count is in agreement with AD theory (see for example the operations counts in [Borggaard and Verma 2000]), its run time is not. We have encountered similar experiences leading to development of the Mad AD package [Forth 2001].

1.4 The Mad AD Package
Our aim in developing the Mad AD package is to implement in MATLAB the various AD algorithms in a careful, step-wise manner, taking care to ensure efficiency and ease of extension at each stage. The first AD algorithm to be implemented was, as described in this paper, the standard forward mode for first derivatives. This immediately provided a facility for calculating derivatives accurate to floating point round-off. Such derivatives are required for Newton-based algorithms for both nonlinear equation solution and optimization. In the absence of AD such derivatives are more conventionally approximated by finite differencing (FD). In order to replace FD as the default method for derivative evaluation, we must show that AD has comparable efficiency for derivative evaluation and further that its inherent accuracy yields greater overall efficiency and robustness when used within derivative-exploiting algorithms.

An early decision, based on expedience, was to adopt an operator-overloading implementation. Programming overloaded classes, operations and functions allows for a rapid coverage of most commonly used MATLAB functionality. Indeed, the first version of the forward mode was written, debugged and facilitating improved...
statistical data fitting [Ringrose and Forth 2002] for less than a man-month’s effort. The alternative source-transformation approach would be far more time-consuming.

We describe our operator-overloaded implementation of forward-mode AD for a single-directional derivative through the \texttt{fmad} class in Section 2. Our implementation of the \texttt{derivvec} class for storing and manipulating multiple directional derivatives, described in Section 3, allows us to enhance the \texttt{fmad} class to propagate an arbitrary number of directional derivatives. This separation of the storage and manipulation of directional derivatives to within the \texttt{derivvec} class allows us to greatly optimize overall performance by optimizing performance of the small number of functions of the \texttt{derivvec} class. To ensure efficiency the functions of the \texttt{derivvec} class only use high-level array operations [The MathWorks Inc. 2003, Section 22]. Also, internal to the \texttt{derivvec} class we use a 2-dimensional array (i.e., a matrix) for storing directional derivatives associated with arrays of arbitrary dimensions. This allows us to use matrices of MATLAB’s \texttt{sparse} class to store sparse directional derivatives with commensurate reduction in run-time and memory-requirements for many large calculations. Section 4 describes how to use the \texttt{fmad} class for Jacobian evaluation using dense, sparse and compressed storage of directional derivatives. Several examples in Section 5 demonstrate the effectiveness of our approach. Section 6 concludes and gives plans for future work.

2. FORWARD MODE AD FOR A SINGLE DIRECTIONAL DERIVATIVE

It is straightforward to implement forward mode AD for a single directional derivative via operator overloading in MATLAB. In Section 2.1 we briefly outline how this is achieved since in Section 3 we will re-use essentially the same MATLAB functions to handle arbitrary numbers of directional derivatives. Although the implementation of this section is extremely simple, it is of immediate practical use for differentiating functions of a single variable as seen in Section 2.2 or conceivably in the matrix-free iterative solution of large-scale nonlinear systems [Nocedal and Wright 1999, p285].

2.1 Implementation of the Forward Mode

The forward mode is implemented via a MATLAB class \texttt{fmad}. A MATLAB class consists of a set of functions that create and manipulate objects of that class. Here the manipulations that concern us are extending the arithmetic operations of MATLAB to those that calculate both an object’s value and an associated directional derivative.

Use of the \texttt{fmad} class to calculate directional derivatives of a user’s MATLAB function or statements is straightforward. The user first initializes value and deriv components of objects of \texttt{fmad} class corresponding to those variables we need derivatives with respect to using the \texttt{fmad} constructor function of Section 2.1.2. Then the user’s MATLAB function or statements are executed to propagate both values and directional derivatives via overloaded arithmetic operations and intrinsic functions as described in Sections 2.1.3 and 2.1.4. Values and derivatives associated with \texttt{fmad} objects are obtained using the extraction functions described in Section 2.1.5.

2.1.1 \texttt{fmad} Objects. \texttt{fmad} objects have two components, value and deriv. The value component stores an object’s value as a class \texttt{double} or \texttt{sparse} array. The
function xad=fmad(x,dx)
% Set value component
switch class(x)
    case {'double','sparse'}
        xad.value=x;
    case 'fmad'
        xad.value=x.value;
    otherwise
        error('FMAD: first argument must be class double, sparse or fmad')
end
sx=size(xad.value);
% Set deriv component
if nargin==2
    switch class(dx)
        case 'derivvec'
            xad.deriv=reshape(dx,size(xad.value));
        case {'double','sparse'}
            sd=size(dx);
            if prod(sx)==prod(sd)
                xad.deriv=reshape(dx,sx);
            else
                xad.deriv=derivvec(dx,size(xad.value));
            end
        otherwise
            error(['FMAD/FMAD: argument dx of illegal class',class(dx)])
    end
    xad=class(xad,'fmad');
else
    error('FMAD: must supply 2 arguments')
end

Fig. 1. Constructor function for the fmad class.

deriv component stores directional derivatives associated with the object. This component may be of class double, sparse or, in the case of multiple directional derivatives, class derivvec. Throughout this section we restrict our description to a single directional derivative, and thus the value and deriv components are of the same size\(^1\) and of class double or sparse. Variables of fmad class are created via the fmad constructor function which necessarily resides in the @fmad sub-directory.

2.1.2 The fmad Constructor Function. In Figure 1, we show the fmad constructor function with, as will be our practice in this paper, some comments removed for brevity. This function takes two inputs, \(x\) and \(dx\), and returns an object \(xad\) of fmad class. Generally a user will supply an array \(x\) of values of class double or sparse, and an associated directional derivative \(dx\) also of class double or sparse. In such cases \(x\) is assigned to the value component \(xad.value\) of the function output \(xad\). Assuming the second argument to fmad is also of class double or sparse then, if it has the same number of elements as the value, it is reshape\(d\) to the same size as the value before being assigned to the deriv component of the output \(xad\).

\(^1\)Here we use the term size in the MATLAB sense that the two have the same number of dimensions and number of elements in each dimension
Fig. 2. The `times` function of the `fmad` class for element-wise multiplication.

```matlab
function z=times(x,y)
if isa(x,'fmad')&isa(y,'fmad')
    z.value=x.value.*y.value;
    z.deriv=x.deriv.*y.value+...+x.value.*y.deriv;
elseif isa(x,'fmad')
    z.value=x.value.*y;
    z.deriv=x.deriv.*y;
else
    z.value=x.*y.value;
    z.deriv=x.*y.deriv;
end
z=class(z,'fmad');
```

Fig. 3. The `sin` function of the `fmad` class

```matlab
function y=sin(x)
xval=x.value;
y.value=sin(xval);
y.deriv=cos(xval).*x.deriv;
y=class(y,'fmad');
```

We also see that if the first input is already of `fmad` class then the function may be used to change just the derivative `deriv` component. Should the second argument `dx` correspond to multiple directional derivatives then this is handled via the `derivvec` class, discussion of which is postponed until Section 3.

2.1.3 Propagating Derivatives Via Operator and Function Overloading. Once `fmad` objects have been initialised then values and derivatives must be propagated by invoking overloaded versions of all intrinsic arithmetic operations and functions that comprise a user’s MATLAB code.

In Figure 2, we show the coding of the `times` function of the `fmad` class invoked to calculate the element-wise multiplication of two arrays `z=x.*y` when one or more of `x` and `y` are of `fmad` class. As the function’s coding indicates, first the MATLAB intrinsic function `isa` is used to determine which of `x` and `y` are of `fmad` class. If both are of `fmad` class, then the `value` component of each is used to determine the appropriate value component for the output `z.value=x.value.*y.value`. Then the output’s directional derivative component is determined by the product rule `z.deriv=x.deriv.*y.value+x.value.*y.deriv` using both the values (`x.value`, `y.value`) and derivatives (`x.deriv`, `y.deriv`) of the inputs `x` and `y`. If just one of `x` or `y` is of `fmad` class then the appropriate branches are correspondingly simpler. Finally, the output `z` is cast to be of `fmad` class using the `class` intrinsic function.

On grounds of efficiency, and following Verma [1998a], we have adopted this casting approach to ensure `z` is of `fmad` class, rather than use the class constructor `fmad` function as advocated by the MATLAB documentation [The MathWorks Inc. 2003, Section 21].

In a similar manner, and using undergraduate level calculus, we may overload other arithmetic operations: addition (`+`), subtraction (`-`), matrix multiplication (`*`), element-wise division (`./`). We may also overload intrinsic functions, such as `sin`, as shown in Figure 3.
2.1.4 Dealing with Array Indexing. A major difference between operator overloading in MATLAB and that in traditional programming languages such as Fortran is in the handling of arrays. In Fortran arrays are regarded as an ordered collection of their scalar components. Consequently the referencing of components via subscripts is built into the language and, even for derived types (c.f., objects in MATLAB), need not be coded. In MATLAB all objects are arrays, and programmers must provide functions to deal with array subsetting.

In Figure 4, we show the `subsref` function invoked for array subscript referencing of `fmad` objects. For example, if \( y \) is of `fmad` class then \( x = y(1:5,:) \) invokes the function call `subsref(y,s)`, where \( s \) is a MATLAB structure with components \( s\.type = '()' \), and \( s\.subs = \{1:5,:\} \). The \( s\.type \) indicates the type of subsetting, with the string `()` indicating conventional array indexing as opposed to cell array subsetting or structure component referencing. The \( s\.subs = \{1:5,:\} \) component gives the actual indexing to be applied to \( y \). As is seen from Figure 4, the `fmad` version of `subsref` simply uses two calls to the intrinsic `subsref` to perform equivalent subsetting on the `value` and `deriv` components of an `fmad` object.

Subscript assignments of, for example \( a(1:5,:) = b \), assign an array (here \( b \)) to a subscript defined sub-region of another array (here \( a(1:5,:) \)). For single-directional derivatives an `fmad` class `subsasgn` function may be written in a similar manner to that for `subsref`.

Usually when designing a new MATLAB class we would provide a function `double` to convert an object of `fmad` class to intrinsic class `double`. We have provided such a function, but it always produces an error message and halts execution of a user’s MATLAB function or script. We have taken this step since if a variable, \( a \), is already of class `double` and we assign an `fmad` variable \( b \) to a component of it, for example \( a(1)=b \), then MATLAB will use the `fmad` class’s `double` function to convert \( b \) to class `double` before the assignment. This will prevent the correct propagation of derivative information and, unknown to the user, the derivatives produced will be incorrect. By trapping this situation as a run-time error the user may take the appropriate action (ensuring \( a \) is of class `fmad`) so that derivative propagation proceeds correctly.

To minimise the number of instances in a user’s code for which we would assign an `fmad` object to a component of a non-`fmad` array we have adopted a technique of Verma [1998]. In MATLAB arrays are frequently created using the `ones` or `zeros` intrinsics, for example to initialise an array \( a \) to be of the same size of an object \( b \) but consist of all zero entries the statement `a=zeros(size(b))` could be used. In MAD we ensure that if \( b \) is an `fmad` object then: `size(b)` returns an `fmad` object; `zeros(size(b))` returns an `fmad` object; consequently \( a \) is an `fmad` object, and we may subscript assign other `fmads` to it, e.g., \( a(1)=b(1) \). The difference between Verma’s implementation of this technique and ours is that whereas ADMAT uses an empty matrix for the derivatives of the object returned by the `size` function.
\texttt{fmad} stores the derivatives as zeros; this trivial memory overhead in our approach is more than compensated for by the reduction in complexity obtained by never having to test if a derivative component is empty.

2.1.5 Extracting Values and Derivatives. Functions \texttt{getvalue} and \texttt{getderivs} are provided that extract the value and deriv components of an \texttt{fmad} variable.

2.2 Use of the \texttt{fmad} Class

Trivially the \texttt{fmad} class may be used to obtain the derivative of a function of a single variable. For example, consider the function \( y = x^2 + x \). If we require the derivative at \( x = 2 \) then we may simply type at the MATLAB prompt,

\begin{verbatim}
x=fmad(2,1);
y=x^2+x;
dy=getderivs(y)
\end{verbatim}

to get the output

\begin{verbatim}
dy =
5
\end{verbatim}

In Section 3 we consider extending the \texttt{fmad} class to calculate multiple directional derivatives with the aid of the \texttt{derivvec} class.

3. DERIVATIVE VECTORS

In Section 2, we have seen how forward mode AD calculates objects’ values and their derivatives as the source code is executed. We see that the derivatives are calculated as linear combinations of derivatives previously calculated. In order for AD to calculate the derivative of one or more outputs with respect to one scalar input requires a single derivative value to be calculated for each scalar value calculated in the code. Usually, calculation of \texttt{nderivs} directional derivatives requires either the propagation of \texttt{nderivs} directional derivatives (or \texttt{nderivs} runs of the AD code for a single directional derivative). Exceptions to this are when the calculation possesses some intrinsic sparsity which may be exploited [Griewank 2000, Chapters 6,7] or [Coleman and Verma 1998b], in which case a number less than \texttt{nderivs} might be used.

In order to calculate multiple directional derivatives simultaneously and efficiently, we aimed to optimise the operations associated with linear combinations of derivative vectors. Additionally we wished to exploit MATLAB’s sparse matrix operations [Gilbert et al. 1992] to reduce memory requirements and improve performance for large scale calculations that exhibit sparsity in their Jacobians. Many Fortran AD tools, such as ADIFOR [Bischof et al. 1996] and TAF [FastOpt 2003], utilise Fortran’s efficiency for array operations to calculate multiple derivatives. ADIFOR may also use the SparseLinC library [Bischof et al. 1996] to propagate sparse derivatives. Our aim was to emulate these capabilities in MATLAB. To do this the \texttt{derivvec} class was designed and implemented.

3.1 Design of the \texttt{derivvec} Class

The \texttt{derivvec} class was designed after studying features of the intrinsic MATLAB \texttt{double} and \texttt{sparse} classes. Key features of these intrinsic classes are:
—In MATLAB an array is a fundamental object. Whole array operations are optimised, and access to individual elements is relatively slow.
—The lowest rank arrays in MATLAB are rank 2 (i.e. matrices), and array indexing is column major, i.e., an increment of one in the first index alone refers to the next element as stored in memory (fast access times).
—MATLAB can handle arbitrary rank ($\geq 2$) arrays of class double.
—MATLAB’s sparse matrices are strictly matrices, not arrays, and hence of rank 2.

Consider a $D$ dimensional $n_1 \times n_2 \times \ldots \times n_D$ MATLAB array representing values, say $A(1:n_1,1:n_2,\ldots,1:n_D)$. A natural way to store the derivatives would be to append an extra dimension to give an array $DA(1:n_1,1:n_2,\ldots,1:n_D,nderivs)$. We would choose to append (rather than prepend) the dimension since then the $i^{th}$ directional derivative $DA(1:n_1,1:n_2,\ldots,1:n_D,i)$ is then readily available and contiguous in memory. As we shall see in Sections 3.2–3.4, this property is crucial in allowing use of high-level, efficient MATLAB matrix operations. We shall refer to this way of storing multiple directional derivatives as the external representation of the derivative vector, since we shall ensure that all functions external to the derivvec class may access the derivatives assuming this multi-dimensional array structure.

Unfortunately, with the simple approach sketched in the previous paragraph, we cannot seamlessly use MATLAB’s sparse matrices to store derivatives for arbitrary dimension arrays. To circumvent this problem we chose to have an internal representation of the derivative vectors in terms of an unrolled matrix. We then explicitly handle the interface of such objects and their interaction with arbitrary rank arrays. In order to do this we explicitly store the size and number of derivatives associated with a derivative object. Objects of the derivvec class therefore have the following components:

nderivs. number of derivatives,
shape. row vector $[n_1,n_2,\ldots,n_D]$ storing size of corresponding value array,
derivs. reshaped matrix of derivatives $=$ reshape(DA,[prod(shape) nderivs]).

In this way the derivatives derivs are stored as a two-dimensional full or sparse matrix. The use of full or sparse storage is determined by the user when they provide a full or sparse matrix as the second, derivative argument dx to the fmad constructor function of Figure 1. The derivative dx is passed to the derivvec constructor function to form the derivs component of the derivvec object. We now explain how representative arithmetic operations may be performed on derivvec objects.

3.2 Addition

Figure 5 contains the coding of the plus function of the derivvec class invoked when two derivvec objects, a and b, are added together to form a third c. We see that the first action of the plus function is a deep copy of the input argument a to the output c, ensuring that c is of class derivvec. In MATLAB addition must be between two arrays that either have an identical size, or else one of the arrays must be a scalar. Consequently, the local variables ssa and ssb are set to be the product of the shape component of a and b, respectively. If they have
function c=plus(a,b)
c=a; % deep copy of a
ssa=prod(a.shape);
sb=b.shape;
ssb=prod(sb);
if ssa==ssb % a,b No. elements equal so simply add
  c.derivs=a.derivs+b.derivs;
elseif ssa==1 % a is scalar
  c.shape=sb; % adopt shape of b
  % replicate a.derivs to have ssb rows
  if issparse(a.derivs)
    % use sparsity to replicate a.derivs
    [i,j,val]=find(a.derivs); % row vectors i,j,val
    nentry=length(j);
    nd=a.nderivs;
    i=(1:ssb)'; % create i for each row needed
    i=i(:,ones(1,nentry)); % replicate i for each entry
    j=j(pad,:);
    val=val(pad,:);
    c.derivs=sparse(i,j,val,ssb,nd)+b.derivs;
  else
    % replicate a ssb times before adding
    c.derivs=a.derivs(ones(1,ssb),:)+b.derivs;
  end
else
  % omitted for brevity
  .
  .
end

the same value \texttt{a.derivs} and \texttt{b.derivs} may be safely added together. If one of \texttt{ssa} or \texttt{ssb} take the value one the corresponding \texttt{fmad} operation must be the legitimate addition of a scalar to an array. Consider first \texttt{ssa==1} corresponding to \texttt{a} being the row vector of derivatives for a scalar \texttt{fmad} object. We must add this row vector to each row of \texttt{b.derivs}. Such an addition could be performed by a loop but might compromise performance. Instead we replicate \texttt{a.derivs} \texttt{ssb} times forming a matrix of the same dimensions as \texttt{b.derivs} before adding. As indicated in the code of Figure 5, if \texttt{a.derivs} is a sparse matrix we extract the \((i,j)\) indices and values of its entries, explicitly replicate them, and use them to form a sparse matrix that is added to \texttt{b.derivs}. If \texttt{a.derivs} is full a simpler indexing operation is used to perform the replication. A similar sequence of operations is performed if \texttt{b} corresponds to a scalar (omitted from Figure 5). There are no loops of any kind in the \texttt{plus} function of Figure 5, and the presence of any derivatives stored as \texttt{sparse} matrices is fully exploited. These properties are crucial to consistently achieving good performance.

The use of array operations rather than loops might be regarded as unnecessary since the release of MATLAB version 6.5 for which the just-in-time (JIT) accelerator capabilities often enable fast performance of operations on arrays performed in loops. This capability was not available during much of code development and,
function c=times(a,b)
if isa(b,'derivvec');
c=b; % pick out b as derivatives
mults=a; % pick out a as multipliers
else
  c=a; % pick out a as derivatives
  mults=b; % pick out b as multipliers
end
ssd=prod(c.shape);
sm=size(mults);
ssm=prod(sm);
mults=mults(:);
% make everything conformable
if ssd==ssm
  derivs=c.derivs;
  % check for sparsity
  if issparse(derivs)
    % only need multipliers corresponding to
    % entries of c.derivs
    [i,j]=find(derivs);
    nd=c.nderivs;
    c.derivs=sparse(i,j,mults(i),ssd,nd).*derivs;
  else
    c.derivs=mults(:,ones(1,c.nderivs)).*derivs;
  end
elseif ssd==1
  c.shape=sm;
  c.derivs=mults*c.derivs;
elseif ssm==1
  c.derivs=mults.*c.derivs;
end

since many of our users employ earlier versions of MATLAB, we do not wish to
rely on it. Note that ADMAT’s use of cell arrays prevents JIT acceleration under
MATLAB version 6.5.

3.3 Element-Wise Multiplication

As a second example of the use of high-level MATLAB matrix operations within
the functions of the derivvec class consider the element-wise multiplication of a
matrix with a derivvec class object as coded in the times function of Figure 6.
This function is invoked from fmad functions with just one of the inputs a or b of
derivvec class. First the function determines which argument is of derivvec class,
makes a deep copy of it for the function result c, unrolls the non-derivvec class
input, and stores it as the local column vector mults. Then local variables ssd
and ssm are used to store the number of elements corresponding to the derivvec
input and the number of elements in the multiplier mults, respectively. If ssd is
equal to ssm the multiplier is conformable with a single directional derivative and
so must be replicated, taking account of sparsity, to be of nderivs columns before
an element-wise multiplication calculates the c.derivs component. If ssd==1 the
derivvec input is recognised as corresponding to a scalar, and the shape of the
output c is given by that of the multiplier. Now c.derivs is given as a matrix with
\(i^{th}\) row given by the single row of the derivvec input multiplied by the \(i^{th}\) element

Fig. 6. The times function of the derivvec class

function b=subsref(a,s)
    b=a; % deep copy of a
    sa=a.shape;
    ssa=prod(sa); % get total number of elements for each directional derivative
    ind=reshape((1:ssa),sa); % form an indexing array
    newind=subsref(ind,s); % grab required indexes using subscripting
    s=size(newind); % use size of newind to get size of result
    b.shape=s;
    newind=newind(:); % reshape newind to column vector
    b.derivs=a.derivs(newind,:); % use indices to grab derivatives

Fig. 7. The subsref function of the derivvec class

of the unrolled mults. Such a matrix is easily obtained by the matrix-multiplication
of mults and c.derivs. Finally, if the multiplier is scalar ssm=1 we multiply all
rows of the input derivvec argument, currently stored as c.derivs, by the scalar.

3.4 Subscript Referencing

To facilitate subscript referencing for multiple directional derivatives in the fmad
class we must supply a derivvec class subscript referencing function so that the
fmad subsref function of Figure 4 operates correctly when y.deriv is of derivvec
class. At first sight, this is is somewhat problematic to code efficiently given the
internal, unrolled storage chosen for the derivatives. One approach would be to
reshape the internal representation of the derivatives to match the external repre-
sentation, append a ‘:’ to the array of indices (c.f. Section 2.1.4), and then perform
a subscript reference. This is undesirable since it precludes the use of sparse matrix
storage for the derivatives. Instead, as shown in Figure 7, we form an indexing array
corresponding to the elements of the external representation of DA and ignoring the
derivatives. This array is then acted upon by the intrinsic subscript referencing
function to return the required rows of the internal representation of the derivative
vector, which may then be trivially accessed. The coding of Figure 7 illustrates
the elegance of this approach with the use of high-level operations again ensuring
efficiency.

A similar strategy is used to code the transpose function, invoked as A.’, of the
derivvec class. First an indexing array is formed, it is then transposed, and the
resulting array used to index the rows of the internal representation of the input
array of derivatives to form the rows of the output array of derivatives.

3.5 Extracting Derivatives

Function getderivs of Section 2.1.5 can be used to extract derivatives from fmad
objects with derivatives stored internally as derivvec objects. The derivatives are
returned in the external representation of Section 3.1 with dimension one degree
higher than the value. Derivatives stored internally as (2-dimensional) sparse
matrices are first converted to full arrays since sparse arrays can only be 2-
dimensional. Since this would prevent users accessing the sparse data structure
necessary for efficiency in large problems a function getinternalderivs is provided
to return the matrix internal representation of an fmad object’s derivatives.

Having described both MAD’s fmad and derivvec classes we present a simple
function J=FmadFullJac(t,y0,N)
% uses full storage in fmad
y=fmad(y0,eye(2*N));% initialise y
dydt=f(t,y,N);% calculate F
J=getinternalderivs(dydt);% grab Jacobian

Fig. 8. Calculating the Jacobian of the Brusselator problem using full storage of derivatives.

example in Section 4 to demonstrate their use.

4. USING THE FMAD CLASS

In this section we describe how to use the fmad class to calculate Jacobians taking as an example the Brusselator test problem. We defer detailed discussion of the relative efficiencies of these techniques until Section 5.

The Brusselator ODE problem [Hairer and Wanner 1991] is supplied as a test case (file brussode.m) for the ode15s stiff differential equation solver in MATLAB [The MathWorks Inc. 2003, Section 14]. It corresponds to a method-of-lines discretisation of a 2-species reaction-diffusion equation on a 1-D mesh of \(N\) points, giving an ODE in standard form \(y' = f(t,y)\) with \(n = \text{length}(y) = 2N\). To facilitate the embedded quasi-Newton solver of the BDF-like ode15s, the sparse \(n \times n\) Jacobian \(Jf = \frac{\partial f}{\partial y}\) must be evaluated. The interface to the function is,

\[
\text{function } dydt = f(t,y,N)
\]

where \(t\) is the scalar time, \(y\) is the solution vector and \(N\) is the number of mesh points.

We now describe how to calculate the Jacobian of the Brusselator function using the fmad class with derivatives stored and manipulated using full, sparse or compressed matrix storage.

4.1 Full Storage

The use of full (dense) storage is illustrated by the function FmadFullJac of Figure 8. For \(y0\) supplied to FmadFullJac the vector \(y\) is initialised to be of fmad class with value given by \(y0\) and derivatives given by the \(2N \times 2N\) identity matrix \(\text{eye}(2\times N)\). Consequently element \(y(j)\) of \(y\) has directional derivative given by the \(j\)th column of \(\text{eye}(2\times N)\), that is the vector with entry one in position \(j\) and zeros elsewhere. Consequently the \(j\)th directional derivative corresponds to derivatives with respect to \(y(j)\). The function \(f\) is then called with the fmad object \(y\) as an argument. The overloaded operators and functions of Sections 2 and 3 then compute the function’s value and derivatives. The derivatives are then extracted from the function’s return value \(dydt\) using the fmad function getinternalderivs and are returned in the full storage matrix \(J\) such that the \(i\)th row of \(J\) comprises the \(j\)th derivatives and since the \(j\)th derivatives are with respect to \(y(j)\) we see that

\[
J(i,j) = \frac{\partial f_i}{\partial y_j}.
\]

4.2 Sparse Storage

The use of sparse storage is illustrated by the function FmadSparseJac of Figure 9. The only difference compared to the code of Figure 8 is that the derivatives are
Fig. 9. Calculating the Jacobian of the Brusselator problem using sparse storage of derivatives.

\begin{verbatim}
function J=FmadSparseJac(t,y0,N)
  % uses sparse storage in fmad
  y=fmad(y0,speye(2*N));% initialise y
dydt=f(t,y,N);% calculate F
  J=getinternalderivs(dydt);% grab Jacobian
\end{verbatim}

Fig. 10. Determining sparsity pattern, coloring, and seed matrix for compressed storage of derivatives.

\begin{verbatim}
sparsity_pattern=jpattern(N); % sparsity pattern
  color_groups=MADcolor(sparsity_pattern); % coloring
  seed=MADgetseed(sparsity_pattern,color_groups); % seed
\end{verbatim}

Fig. 11. Calculating the Jacobian of the Brusselator problem using compressed storage of derivatives.

\begin{verbatim}
function J=Fmadcmpjac(t,y0,N,...
  sparsity_pattern,color_groups,seed)
  % uses compressed fmad
  y=fmad(y0,seed);% initialise y
dydt=f(t,y,N);% calculate F
  Jcomp=getinternalderivs(dydt);% grab compressed Jacobian
  J=MADgetcompressedJac(Jcomp,...
    sparsity_pattern,color_groups);
\end{verbatim}

initialised using the MATLAB function `speye(2*N)`, which returns a sparse identity matrix. In the ensuing overloaded `fmad` calculations all derivatives are stored and manipulated as sparse matrices, and the extracted Jacobian \( J \) will be returned in sparse format.

4.3 Compressed Storage

Here a two stage process is used.

(1) The first stage, shown in Figure 10, involves determining information on the Jacobian’s sparsity pattern and how to perform the compressed Jacobian calculation. Provided the Jacobian’s sparsity pattern is fixed, or is safely overestimated, then this stage need only be performed once, and the information determined reused for multiple Jacobian calculations. For given problem size \( N \) the function `jpattern`, supplied by MATLAB within the file `brussode.m`, returns the Jacobian’s sparsity pattern. The sparsity pattern is a matrix of the same size as the Jacobian but with unit entry in position \((i, j)\) if \( \text{J}(i, j) \) is nonzero and zero otherwise. Given the sparsity pattern, the function `MADcolor` determines a group, or color, for each element of the function’s vector of inputs \( y \) such that those in the same group do not affect the same rows of the Jacobian. Like MATLAB’s `numjac` function [Shampine and Reichelt 1997], `MADcolor` uses the most effective of first-fit and first-fit after reverse column minimum degree orderings. The function `MADgetseed` uses the coloring to construct a seed matrix with \( \text{seed}(i, k) \) taking the value one if \( y(i) \) is in color group \( k \). Note that `sparsity_pattern` is an argument to `MADgetseed` solely as a device to supply the Jacobian size.

(2) The second stage of the calculation is shown in Figure 11. First the seed matrix is used to initialise the derivatives of \( y \). Then, after propagating \( y \) through the calculation of the function \( f \), the compressed Jacobian \( J_{\text{comp}} \) is extracted, and the MAD function `MADgetcompressedJac` is used, in conjunction with the
sparsity pattern and coloring, to return the Jacobian in sparse format.

Recently we have automated the use of the \texttt{fmad} class within ODE and optimisation solvers using high-level interface functions [Forth and Ketzscher 2004] and by directly coding \texttt{fmad} function calls into the solvers [Shampine et al. 2005; Forth and Edvall 2004].

In the following section we compare the performance of the \texttt{fmad} class with the existing ADMAT automatic differentiation package [Verma 1998b] and finite-differencing.

5. TEST CASES

In this section we present several test cases all performed using MATLAB version 6.5 on a PC running Windows XP Professional on a 3.0 GHz Pentium IV processor with 512 MB of RAM. Testing was also performed on a SUN Blade 1000 workstation running UNIX and a Pentium IV PC running Linux. Results on these two platforms are qualitatively similar to those presented here and, in particular, the rankings of different differentiation techniques are the same. Application of \texttt{Mad} to several boundary value problems may be found in Shampine et al. [2005]. Our \texttt{Mad} package is constantly being updated with new functionality; the version used here was that as of December 2004.

5.1 Polynomial Data Fitting

This problem concerns the calculation of the coefficients of the \(m\)-degree polynomial 
\[ p(x) = p_1 + p_2 x + p_2 x^2 + \ldots + p_m x^{m-1} \]
that best fits the points \((x_i, d_i), i = 1, \ldots, n\) in the least squares sense. This leads to the over-determined linear system \(Vp = d\), where \(V\) is the well-known Vandermonde matrix,

\[
V = \begin{bmatrix}
1 & x_1 & x_1^2 & \ldots & x_1^{m-1} \\
1 & x_2 & x_2^2 & \ldots & x_2^{m-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_n & x_n^2 & \ldots & x_n^{m-1}
\end{bmatrix}
\]

The problem of calculating the derivatives of the \(m\) coefficients \(p\) with respect to the \(n\) abscissas \(x\) has been considered previously [Bischof et al. 2002] via the hybrid source-transformation/overloaded MATLAB AD tool ADiMat. These authors provided a short MATLAB function to calculate \(p\) and showed that ADiMat-generated derivative code executed with a similar efficiency to one-sided finite-differencing. Though competitive for \(N < 100\), forward mode ADMAT was uncompetitive for larger \(n\).

Here we effectively repeat the calculation of [Bischof et al. 2002] but using the \texttt{fmad} class and also MATLAB’s \texttt{numjac} finite-difference Jacobian function. The \texttt{numjac} function approximates the Jacobian via one-sided finite-differencing with automatic adjustment of the step-size to ensure accuracy of the larger elements in each column of \(Jf\) [Shampine and Reichelt 1997]. In Table I we show the ratio of Jacobian to function CPU times for this problem; function timings may be found in Table VII of Appendix A. For \(n \geq 40\) we see that \texttt{fmad}(full) with full storage for derivatives has comparable performance to \texttt{numjac}. Using sparse derivative storage with \texttt{fmad}(sparse) improves performance further, outperforming \texttt{numjac} for \(n \geq 40\).
Table I. Ratio CPU($J_f$)/CPU($f$) of Jacobian to function CPU times for the Polynomial Data Fitting problem with $m = 4$. Jacobian and function calculations were timed over loops of 7680/n and 25600 evaluations respectively, and this process was repeated 10 times to give an average CPU time. Further information is given in Table VII of Appendix A.

<table>
<thead>
<tr>
<th>Method</th>
<th>10</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
<th>640</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>numjac</td>
<td>19.2</td>
<td>31.6</td>
<td>56.9</td>
<td>106.6</td>
<td>202.4</td>
<td>393.0</td>
<td>823.2</td>
<td>1528.8</td>
</tr>
<tr>
<td>fmad(full)</td>
<td>42.9</td>
<td>40.8</td>
<td>46.9</td>
<td>75.0</td>
<td>167.0</td>
<td>403.1</td>
<td>802.0</td>
<td>1704.2</td>
</tr>
<tr>
<td>fmad(sparse)</td>
<td>44.1</td>
<td>39.0</td>
<td>34.3</td>
<td>32.4</td>
<td>33.6</td>
<td>71.1</td>
<td>127.2</td>
<td>257.2</td>
</tr>
<tr>
<td>ADMAT(full)</td>
<td>44.1</td>
<td>60.4</td>
<td>97.8</td>
<td>175.3</td>
<td>888.9</td>
<td>7220.0</td>
<td>30399.3</td>
<td>128588.1</td>
</tr>
<tr>
<td>ADMAT(sparse)</td>
<td>47.6</td>
<td>63.0</td>
<td>94.3</td>
<td>150.9</td>
<td>265.9</td>
<td>623.4</td>
<td>922.6</td>
<td>1806.9</td>
</tr>
</tbody>
</table>

Table II. Ratio CPU($J_f$)/CPU($f$) of Jacobian to function CPU times for the Brusselator problem. Jacobian and function calculations were timed over loops of 1000 and 5000 evaluations respectively, and this process was repeated 10 times to give an average CPU time. Further information is given in Table VIII of Appendix A.

<table>
<thead>
<tr>
<th>Method</th>
<th>20</th>
<th>40</th>
<th>80</th>
<th>160</th>
<th>320</th>
<th>640</th>
<th>1280</th>
<th>2560</th>
</tr>
</thead>
<tbody>
<tr>
<td>numjac(comp,vect)</td>
<td>5.4</td>
<td>6.1</td>
<td>6.7</td>
<td>8.0</td>
<td>8.2</td>
<td>8.4</td>
<td>9.7</td>
<td>9.7</td>
</tr>
<tr>
<td>fmad(sparse)</td>
<td>71.8</td>
<td>72.7</td>
<td>67.1</td>
<td>60.2</td>
<td>51.2</td>
<td>41.7</td>
<td>43.1</td>
<td>35.8</td>
</tr>
<tr>
<td>fmad(comp)</td>
<td>64.8</td>
<td>64.2</td>
<td>54.8</td>
<td>46.8</td>
<td>35.7</td>
<td>25.1</td>
<td>19.8</td>
<td>15.2</td>
</tr>
<tr>
<td>ADMAT(comp)</td>
<td>101.4</td>
<td>98.5</td>
<td>84.0</td>
<td>70.1</td>
<td>51.4</td>
<td>35.1</td>
<td>25.9</td>
<td>18.1</td>
</tr>
</tbody>
</table>

and for $n \geq 160$ being five times more efficient. Although the Jacobian $\partial p/\partial x$ is full, sufficient intermediate values (in particular the Vandermonde matrix $V$ itself) have sparse derivatives that the use of sparsity is very beneficial. We see that use of ADMAT’s forward mode with full storage is reasonably efficient for $n = 10$, but its apparent quadratic growth of CPU time with $n$ [Bischof et al. 2002] makes it uncompetitive for all other $n$. Use of sparse derivatives in ADMAT is beneficial, but at best has comparable efficiency to fmad’s full storage.

As observed by Bischof et al. [2002], although $m = 4 \ll n$ for this problem, ADMAT’s reverse mode is less efficient than forward, e.g. for $n = 40$, CPU($J_f$)/CPU($f$) = 14241.

5.2 The Brusselator

For this test case, described in Section 4, the sparse $n \times n$ Jacobian $J_f = \partial f/\partial y$ must be evaluated. By default, MATLAB’s stiff ODE solver ode15s uses the MATLAB-supplied numjac function. If the Jacobian’s sparsity pattern is supplied, then numjac uses Jacobian row compression techniques [Shampine and Reichelt 1997], [Griewank 2000, Chap. 7]. For this example, Jacobian compression enables construction of the Jacobian from just four Jacobian-vector products which, by default, are approximated by one-sided finite differencing. If the user indicates that the function $f(t,y)$ is vectorizable, which is the case here, then the four extra function evaluations may be performed in a single call of the function $f(t,\mathbf{Y})$, where $\mathbf{Y}$ is a matrix whose columns are the perturbed $y$’s.

In Table II, we present the ratio of average Jacobian to average function CPU times CPU($J_f$)/CPU($f$) for various sparsity-exploiting Jacobian calculation techniques. The technique numjac(comp, vect), uses sparse finite differencing via Jaco-
Table III. ODE solution CPU time for the Brusselator problem. Each integration was performed in a loop 10 times, and this process was repeated 10 times to get an average CPU time.

<table>
<thead>
<tr>
<th>Method</th>
<th>CPU(ODE solve) for problem size n (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20  40  80  160  320  640  1280  2560</td>
</tr>
<tr>
<td>numjac(comp,vect)</td>
<td>0.07 0.08 0.11 0.15 0.24 0.42 0.93 2.49</td>
</tr>
<tr>
<td>fmad(sparse)</td>
<td>0.09 0.10 0.13 0.17 0.24 0.45 0.80 1.67</td>
</tr>
<tr>
<td>fmad(comp)</td>
<td>0.10 0.11 0.13 0.17 0.25 0.46 0.80 1.70</td>
</tr>
<tr>
<td>fmad(comp,recolor)</td>
<td>0.10 0.11 0.14 0.18 0.27 0.50 0.98 2.45</td>
</tr>
<tr>
<td>ADMAT(comp)</td>
<td>0.11 0.12 0.14 0.18 0.26 0.44 0.81 1.75</td>
</tr>
<tr>
<td>ADMAT(comp,recolor)</td>
<td>0.11 0.12 0.15 0.19 0.28 0.49 0.99 2.43</td>
</tr>
</tbody>
</table>

Table III indicates that compressed finite differencing is the most efficient Jacobian calculation technique, though for large values of n both compressed AD techniques (fmad(comp) and ADMAT(comp)) are only about two to three times slower.

In Table III, we show the effect of Jacobian technique on CPU times taken to solve the Brusselator ODE problem, as defined in [The MathWorks Inc. 2003, Section 14]. For all integrations, only two Jacobian evaluations are performed. For large n fmad(comp) and ADMAT(comp) outperform numjac(comp,vect), despite the fact that Table II indicates that they should not. This is because at the start of integration ode15s always re-computes the coloring required to use compression with numjac. If we force such a recomputation when using fmad(comp) and ADMAT(comp) (rows fmad(comp,recolor) and ADMAT(comp,recolor) of Table III), then we see the pre-eminence of numjac(comp,vect) over the AD compressed techniques restored. Note also that fmad(sparse) (which requires no coloring) is approximately as fast or outperforms numjac(comp,vect) for this problem.

5.3 Coleman & Verma’s Arrowhead Function

This function [Griewank 2000, Section 7.4], with \( x \rightarrow f(x) \) and \( x, f(x) \in \mathbb{R}^n \),

\[
\begin{align*}
  f_1 &= 2x_1^2 + \sum_{i=1}^{n} x_i^2 \\
  f_i &= x_1^2 + x_i^2, \quad i = 2, \ldots, n \end{align*}
\]

has sparse Jacobian with one full row, one full column and a full diagonal, e.g., for \( n = 7 \) the Jacobian has sparsity pattern,

\[
Jf(x) = \begin{bmatrix}
  & & & \bullet & \bullet & \bullet & \bullet \\
  & & \bullet & \bullet & \bullet & \bullet & \bullet \\
  & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
  \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet
\end{bmatrix}
\]

Table IV. Ratio CPU(Jf)/CPU(f) of Jacobian to function CPU times for the Arrowhead problem. Jacobian and function calculations were timed over loops of 500 and 500,000 evaluations respectively, and this process was repeated 10 times to give an average CPU time. Further information is given in Table IX of Appendix A.

<table>
<thead>
<tr>
<th>Method</th>
<th>CPU(Jf)/CPU(f) for problem size n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td>numjac(vect)</td>
<td>20.8</td>
</tr>
<tr>
<td>fmad(sparse)</td>
<td>90.6</td>
</tr>
<tr>
<td>ADMAT(sparse)</td>
<td>192.4</td>
</tr>
<tr>
<td>ADMIT</td>
<td>285.2</td>
</tr>
</tbody>
</table>

where the • symbols denote non-zero entries of the Jacobian. Such a sparsity pattern prohibits row or column compression techniques. An effective strategy is bi-coloring [Coleman and Verma 1996], [Griewank 2000, Section 7.4]: finding and using combinations of Jacobian-vector products evaluated via forward mode and vector-Jacobian products via reverse mode to evaluate the Jacobian. For the Arrowhead function two Jacobian-vector products can calculate the full column and diagonal, and one vector-Jacobian product calculates the full row. The bi-coloring technique is available in MATLAB via the ADMIT Toolbox [Coleman and Verma 2000].

In Table IV, we show the ratio of Jacobian to function CPU times for the Arrowhead function for various problem sizes n and several Jacobian evaluation techniques. We use Verma’s original function coding for this problem with the exception of row numjac(vect), where we use a slightly modified version to allow for vectorized finite differencing. Although most efficient for small n, numjac(vect)’s inability to exploit sparsity makes it two orders of magnitude slower than fmad(sparse) and ADMIT for the largest n. We see that fmad(sparse) clearly out-performs ADMAT(sparse) and even the sophisticated ADMIT technique.

The results for this problem contradict the conventional wisdom [Griewank 2000, p137] that an effective compression should outperform sparse forward mode due to the latter’s overheads arising from manipulating sparse data structures [Gilbert et al. 1992]. In Appendix B we show that fmad’s sparse mode requires just 7n + 2 floating point operations to propagate derivative information compared to the 12n + 3 floating point operations required by ADMIT. So sparse forward mode may use fewer flops than a bi-coloring approach (it always uses fewer flops than compressed forward mode [Griewank and Reese 1991]). Also ADMIT itself has substantial overheads: the computation and necessary values of variables must be taped [Griewank 2000, Chap. 3], the tape traversed in both forward and reverse directions to propagate derivatives and adjoints, and the Jacobian must be assembled from the propagated derivatives and adjoints. For ADMIT there is also the one-off effort of determining the sparsity pattern and a suitable coloring, although this is not included in the timings of Table IV. We note that previous articles on bi-coloring [Coleman and Verma 1998b; 2000] have demonstrated that it is possible to get a good coloring for a given sparsity pattern but contain no timings of Jacobian calculation via ADMAT; the single timed example in Coleman and Verma [1998b] uses an ADOL-C [Griewank et al. 1996] tape for the Jacobian calculation.
Table V. Ratio CPU(Jf)/CPU(f) of Jacobian to function CPU times for Large-Scale Examples from MATLAB Optimization Toolbox. Jacobian and function calculations were timed over loops of 10 and 1000 evaluations respectively, and this process repeated 10 times to give average CPU times. Further information on these problems is given in Table X of Appendix A.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Hand-coded</th>
<th>CPU(Jf)/CPU(f) for given technique</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sfd(nls)</td>
<td>fmad(sparse)</td>
</tr>
<tr>
<td>nlsf1a</td>
<td>5.1</td>
<td>43.2</td>
</tr>
<tr>
<td>brownf</td>
<td>3.9</td>
<td>1123.2</td>
</tr>
<tr>
<td>brownfg</td>
<td>3.8</td>
<td>6.4</td>
</tr>
<tr>
<td>tbroyf</td>
<td>4.7</td>
<td>898.5</td>
</tr>
<tr>
<td>tbroyfg</td>
<td>-</td>
<td>14.6</td>
</tr>
</tbody>
</table>

5.4 Large-Scale Examples from MATLAB Optimization Toolbox

In Table V, we give Jacobian to function CPU time ratios for test problems of the MATLAB Optimization Toolbox [The Mathworks Inc. 2003]; we include two Hessian calculations regarding them as the Jacobian of hand-coded gradients. For problem brownfg (obtaining the Hessian of the gradient of the Brown problem), the MATLAB supplied hand-coded Hessian was incorrect. With the aid of MATLAB’s Symbolic Toolbox, it took about half a day to correctly derive Hessian code (denoted Hand-coded in Table V), while all the AD techniques were coded in just a few minutes. We made just two changes to the supplied MATLAB code to enable automatic differentiation. For the Brown brownfg problem some two-dimensional array indexing \(x(i+1,1)\) was changed to one-dimensional \(x(i+1)\) to enable use of ADMAT’s sparse forward mode, and in tbroyfg initialisation of a vector \(z\) was changed from,

\[n = \text{length}(x); \ j = 1:(n/2); \ z = \text{zeros}(\text{length}(j),1);\]

to

\[n = \text{length}(x); \ j = 1:(n/2); \ z = \text{zeros}(n/2,1);\]

to force \(z\) to be of fmad class, c.f., Section 2.1.4.

The one technique used in Table V we have not yet met is sfd(nls), referring to MATLAB’s Optimization Toolbox sparse finite-difference functions sfdnls for Jacobians/Gradients (using function evaluations) and sfd for Hessians (using gradient evaluations). Both use compression if the Jacobian/Hessian sparsity pattern is supplied but, unlike numjac, neither take advantage of vectorization.

From Table V, we see that in all cases for which hand-coding is available it out-performs all other techniques. For the sparse Jacobian (nlsf1a) and Hessian calculations (brownfg, tbroyfg) we see that fmad(comp), MAD’s forward mode AD with compression, out-performs both compressed finite-differencing (sfd(nls)) and ADMAT’s forward mode with compression (ADMAT(comp)). For the gradient calculations (brownf, tbroyf), then after hand-coding fmad(sparse) is most efficient and out-performs ADMAT’s reverse mode. It is also 50 to 100 times faster than finite-differencing since there is sparsity in the gradient calculation (the functions are partially value separable [Griewank 2000, p206]), but the function gradient itself is dense (preventing sfd(nls) using compression).

Table VI shows the effect of varying the derivative calculation technique on the
Table VI. CPU time (CPU(calculated technique)) for optimization of the Large-Scale Examples from the MATLAB Optimization Toolbox. For each problem, its type, the MATLAB optimization function used and how derivatives are calculated is specified. Each optimization was timed over a loop of 10 evaluations, and this process repeated 10 times giving an average CPU over 100 evaluations.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Hand-coded</th>
<th>sfd(nls)</th>
<th>fmad-(sparse)</th>
<th>fmad-(comp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nlsf1a</td>
<td>Nonlinear solve</td>
<td>0.46</td>
<td>0.79</td>
<td>0.58</td>
</tr>
<tr>
<td>brownf</td>
<td>Minimisation</td>
<td>1.19</td>
<td>-</td>
<td>2.59</td>
</tr>
<tr>
<td>brownfg</td>
<td>Minimisation</td>
<td>0.62</td>
<td>1.15</td>
<td>1.92</td>
</tr>
<tr>
<td>tbroyf</td>
<td>Constrained minimisation</td>
<td>1.22</td>
<td>-</td>
<td>4.71</td>
</tr>
<tr>
<td>tbroyfg</td>
<td>Constrained minimisation</td>
<td>-</td>
<td>1.30</td>
<td>1.35</td>
</tr>
</tbody>
</table>

Run-times of various optimization problems. Hand-coding gives fastest overall optimization times. For the sparse Jacobian/Hessian cases (nlsf1a, brownfg, tbroyfg) fmad's compressed forward mode (fmad(comp)) outperforms sparse-finite differencing (sfd(nls)). For the two gradient cases (brownf, tbroyf) fmad's sparse forward mode (fmad(sparse)) gives very acceptable performance, far better than would be expected from finite-differencing (c.f., sfd(nls) in Table V).

6. CONCLUSIONS

Sections 2 and 3 of this paper detail the implementation of forward mode AD for first derivatives in the MAD package. A major feature of our fmad class for first derivatives is that it uses object components of intrinsic classes double or sparse for single directional derivatives and our derivvec class for multiple directional derivatives. Internal to the derivvec class multiple directional derivatives are stored as matrices (2-D arrays) using MATLAB's intrinsic full or sparse matrix classes. By making careful use of MATLAB's high-level matrix operations we have heavily optimised functions of the derivvec class. As seen in Section 4, use of the fmad class is straightforward.

The results of Section 5 demonstrate the effectiveness of our approach. For the dense Jacobian polynomial data fitting case of Section 5.1, although the finite-differencing of numjac is most efficient for small problem size \( n \), fmad's sparse forward mode is five times faster than numjac for large \( n \). Compressed vectorized finite-differencing outperforms MAD in evaluating Jacobians for the Brusselator problem of Section 5.2, although when solving the associated stiff ODE fmad's sparse mode outperforms numjac for large \( n \) since it does not need to calculate an expensive coloring. For small \( n \), numjac is the fastest technique for the arrowhead Jacobian of Section 5.3, but fmad(sparse) is 450 times faster for \( n = 1280 \). For the three sparse Jacobian/Hessian problems from the MATLAB Optimization Toolbox of Section 5.4 fmad's compressed mode just outperforms the Toolbox's sparse-finite differencing functions sfd/sfdnls. For the two gradient calculations fmad(sparse) is some 50 and 100 times faster than finite-differencing. When these techniques
were applied to the five optimization calculations of Table VI then in three cases an \texttt{fmad} technique was seen to outperform finite-differencing, and for the other two finite-differencing was so expensive it was not even tried (c.f., Table V)!

In all the test problems considered one of either \texttt{fmad}(sparse) or \texttt{fmad}(comp) outperformed comparable, or even more sophisticated, techniques (reverse mode, bi-coloring) from the ADMAT/ADMIT package. For large numbers of directional derivatives ADMAT’s forward mode performs poorly compared to \texttt{fmad} because its derivative manipulation operations use loops over each directional derivative. This is not as efficient as the matrix operations of the \texttt{derivvec} class and gives \texttt{fmad} a clear superiority in efficiency. When only a small number of directional derivatives are required, for example when Jacobian compression is used, this superiority of \texttt{fmad} is reduced. Performance may have been further improved because in \texttt{fmad} we never test to see if an \texttt{fmad} object has an empty derivative component whereas ADMAT does (see Section 2.1.4); also \texttt{fmad}’s runtime switching between code for one or multiple directional derivatives is performed by the MATLAB system based on the class (\texttt{double} or \texttt{derivvec}) of the derivatives rather than ADMAT’s use of branching. Additionally ADMAT may only use MATLAB’s sparse matrix class for storing multiple directional derivatives of vector objects and even then ADMAT’s internal use of loops degrades performance. The \texttt{derivvec} class enables \texttt{fmad} to use such sparse matrix storage for arbitrary dimension array objects with excellent efficiency. This is particularly apparent for partially separable functions such as \texttt{brownf} and \texttt{tbroyf} of Table V, and even the Arrowhead problem of Table IV for which previous authors [Coleman and Verma 1998b; 2000] have suggested that more sophisticated algorithms such as bi-coloring are needed.

The capabilities of Mad are currently being extended in three areas.

\textbf{Source Transformation.} For compiled languages source-transformation AD tools tend to produce differentiated code that out-performs its operator overloaded counterpart [Tadjouddine et al. 2001]. When differentiating MATLAB code containing array operations, as for the examples of Section 5, we have observed good performance as the problem size increases. We are now developing a source-transformation tool for MATLAB which, in preliminary tests, has demonstrated improved performance over the \texttt{fmad} class, particularly for smaller problems [Kharche 2004]. Some work has already been done in this area [Vehreschild 2001; Bischof et al. 2003; Bischof et al. 2002].

\textbf{Reverse Mode.} We are presently implementing the reverse mode of AD for first derivatives using a taping approach [Griewank 2000, Chapter 3]. Our implementation reuses the \texttt{derivvec} class for storing and propagating multiple adjoints.

\textbf{Automatic Sparsity Detection.} Results from Sections 5.2 and 5.4 demonstrate the effectiveness of Jacobian compression which requires a safe over-estimate of the Jacobian sparsity pattern. We are working on this sparsity estimation problem and aim to provide more robust and efficient capabilities than those of Verma [1998b].

In the longer term we hope to include elimination AD based approaches [Griewank 2000, Sections 8.1-8.2], [Naumann 1999; 2004] into Mad, since for Jacobian calculations it has theoretical efficiency advantages over conventional forward and reverse modes and may be implemented in the source-transformation framework [Forth et al. 2004].
Finally, MAD’s performance and reliability is sufficient for it to be distributed commercially (after a free evaluation period a small license fee is payable) [Forth and Edvall 2004] for both stand-alone use and in-conjunction with the TOMLAB optimisation package [Holmström and Edvall 2004; Holmström et al. 2004]. Several TOMLAB users, both academic and industrial, together with colleagues at Cranfield University have used MAD to differentiate involved MATLAB functions for applications such as race car trajectory optimization [Bradshaw 2004] and nonlinear response surface fitting [Ringrose and Forth 2004; 2004]. With just one exception, whenever fmad failed to compute correct derivatives and the user submitted a bug report we have either extended fmad to deal with the user’s code or suggested a simple change to the user’s code (e.g., ensuring arrays are of fmad class before subscript assignments as described in Section 2.1.4, or removing non-differentiable branches such as if abs(x)<=1e-6; x=0; end) to enable differentiation. The exception was a case with one million independent variables and for which reverse mode would appear necessary.

APPENDIX

A. FUNCTION CPU TIMES

Tables VII to IX give function CPU times for the test cases of Sections 5.1 to 5.3. As well as function CPU times, Table X gives problem types, sizes and Jacobian information for the Optimization Toolbox problems of Section 5.4.

B. COMPLEXITY ANALYSIS FOR THE ARROWHEAD EXAMPLE

Coleman and Verma’s [1996] coding for the arrowhead example of Section 5.3 is shown in Figure 12. For the purposes of analysis we consider an evaluation procedure [Griewank 2000, Chap. 2] of statements with scalar left hand-sides and right-hand sides that could be written as a single MATLAB statement. Such an evaluation procedure for the arrowhead problem is given in the first column of Table XI. Following the notation of [Forth et al. 2004] we see we have n independent variables v_i, i = 1, . . . , n, m = n dependent variables v_i, i = 2n + 4, . . . , 3n + 3 and p = n + 3 intermediate variables v_i, i = n + 1, . . . , 2n + 3. The second column of Table XI gives the local derivatives c_{i,j} = \partial v_i / \partial v_j for each statement. We see that we have N_{c=1} = 2n + 2 trivial local derivatives with value |c_{i,j}| = 1, and N_{c\neq1} = 2n + 1 nontrivial local derivatives with |c_{i,j}| \neq 1, 0.

Forth et al. [2004] show that, provided local derivatives c_{i,j} have already been determined, the cost in floating point operations of propagating q directional deriv-
Problem Type 

- Local derivatives
- Hessian from gradient
- Gradient from function

Function Jacobian
Hessian from gradient
Gradient from function

<table>
<thead>
<tr>
<th>Problem</th>
<th>Problem Type</th>
<th>(m, n)</th>
<th>n</th>
<th>p</th>
<th>CPU(f)(m,s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsfia</td>
<td>Function Jacobian</td>
<td>(1000,1000)</td>
<td>3</td>
<td>3</td>
<td>0.35</td>
</tr>
<tr>
<td>brownf</td>
<td>gradient from function</td>
<td>(1,1000)</td>
<td>1000</td>
<td>1000</td>
<td>1.57</td>
</tr>
<tr>
<td>brownfg</td>
<td>Hessian from gradient</td>
<td>(1000,1000)</td>
<td>3</td>
<td>3</td>
<td>5.74</td>
</tr>
<tr>
<td>tbrof</td>
<td>gradient from function</td>
<td>(1,1000)</td>
<td>800</td>
<td>800</td>
<td>1.16</td>
</tr>
<tr>
<td>tbrofg</td>
<td>Hessian from gradient</td>
<td>(800,800)</td>
<td>6</td>
<td>8</td>
<td>4.27</td>
</tr>
</tbody>
</table>

Fig. 12. Coding of the $\mathbb{R}^n \rightarrow \mathbb{R}^m$ arrowhead function.

Table XI. Evaluation trace for the $\mathbb{R}^n \rightarrow \mathbb{R}^m$ arrowhead example

<table>
<thead>
<tr>
<th>operation</th>
<th>local derivatives</th>
<th>sparsity pattern $\chi_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_i = z_j, i = 1, \ldots, n$</td>
<td>$v_i = v_{i-1} * v_{i+1}, i = n+1, \ldots, 2n$</td>
<td>$\chi_i = e_i$</td>
</tr>
<tr>
<td>$v_2_{n+1} = \sum_{j=1}^n v_j * v_j$</td>
<td>$\sum_{j=1}^n v_j * v_j$</td>
<td>$\chi_i = e_i$</td>
</tr>
<tr>
<td>$v_2_{n+2} = v_{n+1} + v_{n+1}$</td>
<td>$v_{n+2} = v_{n+1} + v_{n+1}$</td>
<td>$\chi_i = e_i$</td>
</tr>
<tr>
<td>$v_2_{n+3} = v_2 + v_1$</td>
<td>$v_{n+3} = v_2 + v_1$</td>
<td>$\chi_i = e_i$</td>
</tr>
<tr>
<td>$v_2_{n+4} = v_{2n+2} + v_{n+3}$</td>
<td>$v_{n+4} = v_{2n+2} + v_{n+3}$</td>
<td>$\chi_i = e_i$</td>
</tr>
<tr>
<td>$v_2_{n+5} = v_{n+5} + v_{n+3}, i = 2n+5, \ldots, 3n+3$</td>
<td>$v_{n+5} = v_{n+5} + v_{n+3}, i = 2n+5, \ldots, 3n+3$</td>
<td>$\chi_i = e_i$</td>
</tr>
</tbody>
</table>

Now consider the case of forward propagation of directional derivatives stored as sparse vectors as used by `fmad`’s sparse forward mode. The sparsity pattern of each variable in the calculation is given in the third column of Table XI to assist in determining floating point operations counts. We start with derivatives for the first n variables initialised to rows of the n x n identity matrix, 

$$\nabla v_i = e_i, i = 1, \ldots, n.$$ 

Now we consider in turn each set of operations given in Table XI.

- For $i = n+1, \ldots, 2n$ we have $\nabla v_i = c_{i,n} \nabla v_{i-1}$ for a cost of n multiplications since the $\nabla v_{i-1}$ correspond to derivatives of the independents, each of which has only one entry.

- For $i = 2n + 1$ we have $\nabla v_{2n+1} = \sum_{j=1}^n c_{2n+1, j} \nabla v_j$ for a cost of n multiplications, since each $\nabla v_j$ corresponds to derivatives of the independents which have only one nonzero entry, and n additions as each multiplied $\nabla v_j$ is added to a dense working vector [Gilbert et al. 1992]. The resulting $\nabla v_{2n+1}$ is full.

- For $i = 2n + 2$ we have $\nabla v_{2n+2} = \nabla v_{n+1} + \nabla v_{2n+1}$ which, since $\nabla v_{2n+1}$ is full but $\nabla v_{n+1}$ has one entry, incurs $(n+1)$ additions as they are accumulated in the working vector. The resulting $\nabla v_{2n+2}$ is full.
Thus sparse propagation of derivatives requires just $7n$ floating point operations.
Efficient Forward Mode AD in MATLAB


ACKNOWLEDGMENTS

The author would like to thank colleagues: Robert Ketzscher for valuable work performed optimizing Mad [Shampine et al. 2005]; and Venkat Sastry, John Pryce and Yi Cao for helpful suggestions regarding MATLAB performance optimization.

Received May 2004; revised May 2005; accepted August 2005.