European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS 2004
P. Neittaanmäki, T. Rossi, S. Korotov, E. Oñate, J. Périaux, and D. Knörzer (eds.)
Jyväskylä, 24–28 July 2004
ISBN 951-39-1869-6

HIGH-LEVEL INTERFACES FOR THE MAD (MATLAB AUTOMATIC DIFFERENTIATION) PACKAGE

Shaun A. Forth and Robert Ketzscher

Applied Mathematics and Operational Research Group,
Cranfield University (Shrivenham Campus), Swindon, SN6 8LA, UK
e-mails: S.A.Forth@cranfield.ac.uk, R.Ketzscher@cranfield.ac.uk
web page: http://www.rmcs.cranfield.ac.uk/amor

Key words: Automatic Differentiation, Numerical Optimization, Numerical Solution of Stiff ODEs

Abstract. *Presently, the MAD Automatic Differentiation package for matlab comprises an overloaded implementation of forward mode AD via the* `fmad` *class. A key design feature of the* `fmad` *class is a separation of the storage and manipulation of directional derivatives into a separate* `derivvec` *class. Within the* `derivvec` *class, directional derivatives are stored as matrices (2-D arrays) allowing for the use of either full or sparse matrix storage. All manipulation of directional derivatives is performed using high-level matrix operations - thus assuring efficiency. In this paper: we briefly review implementation of the fmad class; we then present our implementation of high-level interfaces allowing users to utilise MAD in conjunction with stiff ODE solvers and numerical optimization routines; we then demonstrate the ease and utility of this approach via several examples; we conclude with a road-map for future developments.*

1 Introduction

Automatic Differentiation (AD) enables the calculation of partial derivatives of functions defined by computer code to an accuracy commensurate with floating point round-off [5]. Well-developed AD packages exist for codes written in Fortran, C and C++, for which details may be found at the web-site `www.autodiff.org`. Until recently only the operator overloading ADMAT tool [13, 2] was available for MATLAB code. Since then a hybrid source-transformation/operator overloading approach has been investigated [12, 1].

Our own overloaded tool MAD (MATLAB Automatic Differentiation) has been in development for several years [10, 3] and is now commercially available via the TOMLAB company [4]. In [11] we demonstrated how, in conjunction with the designer of a collocation-based boundary-value solver, the use of AD in MATLAB could be *hidden* from the application programmer. In [11] the choice of the standard AD algorithm was pre-determined. Small scale Jacobians of insufficient size to warrant sparsity-exploitation demanded use of forward mode with dense storage of derivatives. In other applications more sophistication is demanded and a problem-dependent choice of AD technique is required. For example, in a method-of-lines discretization of a PDE, should dynamic propagation of derivatives using sparse-matrix storage be preferred to Jacobian compression? In the authors' opinion, if Automatic Differentiation is to be truly automatic, then such choices should themselves be automated and hidden from the application programmer. In this paper we present an approach for doing so in conjunction with MATLAB's stiff ODE and optimization solvers and utilizing the MAD tool.

In Section 2 of this paper we review implementation of forward mode AD in MATLAB using MAD[3]. In Section 3 we describe the implementation of high-level interfaces for making use of MAD from within standard MATLAB routines for numerical optimization and the solution of stiff differential equations. Section 4 presents test cases to demonstrate the utility and efficiency of this approach. Section 5 presents a road-map for MAD's future and concludes.

2 MAD - efficient forward mode AD through overloading

In MATLAB there are a variety of intrinsic classes with associated functions and operations. We use the object-oriented programming features of MATLAB to introduce two new classes `fmad` and `derivvec` [3]. Whenever MATLAB encounters objects of this class, for example when two such objects are matrix-multiplied, it will not use the standard `times` function designed for objects of class `double`, but instead will use the `times` functions defined in our new classes.

The class `fmad` is designed to store an array together with its directional derivatives. These derivatives are then usually of `derivvec` class.

2.1 The `fmad` class

The purpose of the `fmad` class is to deal with objects containing both function values and derivatives. To create an object of such class, the `fmad` constructor is used. Its syntax is simply `x=fmad(value,derivatives)`, where the first argument `value` can either be a scalar or a (possibly multi-dimensional) array and the second argument `derivatives` provides the derivatives in an array of the appropriate form. If there are n_d directional derivatives to be stored, then for a `value` array of dimensions $[i_1, i_2, \ldots, i_n]$ the supplied `derivatives` have to be `reshape`-able[1] to dimensions $[i_1 \times i_2 \times \ldots \times i_n, n_d]$ with each column taken as one of the n_d directional derivatives of the value array. Within the constructor a structure is defined with two components containing the two parameters, i.e. `x.value=value` and `x.derivs=derivatives`.

To enable the use of its objects within MATLAB, the `fmad` class needs to provide a wide range of functions. When encountering an expression such as `z=x*y`, MATLAB will use the default `times` operation if both arguments `x` and `y` are of class `double`. However, if one of them is of different class, such as `fmad`, it will look for an implementation of the `times` operation within that class.

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

Figure 1: Implementation of the `times` function within `fmad`

Figure 1 shows the implementation of the `times` function in `fmad`. Note that in MATLAB "`...`" denotes continuation and `x.value` denotes the value component of the object `x`. Through the MATLAB intrinsic `isa` the classes of the function arguments are tested. If both are of class `fmad` we have to use the chain rule to obtain value and derivative components as illustrated. Otherwise, if one of the two parameters is not of class `fmad`, and hence does not contain derivative information, then it multiplies the value and derivative components of the `fmad` argument.

Once a user defines an `fmad` object somewhere in the code, most function calls involving that object will result in `fmad` objects which will contain both values and derivatives. Similar to the `times` example, most MATLAB functions have been coded for `fmad` arguments. Thus, even for complicated codes, by provision of MATLAB intrinsics within `fmad` the user can obtain numerically exact derivatives.

Rather than worrying about the internal structure of the storage of `fmad` objects, the

[1] In MATLAB the `reshape` intrinsic is used as `B=reshape(A,[i1,i2,...,in])` to return `B` as a matrix of dimensions `[i1,i2,...,in]` whose elements are those of `A` in array element (column-wise) order.

user can use the two functions `getvalue` and `getderivs` to retrieve value and derivative information. For example, for the function $y = x_1 x_2^2$ with $\mathbf{x} = [2\ 3]$, we can calculate $\partial y/\partial x_1$ by

```
x=fmad([2 3],[1 0]);
y=x(1)*x(2)^2;
yval=getvalue(y)
dydx1=getderivs(y)
```

which correctly returns a function value of `yval=18` and a derivative value `dydx1=9`. Hidden from the user, the `fmad` implementations of the `times` and the `power` functions were used.

We now examine how MAD deals with the storage of the derivative information within the `derivvec` class.

2.2 The `derivvec` class

For more than one directional derivative, the derivative part of an `fmad` object is stored in a component of `derivvec` class. If an array `A` of dimensions $[i_1, i_2, \ldots, i_n]$ is passed to `fmad` and there are n_d directional derivatives stored in array `DA`, then ideally the user could refer to these derivatives by simply adding another index, i.e. `DA(k1,k2,...,kn,i)` should return the i-th directional derivative of the element $A(k_1, k_2, \ldots, k_n)$.

The function `getderivs` is written for this purpose. After executing the code

```
x=fmad([2 3],eye(2));
A=[x(1) x(2); x(1)*x(2) x(1)*x(2)^2];
DA=getderivs(A)
```

`DA` contains all partial derivatives of `A` as required.

The `derivvec` class is coded such that within the `fmad` class we may code as if the derivative component of `fmad` objects is of the same dimensions as the value component. However, internal to the `derivvec` class the derivatives are `reshape`d to a (2-dimensional) matrix with each directional derivative stored as one column. This allows us to take advantage of MATLAB's optimised matrix operations and use of intrinsic sparse matrices. For this example the derivatives of `A` returned in internal storage form are

```
getinternalderivs(A)
ans =
     1     0
     3     2
     0     1
     9    12
```

where we see that the first column contains derivatives with respect to x_1 and the second column with respect to x_2.

3 High-Level Interfaces

Derivatives are frequently required to be supplied by users to numerical software. For example, MATLAB's variable order stiff ODE solver `ode15s` requires the user to provide a vector-valued function $\mathbf{f}(t, \mathbf{y})$ to define the ODE $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$. It is also advantageous for the user to supply a MATLAB function to calculate the Jacobian, $\mathbf{Jf} = [\partial f_i/\partial y_j]$, for use in `ode15s`' embedded quasi-Newton solve. If the Jacobian is sparse, and the user cannot supply the Jacobian code, then they should supply the sparsity pattern of the Jacobian. Of course, using AD we can calculate the Jacobian, taking advantage of sparsity. As a second example, consider the constrained optimization problem, $\min y = f(\mathbf{x})$ such that $\mathbf{C}(\mathbf{x}) \leq 0$ and $\mathbf{c}(\mathbf{x}) = 0$. Derivatives are required for gradient descent algorithms and their accuracy is crucial in assessing convergence via the Karush-Kuhn-Tucker conditions [9]. Consequently, in the `fmincon` routine of the MATLAB Optimization Toolbox [6], users are advised to supply functions to calculate the gradient of the objective function $\nabla f = [\partial f(\mathbf{x})/\partial x_j]$ and the Jacobians of the constraint functions $\mathbf{JC} = [\partial C_i/\partial x_j]$ and $\mathbf{Jc} = [\partial c/\partial x_j]$.

It is important to realise that within a single ODE solve or optimisation calculation, gradients and/or Jacobians are required many times. In particular for optimisation, the gradient, and if constraints are present their Jacobians, **must** be evaluated at **each** iteration. In stiff ODE solution, strategies are used to minimise the number of times the Jacobian is required, but even so several such evaluations might be performed. Consequently it might be possible to automate the choice of AD algorithm used and, if techniques cannot be rejected out of-hand (e.g. due to lack of sparsity), use timings of techniques to choose the most efficient. The overhead of this strategy will be felt only in the first few Jacobian evaluations, thereafter the most efficient method shall be used.

In order to facilitate such use of AD in numerical software we have written a generic Jacobian utility function `MADJacInternal` which enables the calculation of the Jacobians of an arbitrary number of outputs with respect to arbitrary inputs of a specified function. Information about the Jacobian calculation is passed to `MADJacInternal` and retrieved back. In so doing, repeated calculations allow it to build up sufficient data to choose an efficient AD technique. High-level generic functions are then provided for applications such as ODE solution and numerical optimization, which may then call `MADJacInternal`, and consequently make use of the most appropriate AD algorithm provided by MAD.

We now describe `MADJacInternal` in some detail before describing interfaces for ODEs and optimization.

3.1 `MADJacInternal`

The task for `MADJacInternal` is, given the following:

1. A function whose interface is,

```
[y1,y2,...,yNout] = f(x1,x2,...,xNin)
```

2. A list `ActiveIndependents =[i1,i2,...,iN]` of independent variables,
   e.g. `ActiveIndependents=[1 3]` indicates we need Jacobians with respect to `x1` and `x3`.

3. A list `ActiveDependents =[j1,j2,...,jM]` of dependent variables,
   e.g. `ActiveDependents=[2 4]` indicates we need Jacobians of outputs `y2` and `y4`.

Then `MADJacInternal` should calculate the required Jacobians. For the lists `ActiveIndependents=[1 3]` and `ActiveDependents=[2 4]` we would require

$$\texttt{Dy2Dx1} = \frac{\partial \texttt{y2}}{\partial \texttt{x1}}\,, \texttt{Dy2Dx3} = \frac{\partial \texttt{y2}}{\partial \texttt{x3}},\ \texttt{Dy4Dx1} = \frac{\partial \texttt{y4}}{\partial \texttt{x1}} \text{ and } \texttt{Dy4Dx3} = \frac{\partial \texttt{y4}}{\partial \texttt{x3}},$$

where the `x`'s or `y`'s can be of arbitrary size. Optionally we might also require the function values themselves `y1,y2,...,yNout`.

The interface to `MADJacInternal` is therefore of the form,

```
[MADobj,Dyj1/Dxi1,Dyj1/Dxi2,...,Dyj1/DxiN,Dyj2/Dxi1,...,DyjM/DxiN]...
  = MADJacInternal(MADobj,Mode,Nout,ActiveIndependents,...
    ActiveDependents,x1,x2,...,xNin)                          (1)
```

when the string `Mode='J'` for Jacobian calculation, and

```
[MADobj,y1,y2,...,yNout,...
Dyj1/Dxi1,Dyj1/Dxi2,...,Dyj1/DxiN,Dyj2/Dxi1,...,DyjM/DxiN]...
  = MADJacInternal(MADobj,Mode,Nout,ActiveIndependents,...
    ActiveDependents,x1,x2,...,xNin)                          (2)
```

when `Mode='FJ'` for function and Jacobian calculation. `MADJacInternal` copes with an arbitrary number of function arguments `x1,x2,...,xNin` through use of the intrinsic `varargin` facility so that within `MADJacInternal` the arguments are available as a cell array `x{1},x{2},...,x{Nin}`. The only argument in the above interfaces yet to be described is the structure `MADobj` which is used to store information concerning the Jacobian calculation.

3.1.1 The `MADobj` Structure

We have seen how the `MADJacInternal` function will make use of an argument `MADobj` to store Jacobian calculation information. At the time of writing `MADobj` has the following components.

`func_handle` - function handle to `f`.

`handle_info` - function handle information, e.g. full path to file containing function.

`n` - sum of number of elements in all independent variables.

`m` - sum of number of elements in all dependent variables.

`use_ad_fwd_full` - takes value 1 if forward mode with full storage should be used, and value 0 if it should not.

`ad_fwd_full_time` - CPU time for forward mode with full storage.

`use_ad_fwd_sparse` - takes value 1 if forward mode with sparse storage should be used, and value 0 if it should not.

`ad_fwd_sparse_time` - CPU time for forward mode with sparse storage.

`Sparsity_Pattern` - Jacobian sparsity pattern.

`use_ad_fwd_compressed` - takes value 1 if forward mode with compressed storage should be used, and value 0 if it should not.

`ad_fwd_compressed_time` - CPU time for forward mode with compressed storage.

`color_groups` - coloring of independent variables for compressed storage obtained from `Sparsity_Pattern`.

`seed` - seed matrix for compressed storage obtained from `color_groups`.

`reason` - text string giving reasons used by `MADJacInternal` for choosing or rejecting an AD technique.

When initialised with `MADobj=MADsetup(f)`, where `f` is the function handle of the function to be differentiated, the components `func_handle` and `handle_info` are initialised appropriately and all other components are set to be empty matrices `[]`.

The user can also supply an optional argument `'sparsity_fixed'`, followed by a string '`true`' or '`false`', e.g. `MADobj=MADsetup(f,'sparsity_fixed','true')`. This parameter specifies whether the sparsity pattern is fixed throughout or not (default). If this argument is not explicitly set to 'true', we set `MADobj.use_fwd_compressed=0` to ensure that compressed matrix storage [5, Chap. 7] is never used.

At the time of writing, MAD has 3 techniques available for Jacobian calculation, all using the forward mode, but with either full matrix storage of derivatives, sparse matrix storage [5, Chap. 6] or compressed matrix storage. The components `use_ad_fwd_full`, `use_ad_fwd_sparse` and `use_ad_fwd_compressed` of `MADobj` are used as follows. Firstly if one technique is found less efficient than another then its corresponding component is set to zero. For example, if initially forward mode with compression (if permitted) is faster than forward mode with full storage, then we should never use full storage again so we set `use_ad_fwd_full=0` **but** we do not set `use_ad_fwd_compressed=1` since it might be

inferior to the sparse matrix storage approach of `use_ad_fwd_sparse`. Once compressed and sparse mode have been compared, then one is again rejected, leaving the most efficient technique to be used thereafter. Continuing our example, if sparse storage is more efficient than compressed, then we reject compression by setting `use_ad_fwd_compressed=0` and must thereafter use the remaining sparse storage approach and indicate this by setting `use_ad_fwd_sparse=1`. From this we see that values of components of `MADobj` must be obtained by performing Jacobian calculations which we do via successive calls to `MADJacInternal`.

3.2 Calling `MADJacInternal`

On the first call of `MADJacInternal` as in (1) or (2) the following actions are taken.

- The number of elements in all independent arguments, `x{i}` with `i` in `ActiveIndependents`, is calculated and stored as `MADobj.n`.

  1. If `MADobj.n<MADMinSparseN`, with `MADMinSparseN` a configurable global parameter of default value 10, then the Jacobian calculation is deemed too small to be worthy of sparsity exploitation. We set `MADobj.use_ad_fwd_full =1`, `MADobj.use_ad_fwd_sparse =0`, `MADobj.use_ad_fwd_compressed =0`, indicating that thereafter forward mode with full storage will be used. The Jacobians are evaluated using the `fmad` class, the associated CPU time is stored in `MADobj.ad_fwd_full_time` and we set the total number of dependent variables `MADobj.m`.
  2. We now consider the sparsity of the Jacobian.
     (a) If the '`sparsity_fixed`' flag is set to '`true`', we seek the Jacobian sparsity pattern. At present this is done by initialising the independent variables to be of `fmad` class, with derivatives taken as appropriate rows of the sparse identity matrix `speye(MADobj.n)` but with values perturbed to reduce the probability of (un)fortuitous cancellations or derivatives of dependent variables (un)fortuitously taking zero values. The function is then evaluated and the sparsity of the resulting Jacobian stored as `MADobj.Sparsity_Pattern`. We note that such an approach is inefficient since it uses an additional Jacobian calculation, admittedly with efficient sparse storage of derivatives, in order to determine the sparsity pattern. The Jacobian is then re-evaluated using unperturbed values, the CPU time stored in `MADobj.ad_fwd_sparse_time` and we set the total number of dependent variables `MADobj.m`.
     (b) If the '`sparsity_fixed`' flag is not set, or explicitly set to '`false`', we are only interested in whether sparse mode outperforms full storage mode. Thus the derivatives are initialised through `speye(MADobj.n)` and the

CPU time for the Jacobian evaluation stored in `MADobj.ad_fwd_sparse_time`. Again the total number of dependent variables is set in `MADobj.m`.

3. We then check for the case of a large or a sparse Jacobian, in which case we would never wish to use the non-sparsity exploiting forward mode with full storage due to excessive CPU and memory requirements. To do this we first see if `MADobj.n>MADMaxDenseN` where `MADMaxDenseN` is a configurable global variable with default value 100 and corresponds to the largest total number of independent variables for which we will attempt to use forward mode with full storage. We then check if

$$\texttt{nnz(MADobj.Sparsity\_Pattern)} \;<\; \texttt{MADobj.n*MADobj.m*MADMaxSparseFracForFull},$$

i.e. the fraction of non-zeros in the Jacobian is less than the configurable parameter `MADMaxSparseFracForFull` (default value 0.5). In either of these cases we set `MADobj.use_ad_fwd_full=0` so that forward mode without any sparsity exploitation is never used.

- In both steps 1 and 2 above the Jacobians, revised `MADobj` and, if `Mode='FJ'`, the function results `y1,...,yNout`, are passed back to the calling function.

On subsequent calls to `MADJacInternal` an AD technique yet to be evaluated, including Jacobian compression, is performed and it's CPU time obtained. Any technique for which the CPU time is worse than another is eliminated until one technique remains and that is used thereafter.

Having developed the `MADJacInternal` function, interfaces for other numerical software can now be written.

3.3 High-Level Interfaces

Consider, for example, supplying a function to calculate the Jacobian $d\mathbf{y}/dt = \mathbf{f}(t, \mathbf{y})$ for the stiff ODE solver `ode15s`. We supply a MAD function `MADsetupODE` as in Figure 2.

```
function MADsetupODE(varargin)
global MADODE
MADODE=MADsetup(MADODE,varargin{:});
```

Figure 2: The Function MADsetupODE

On using this function `MADsetupODE(@f)`, a global variable `MADODE` is initialised by passing the function handle argument `@f` to the `MADsetup` function described in Section 3.1.1. We may then use the `MADJacODE` function of Figure 3 to provide the Jacobian

within ode15s. Arguments t, y and any additional user arguments within the varargin are passed into MADJacODE. If the sparsity pattern does not change, the user can call MADsetupODE(@f,'sparsity_fixed','true') for MADJacODE to possibly take advantage of Jacobian compression when it calls MADJacInternal.

```
function Jac=MADJacODE(t,y,varargin)
global MADODE
Mode='J';
Nout=1;
ActiveIndependents=2;
ActiveDependents=1;
[MADODE,Jac]=MADJacInternal(MADODE,...
    Mode,Nout,ActiveIndependents,...
    ActiveDependents,t,y,varargin{:});
```

Figure 3: The MADJacODE function

Since only the Jacobian is required we set Mode='J'. Since we need derivatives of the single function output with respect to the second argument y we accordingly set Nout=1, ActiveDependents=1 and ActiveIndependents=2. We may then call MADJacInternal to evaluate the Jacobian, storing the dummy argument and dummy output MADobj of MADJacInternal as a global variable MADODE in MADJacODE. When MADJacODE is called a second time then the data of MADODE is still available to MADJacInternal. We may now use MAD's automatic differentiation Jacobians within ode15s by nominating MADJacODE as the function for Jacobian evaluation.

The reason for using a function argument and returned value MADODE stored as a global variable in an application interface such as MADJacODE is that then we can simultaneously use MADJacInternal for calculating multiple Jacobians, provided the information for each is stored in a separate global variable. For example, in constrained optimization with the Optimization Toolbox routine fmincon, we use global variable MADOBJOPT within the function MADFandGradObjOpt for calculating the objective function's gradient, and the global variable MADCONSTROPT within the function MADFandJacConstrOpt of Figure 4. This function shows how MADJacInternal may be used to calculate Jacobians of several function outputs, in this case the value of the equality and inequality constraint functions. We may now use fmincon, with function and constraint derivatives enabled by first initialising the MADOBJOPT and MADCONSTROPT via MADsetupObjOpt and MADsetupConstrOpt, by the following code.

```
MADsetupObjOpt(@objfun)
MADsetupConstrOpt(@confun,'sparsity_fixed','true')
[x,fval] = fmincon(@MADFandGradObjOpt,x0,[],[],[],[],[],[],...
             @MADFandJacConstrOpt,options);
```

```
function [C,c,JC,Jc]=MADFandJacConstrOpt(x,varargin)
global MADCONSTROPT
if nargout==2 % just constraints needed
   [C,c]=feval(MADCONSTROPT.func_handle,x,varargin{:});
elseif nargout==4
   Mode='FJ'; % need constraints and Jacobians
   Nout=2; % two outputs from function
   ActiveIndependents=1; % only 1st arg x is active
   ActiveDependents=[1 2];
   [MADCONSTROPT,C,c,JC,Jc]=MADJacInternal(MADCONSTROPT,Mode,Nout,...
      ActiveIndependents,ActiveDependents,x,varargin{:});
   JC=full(JC'); % needed for Toolbox convention
   Jc=full(Jc'); % needed for Toolbox convention
end
```

Figure 4: The `MADFandJacConstrOpt` Function

Here `@objfun` and `@confun` are function handles for objective and constraints respectively. Note that for the the calculations of the objective function we calculate a gradient, i.e. a Jacobian with one row, where compression will save nothing and hence `sparsity_fixed` is not set.

Functions `MADreportODE`, `MADreportObjOpt` and `MADreportConstrOpt` are also provided to display information regarding the corresponding Jacobian calculation.

4 Results

Here we present a stiff ODE, an unconstrained and a constrained optimization test case. All CPU timings are in seconds based on the average of 10 runs on a 2.4 GHz Pentium IV running Windows 2000 and using MATLAB 6.5.

4.1 Ordinary Differential Equations - Brusselator

We used MATLAB's stiff ODE Brusselator example (`brussode`) which makes use of `ode15s` to test performance of `MADJacODE`. Table 1 shows the CPU times obtained for 3 problem sizes N. From row 1, we see that use of finite-differencing without compression is much slower than `MADJacODE` of row 4. However, compressed finite-differencing with (row 3) or without (row 2) vectorization is substantially faster and outperforms `MADJacODE`. Within a single execution of this example the ODE solver `ode15s` only recalculates the Jacobian twice. Consequently `MADJacODE` will effectively calculate the Jacobian 3 times, the first to calculate the sparsity pattern (see Section 3.2). If we repeat the ODE solution with `MADJacODE` a second time then the sparsity pattern is already computed, the appropriate AD technique selected and there is a consequent improvement in efficiency as shown

Jacobian Technique	Number of grid points N 50	100	200
Jacobian by Finite-Differencing	0.2265	0.6453	2.7281
Jacobian by Compressed Finite-Differencing	0.1406	0.2016	0.3313
Jacobian by Compressed, Vectorized Finite-Differencing	0.1375	0.1969	0.3282
Jacobian by `MADJacODE`	0.1844	0.2531	0.4275
Repeat of Jacobian by `MADJacODE`	0.1562	0.2062	0.3110

Table 1: CPU times (s) for the Brusselator problem

in row 4 of Table 1. The CPU times are now on a par with compressed, vectorized finite-differencing without the necessity of supplying the sparsity pattern. From `MADreportODE` we observe that for $n = 50$ sparse forward mode, and for $n > 50$ compressed forward mode are selected.

4.2 Unconstrained Optimization - Extended Rosenbrook

To test the performance of `MADFandGradObjOpt` we considered the extended Rosenbrook function [8, 7],

$$f(\mathbf{x}) = \sum_{i=1}^{n/2} 100(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2,$$

which has a global minimum at $\mathbf{x} = (1, 1, \ldots, 1)$. Using the default BFGS algorithm of `fminunc` for medium sized problems we obtained timings and the L_2 norm of the absolute errors in $\mathbf{x}$ as given in Table 2. By default `fminunc` uses finite differences to approximate

	Size of $\mathbf{x}$ (n)					
	10		20		30	
Gradient Technique	Time(s)	Error	Time(s)	Error	Time(s)	Error
Finite Differencing	0.0781	0.0139	0.1906	0.0213	0.3844	0.0086
FD and reduced tolerance	0.1828	0.0021	0.4891	0.0016	0.3640	0.0086
Analytically supplied	0.0360	0.0001	0.0500	0.0002	0.0672	0.0001
`MADFandGradObjOpt`	0.0344	0.0001	0.0907	0.0002	0.1468	0.0001

Table 2: CPU Times and final solution error for the Rosenbrook Example

the gradient. Due to the flat bottom of the function, it is difficult to obtain the gradient sufficiently accurately from finite differences and `fminunc` returns with poor solutions. Thus the default convergence tolerances were changed to the square root of relative precision $\sqrt{\texttt{epsilon}}$ for both x and function values giving the second row of results, generally

reducing the error. The third row shows the results with the gradient supplied analytically and the final row obtained via use `MADFandGradObjOpt`. We see that for $n = 10$ finite differencing is not only slower, but also less accurate than analytically supplied derivatives or our AD. As we increase the problem size, even with reduced tolerances the FD approach maintains a large error. Although as expected AD is slower than analytically supplied derivatives, it is still much faster than FD and achieves the solution accuracy of analytic derivatives.

Through `MADreportObjOpt` we see that for $n > 10$ sparse AD mode is selected, as sparse storage was faster than full, but compression failed to reduce costs since $m = 1$.

4.3 Constrained Optimization - Multivariate Regression Model

We compared the finite differences and automatic differentiation for constrained optimization by fitting regression models as in [10]. A quadratic model is used within a data-fitting problem, with constraints being set on the eigen-decomposition of the quadratic term. Table 3 shows the results for one particular sets of constraints. The objective

Number of unknowns	66	180	384
Number of constraints	18	75	168
Derivative Technique	CPU Time		
Finite Differencing Time	12.234	61.203	743.219
`MADFandGradObjOpt`,`MADFandJacConstrOpt`	3.422	25.954	496.266
AD Mode for `MADFandJacConstrOpt`	Full	Sparse	Compressed

Table 3: Regression Model

function and constraints are now sufficiently involved that analytic derivatives are not available. As well as the improved run time obtained with MAD we also note that the error in the position of the minimum obtained was 2 orders of magnitude smaller than that obtained using finite-differencing. Interestingly, the bottom row of Table 3 shows the mode of AD for the constraints shifting automatically from full, to sparse to compressed as the problem size increases.

5 Conclusions and Future Developments

In this paper we have presented the `MADJacInternal` function which enables automated, performance driven selection of a Jacobian evaluation algorithm via the forward mode `fmad` class of the MAD package. By wrapping calls to `MADJacInternal` in interface functions designed to inter-operate with higher level numerical algorithms, such as stiff ODE solution and optimization, the use of AD becomes trivial. If the user confirms that the sparsity pattern is fixed, we will try compression techniques. The user does not have to supply the sparsity pattern (as is required with present generation numerical packages such as `ode15s`). In the 3 cases presented here, the use of AD gave comparable or

improved run-times compared to finite-difference generated Jacobians. In addition, for optimization problems the error in the solution was reduced. Given its well-defined interface and ease of use, we foresee `MADJacInternal` being used directly within numerical packages in MATLAB. This will ease the transition of AD into application software.

At the time of writing, MAD's capabilities are being extended in two ways. To avoid the user specifying whether the sparsity pattern is fixed or not, a sparsity detection class is under development, broadly similar to that of [13], but featuring capabilities to detect branching based on the value of active variables, and consequently warning if the sparsity pattern may not be fixed. Secondly, an extended Jacobian based approach enabling reverse mode calculation for small to medium size problems is being written. Plans are in place for a tape based reverse mode and some preliminary investigations into source transformation are underway.

Acknowledgements

We thank Trevor Ringrose for use of the test case of Section 4.3.

REFERENCES

[1] C. Bischof, H. Bücker, B. Lang, A. Rasch, and A. Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages p. 65–72. IEEE Computer Society, 2002.

[2] T. F. Coleman and A. Verma. ADMAT: An automatic differentiation toolbox for MATLAB. Technical report, Computer Science Department, Cornell University, 1998.

[3] S. A. Forth. An implementation of forward mode automatic differentiation in MATLAB. *In Preparation*, 2004.

[4] S. A. Forth and M. M. Edvall. *User Guide for* MAD - *a* MATLAB *Automatic Differentiation Toolbox TOMLAB/MAD*. TOMLAB, Tomlab Optimization Inc., 455 Union St. No 13, Arcata CA, USA, version 1.1 edition, March 2004. See `http://tomlab.biz/docs/TOMLAB_MAD.pdf`.

[5] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, Penn., 2000.

[6] The Mathworks Inc., 3 Apple Hill Drive, Natick MA 01760-2098. *Optimization Toolbox User's Guide*, July 2002. Online at `http://www.mathworks.com/access/helpdesk/help/pdf_doc/optim/optim_tb.pdf`.

[7] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. Algorithm 566: FORTRAN subroutines for testing unconstrained optimization software [C5 [E4]]. *ACM Transactions on Mathematical Software*, 7(1):136–140, Mar. 1981.

[8] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1):17–41, Mar. 1981.

[9] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operational Research. Springer, New York, 1999.

[10] T. J. Ringrose and S. A. Forth. Improved fitting of constrained multivariate regression models using automatic differentiation. In W. Hardle and B. Ronz, editors, *COMPSTAT 2002: Proceedings in Computational Statistics, 15th Symposium*, pages 383–388, Berlin, Germany, 2002. Physica-Verlag, Heidelberg.

[11] L. Shampine, R. Ketzscher, and S. A. Forth. Using AD to solve BVPs in MATLAB. Applied Mathematics and Operational Research Report AMOR 2003/4, Cranfield University (RMCS Shrivenham), Swindon, SN6 8LA, UK, 2003.

[12] A. Vehreschild. Semantic augmentation of MATLAB programs to compute derivatives. Diploma thesis, Institute for Scientific Computing, Aachen University of Technology (RWTH Aachen), Germany, Nov. 2001.

[13] A. Verma. *Structured Automatic Differentiation.* PhD thesis, Cornell University Department of Computer Science, Ithaca, NY, 1998.