User Guide for
Mad - a Matlab Automatic Differentiation Toolbox
TOMLAB /MAD
Version 1.4

The Forward Mode

Shaun A. Forth[1] and Marcus M. Edvall[2]

June 20, 2007

[1]Engineering Systems Department, Cranfield University, Defence Academy, SWINDON SN6 8LA, U.K., S.A.Forth@cranfield.ac.uk.
[2]Tomlab Optimization Inc., 855 Beech St #121, San Diego, CA, USA, medvall@tomopt.com.

Abstract

MAD is a MATLAB library of functions and utilities for the automatic differentiation of MATLAB functions/statements via operator and function overloading. Currently the forward mode of automatic differentiation is supported via the `fmad` class. For a single directional derivative objects of the `fmad` class use MATLAB arrays of the same size for a variable's value and its directional derivative. Multiple directional derivatives are stored in objects of the `derivvec` class allowing for an internal 2-D, matrix storage so allowing the use of sparse matrix storage for derivatives and ensuring efficient linear combination of derivative vectors via high-level MATLAB functions. This user guide covers:

- installation of MAD on UNIX and PC platforms,
- using TOMLAB /MAD,
- basic use of the forward mode for differentiating expressions and functions
- advanced use of the forward mode including:
  - dynamic propagation of sparse derivatives,
  - sparse derivatives via compression,
  - differentiating implicitly defined functions,
  - control of dependencies,
- use of high-level interfaces for solving ODEs and optimization problems outside of the TOMLAB framework,
- differentiating *black-box* functions for which derivatives are known.

Contents

1 Introduction

Automatic (or algorithmic) differentiation (AD) is now a widely used tool within scientific computing. AD concerns the mathematical and software process of taking a computer code, which calculates some outputs $\mathbf{y}$ in terms of inputs $\mathbf{x}$ through the action of some function $\mathbf{y} = \mathbf{F}(\mathbf{x})$, and calculating the function's derivatives, e.g. the Jacobian $\mathbf{F}'(\mathbf{x}) = \partial\mathbf{F}/\partial\mathbf{x}$. The standard reference is the book [Gri00]. It is common notation to take $\mathbf{x} \in I\!R^n$ and $\mathbf{y} \in I\!R^m$. In AD the function $\mathbf{F}$ is considered to be defined by the computer code which is termed the *source* or *source code*.

A variety of tools exist for AD of the standard programming languages including:

Fortran : ADIFOR [BCKM96], TAF/TAMC [GK96], Tapenade [INR05], ADO1 [PR98].

C,C++ : ADIC [BRM97], ADOL-C [GJU96]

All these tools will calculate Jacobian-vector products $\mathbf{F}'\mathbf{v}$ by the forward (tangent-linear) mode of AD. Some will calculate vector-Jacobian products $\mathbf{u}^T\mathbf{F}' = \mathbf{F}'^T\mathbf{u}$ via the reverse (adjoint) mode. Some will efficiently calculate matrix products $\mathbf{F}'\mathbf{V}$, $\mathbf{U}^T\mathbf{F}'$ and thus facilitate calculation of the Jacobian $\mathbf{F}'$ by taking $\mathbf{V} = \mathbf{I}_n$ or $\mathbf{U} = \mathbf{I}_m$ (where $\mathbf{I}_n$ is the $n \times n$ identity matrix). Additionally a few will determine second derivatives (Hessians, Hessian-vector products) and even arbitrary order derivatives. An up to date list of AD tools may be found at the website [aut05]. The recent textbook [Gri00] and conference proceedings [BCH+05] detail the current state of the art in the theory and implementation of AD techniques.

For these languages listed above AD is implemented either via source-text transformation [BCKM96, GK96, INR05, BRM97] or overloading [PR98, GJU96]. In source-text transformation the original code that numerically defines the function $\mathbf{F}$ is transformed, via sophisticated compiler techniques, to new code that calculates the required derivatives. In overloading the ability offered by modern programming languages to define new data-types and overload arithmetic operations and intrinsic function calls is utilised to systematically propagate derivatives through the source code and hence calculate the necessary derivatives of the function.

In MATLAB progress has been slower. Rich and Hill [RH92] detail a TURB-C function, callable from MATLAB that allows a function defined by a character string to be differentiated by the forward mode and returned as another character string. The resulting string representing the derivatives may then be *evaluated* for any particular input x. We view this approach more as a symbolic algebra technique for differentiating a single statement rather than AD which is concerned with many lines of code containing subfunctions, looping etc. and which may make use of symbolic/compiler techniques.

Modern developments began with Coleman & Verma's produced the ADMIT Toolbox [CV00] allowing functions specified by C-code to be differentiated and accessed from within MATLAB through use of ADOL-C. Of even greater relevance is the pioneering work of Verma who demonstrated in his ADMAT tool [Ver99, Ver98b] how the object oriented features of MATLAB version 5 (not available to Rich and Hill in 1992) could be used to build up overloaded libraries of MATLAB functions allowing first and second order derivatives to be calculated using both forward and reverse AD. ADMIT may now calculate Jacobians via ADMAT.

Our experience of ADMAT is that, although it has floating point operations counts in line with theory, it runs rather slowly. Additionally we have had some difficulties with robustness of the software. With these deficiencies in mind we reimplemented the forward mode of AD for first derivatives in a new MATLAB toolbox named MAD (MATLAB Automatic Differentiation) [For06].

In common with other overloaded, forward mode AD tools MAD evaluates a function $\mathbf{F}(\mathbf{x})$ at $\mathbf{x} = \mathbf{x}_0$ and in the course of this also evaluates the directional derivative $\mathbf{F}'(\mathbf{x}_0)\mathbf{v}$ This is done by first initialising $\mathbf{x}$ as an `fmad` object using (in MATLAB syntax) `x = fmad(x0,v)` to specify the point $\mathbf{x}$ of evaluation and the directional derivative $\mathbf{v}$. Then evaluation of `y=F(x)` propagates `fmad` objects which, through overloading of the elementary operations and functions of MATLAB ensures that directional derivatives and values of all successive quantities and in particular $\mathbf{y}$ are calculated. By specifying $\mathbf{v} = \mathbf{V}$, a matrix forming several directional derivatives, then several directional derivatives may be propagated simultaneously. In particular, if $\mathbf{V} = \mathbf{I}$ the identity matrix, then the Jacobian $\mathbf{F}'(\mathbf{x})$

at $\mathbf{x} = \mathbf{x}_0$ is evaluated. The tool is not restricted to vector valued functions. If the result of a calculation is a N-dimensional array, then directional derivatives are N-dimensional arrays, and an $N+1$ dimensional array may be propagated to handle multiple directional derivatives.

Our implementation features:

- A new MATLAB class `fmad` which overloads the builtin MATLAB arithmetic and some intrinsic functions.
- Separation of storage and manipulation of derivatives into a separate class `derivvec`. This greatly reduces the complexity of the MATLAB code for the `fmad` class. Additionally it allows for the straightforward optimisation of the derivative vector operations using the high level matrix/array functions of MATLAB.
- The `derivvec` class also allows for the use of MATLAB's sparse matrix representation to exploit sparsity in the derivative calculations at runtime for arbitrary dimension arrays.
- *New for version 1.2:* Support for compressed Jacobian calculations when a sparse Jacobian has known sparsity pattern detailed in section 5.4.
- *New for version 1.2:* High-level interfaces for enabling use of MAD with MATLAB's Optimization Toolbox and stiff ODE solvers as described in Section 6.
- *New for version 1.2:* Black box interfaces of section 7 for interfacing MAD with code for which derivative code already has been developed or for calling external procedures for which derivatives are known.
- *New for version 1.2:* all examples now available as script M-files and added to the MATLAB path on startup.
- *New for version 1.2:* Improved help facilities allowing removal of the Appendix of previous versions of this manual and introduced in Section 2.5.

Coverage of MATLAB's huge range of intrinsic functions is being constantly extended with new functionality added as demanded; so allowing for carefully testing on user's test cases before widespread release.

An implementation of reverse mode AD is currently under development.

This document constitutes a User Guide for the `fmad` class of MAD. Section 2 covers installing MAD. Section 4 introduces basic usage of the forward mode via some simple examples. Section 5 details more advanced usage of the forward mode and in particular provides guidance on how MATLAB functions should be written to give effective usage of MAD. Section 6 introduces the high-level interfaces first described in [FK04]. These allow users of MATLAB' Optimisation Toolbox [Mat06b] or MATLAB's stiff ODE solvers [mat06a, Chapter 5] to easily use MAD for calculating function gradients or Jacobians. Section 7 details *black box* interfaces allowing users to interface MAD to functions whose derivatives are known and which possibly are coded in external *mex* files written in Fortran or C.

2 Installation of Mad

In this section we detail the steps needed to install MAD on a UNIX or PC platform.

2.1 Requirements

MAD requires MATLAB version 6.5 or higher.

2.2 Obtaining Mad

MAD is available from Tomlab Optimization.

2.3 Installation Instructions (UNIX)

MAD is supplied in the gzipped TOMLAB tar file `*.tar.gz`. Please follow the these instructions.

1. Place `*.tar.gz` in an appropriate directory `dir`. TOMLAB and MAD will be placed in its own sub-directory so you don't need to create a specific `TOMLAB or MAD` directory.
2. Now, from within directory `dir`, `gunzip`, and `tar` extract. MAD will be installed in `tomlab/mad`

   ```
   gunzip *.tar.gz
   tar xvf *.tar
   ```

   If you have any difficulties using `gunzip` or `tar` please check with your system's administrator that they are installed and accessible to you before contacting the author.
3. To be able to use MAD it must be in your MATLAB `path` and must have several other `paths` and MATLAB global variables initialized. To ensure this occurs correctly on starting TOMLAB you must edit your personal TOMLAB `startup.m` in `/tomlab`. MAD succesfully installed should appear in your MATLAB command window when starting TOMLAB.

2.4 Installation Instructions (Windows)

Follow the directions in the TOMLAB installer. Edit the TOMLAB `startup.m` file to make sure that MAD will be initialized when starting TOMLAB. MAD succesfully installed should appear in your MATLAB command window when starting TOMLAB.

2.5 Online Help Facilities

After successful installation you can get help on many aspects of using MAD from the MATLAB command line. For example,

- `help MAD`
  gives a list of web-pages, directories and classes associated with MAD.
- `help fmad/Contents`
  lists all the functions of the `fmad` class used for forward mode automatic differentiation.

- `help madutil`
  lists all MAD's utility functions designed to ease use of automatic differentiation.
- `help MADEXAMPLES`
  lists the folders contained example scripts and source files, many of which we meet in this user guide.
- `help MADbasic`
  lists the example scripts and function files for all the basic examples.

All the example scripts, which start with `MADEX` (e.g. `MADEXForward1`), have been written in cell mode allowing us to generate the LaTeX source code for most examples featured in this user guide.

3 Using TOMLAB /MAD

MAD can be used for automatic differentiation in TOMLAB.

TOMLAB features several methods for obtaining derivatives of objective function, gradients, constraints and jacobians.

- Supplying analytical derivatives. This is the recommended method at almost all times, as the highest speed and robustness are provided.
- Five methods for numerical differentiation. This is controlled by the flags, *Prob.NumDiff* and *Prob.ConsDiff*.
- The solver can estimate the derivatives internally. This is not available for all solvers. TOMLAB /SNOPT also provides derivative checking. See the TOMLAB /SNOPT manual for more information.
- Automatic differentiation with MAD. This is controlled by setting the flags *Prob.ADObj* and *Prob.ADCons* to 1 or -1. If the flags are set to 1, only the objective function and constraints are given. If the flags are set to -1, second derivatives will be calculated. In this case the first derivatives must be supplied.

3.1 Example

The following example illustrates how to use TOMLAB /MAD when the objective function and constraints are given. MAD uses global variables that need to be initialized.

```
madinitglobals;
Prob = conAssign( ... ); % A standard problem structure is created.
Prob.ADObj  = 1;         % First derivatives obtained from MAD for objective.
Prob.ADCons = 1;         % First derivatives obtained from MAD for constraints.
Result = tomRun('snopt', Prob, [], 1) % Using driver routine tomRun.
```

If the user has provided first order information as well, the following code could be used. Observe that the solver must use second order information for this to be useful.

```
madinitglobals;
Prob = conAssign( ... );  % A standard problem structure is created.
Prob.ADObj  = -1;         % Second derivatives obtained from MAD for objective.
Prob.ADCons = -1;         % Second derivatives obtained from MAD for constraints.
Result = tomRun('knitro', Prob, [], 1) % Using driver routine tomRun.
                          % KNITRO uses second order information.
```

The number of MATLAB functions that MAD supports is limited. Please report needed additions to support@tomopt.com.

4 Basic Use of Forward Mode for First Derivatives

4.1 Differentiating Matlab Expressions

Using forward mode AD, MAD can easily compute first derivatives of functions defined via expressions built-up of the arithmetic operations and intrinsic functions of MATLAB. For the moment let us simply consider the trivial operation $y = f(x) = x^2$ for which the derivative is $2x$. Let us determine this derivative at $x = 3$ for which $f(3) = 9$ and $df(3)/dx = 6$. We will make use of the `fmad` constructor function `fmad`, and the functions `getvalue` and `getderivs` for extracting values and derivatives from `fmad` objects.

Example 4.1 MADEXForward1: Forward Mode Example $y = x^2$

This example demonstrates how to differentiate simple arithmetic expressions in Matlab.

See also MADEXForward2

Contents

Differentiating $y = f(x) = x^2$

MAD can easily compute first derivatives of functions defined via expressions built-up of the arithmetic operations and intrinsic functions of Matlab using forward mode AD. For the moment let us simply consider the trivial operation

$$y = f(x) = x^2$$

for which the derivative is

$$\frac{dy}{dx} = 2x.$$

Let us determine this derivative at `x=3` for which `f(3)=9` and `df(3)/dx=6`.

Using fmad

We first define `x` *to be of* `fmad` *class and have value 3 and derivative 1 (since the derivative with respect to the independent variable is one,* `dx/dx=1`*). The command and Matlab output is as below.*

```
format compact % so there are no blank lines in the output
x=fmad(3,1) % creates the fmad object with value 3 and derivative 1
```

```
value =
     3
derivatives =
     1
```

Calculating y

The value of the object `x` *is* `3` *and its derivatives correspond to an object of size* `[1 1]` *(a scalar) with one derivative of value* `1`*. We now calculate* `y` *exactly as we would normally, allowing the* `fmad` *function libraries to take care of the arithmetic associated with values and derivatives.*

```
y=x^2
```

```
value =
     9
derivatives =
     6
```

Extracting the value

We extract the value of `y` *using the MAD function* `getvalue`*.*

```
yvalue=getvalue(y)
```

```
yvalue =
     9
```

Extracting the derivatives

Similarly we extract the derivatives using `getderivs`*.*

```
yderivs=getderivs(y)
```

```
yderivs =
     6
```

4.2 Vector-Valued Functions

Let us now consider a simple, linear vector-valued function for which we may use `fmad` to calculate Jacobian-vector products or the function's Jacobian. We also see, for the first time `fmad`'s `getinternalderivs` function.

Example 4.2 MADEXForward2: Forward Mode Example $y = A * x$

This example demonstrates how to calculate Jacobian-vector products and how to calculate the Jacobian for a simple vector-valued function.

See also MADEXForward1

Contents

Differentiating $y = f(x) = A * x$

MAD easily deals with vector-valued functions. Let us consider the function

$$y = f(x) = A * x$$

at `x=[1; 1 ;1 ]` *where* `A` *is the matrix,*

```
A=[ 1  4  7
    2  5  8
    3  6  9]
```

for which the Jacobian is the matrix `A` *and directional derivatives in the 3 respective coordinate directions are the 3 respective columns of* `A`*.*

```
format compact;
A=[1 4 7;2 5 8; 3 6 9];
xval=[1;1;1];
```

Directional derivative in [1;0;0] direction

Using the `fmad` *constructor function we initialise* `x` *with its directional derivative, calculate* `y` *and extract the derivative. We find the correct directional derivative* `[1;2;3]`*.*

```
x=fmad(xval,[1;0;0])
y=A*x;
dy=getderivs(y)
```

```
fmad object x
value =
     1
     1
     1
derivatives =
     1
     0
     0
dy =
     1
     2
     3
```

Directional derivative in [0;1;0] direction

Similarly in the `[0;1;0]` *direction we get* `[4;5;6]`*.*

```
x=fmad(xval,[0;1;0]);
y=A*x;
dy=getderivs(y)
```

```
dy =
     4
     5
     6
```

Calculating the Jacobian

To obtain the full Jacobian we initialise the derivative with the 3 x 3 identity matrix.

```
x=fmad(xval,eye(3))
y=A*x
dy=getderivs(y)
```

```
fmad object x
value =
     1
     1
     1
derivvec object derivatives
Size = 3  1
No. of derivs = 3
derivs(:,:,1) =
     1
     0
     0
derivs(:,:,2) =
     0
     1
     0
derivs(:,:,3) =
     0
     0
     1
fmad object y
value =
    12
    15
    18
derivvec object derivatives
Size = 3  1
No. of derivs = 3
derivs(:,:,1) =
     1
     2
     3
derivs(:,:,2) =
     4
     5
     6
derivs(:,:,3) =
     7
     8
     9
dy(:,:,1) =
     1
     2
     3
dy(:,:,2) =
     4
     5
     6
dy(:,:,3) =
     7
     8
```

```
     9
```

But dy is a 3-D array not a matrix!

Notice now that we have a 3-dimensional array returned for a result we expect to be a matrix. This is easily remedied using the Matlab intrinsic `squeeze`*, which squeezes out singleton dimensions, or more conveniently using* `getinternalderivs` *to extract the derivatives from the* `fmad` *object* `y`*.*

```
dy=squeeze(dy)
dy=getinternalderivs(y)
```

```
dy =
     1     4     7
     2     5     8
     3     6     9
dy =
     1     4     7
     2     5     8
     3     6     9
```

This example brings us to an important topic, how are the derivatives stored in MAD?

4.3 Storage of Derivatives in Mad

MAD stores derivatives in two ways depending upon whether a variable of `fmad` class possesses single, or multiple directional derivatives.

4.3.1 Single Directional Derivatives

For a variable of `fmad` class with a single directional derivatives the derivatives are stored as an array of class `double` with the same shape as the variable's value. The `getderivs` function will therefore return this array. We have seen this behaviour in Example 4.1 and Example 4.2.

4.3.2 Multiple Directional Derivatives

For a variable of `fmad` class with multiple directional derivatives, the derivatives are stored in an object of class `derivvec`. MAD has been written so that multiple directional derivatives may be thought of as an array of one-dimension higher than that of the corresponding value using, what we term, the derivative's *external representation*. MAD also has an *internal representation* of derivatives that may often be of use, particularly when interfacing MAD with another MATLAB package. Accessing the internal representation is covered in section 5.1. Consider an N-dimensional MATLAB array `A` with `size(A)=[n1 n2 n3 ... nN]`, such that `(n1 n2 ... nN)` are positive integers. Then the external representation of `A`'s derivatives, which we shall call `dA` is designed to have `size(dA)=[n1 n2 n3 ... nN nD]`, where `nD`>1 is the number of directional derivatives. An example or two may make things clearer.

Example 4.3 MADEXExtRep1: External representation of a matrix

This example illustrates MAD's external representation of derivatives for a matrix.

See also MADEXExtRep2, MADEXIntRep1

Contents

Define a 2 x 2 matrix A.

```
format compact
A=[1 2; 3 4]
```

```
A =
     1     2
     3     4
```

Multiple directional derivatives for A

Define a 2 x 2 x 3 matrix (note the transpose) to provide 3 directional derivatives for `dA`*.*

```
dA=reshape([1 2 3 4;5 6 7 8;9 10 11 12]',[2 2 3])
```

```
dA(:,:,1) =
     1     3
     2     4
dA(:,:,2) =
     5     7
     6     8
dA(:,:,3) =
     9    11
    10    12
```

The fmad object

We create the `fmad` *object,*

```
A=fmad(A,dA)
```

```
value =
     1     2
     3     4
Derivatives
Size = 2  2
No. of derivs = 3
derivs(:,:,1) =
     1     3
     2     4
derivs(:,:,2) =
     5     7
     6     8
derivs(:,:,3) =
     9    11
    10    12
```

The derivs component

The last index of the `A`*'s* `derivs` *component refers to the directional derivatives and we see that, as expected, we have a 2 x 2 x 3 array which makes up 3 sets of 2 x 2 directional derivative matrices.*

Note that `fmad` (like MATLAB and Fortran) assumes a *column major* (or first index fastest) ordering for array storage. The derivatives supplied to the constructor function `fmad` are therefore assumed to be in column major order and the external representation `reshape`s them appropriately. We therefore needn't bother reshaping the derivatives before passing them to `fmad` as the example below shows.

Example 4.4 MADEXExtRep2: External Representation Example

This example shows the use of column-major ordering for an input derivative.

See also MADEXExtRep1, MADEXIntRep1

Contents

Define a 2 x 2 matrix A.

```
format compact
A=[1 2; 3 4]
```

```
A =
     1     2
     3     4
```

Multiple directional derivatives for A

Define a vector to provide directional derivatives for `dA`*.*

```
dA=[1 2 3 4 5 6 7 8 9 10 11 12];
```

The fmad object

We create the `fmad` *object,*

```
A=fmad(A,dA)
```

```
value =
     1     2
     3     4
Derivatives
Size = 2  2
No. of derivs = 3
derivs(:,:,1) =
     1     3
     2     4
derivs(:,:,2) =
     5     7
     6     8
derivs(:,:,3) =
     9    11
    10    12
```

The derivs component

Notice how the elements of `dA` *are taken in column major order to provide the 3 directional derivatives of A in turn.*

4.4 Use of Intrinsics (Operators and Functions)

We have produced overloaded `fmad` functions for most commonly used MATLAB operators and functions. For example addition of two `fmad` objects is determined by the function `@fmad\plus`, subtraction `@fmad\minus`, etc. To see which overloaded intrinsic operators and functions are provided type

```
help fmad.Contents
```

at the MATLAB prompt. The help information for individual functions describes the function and any additional information such as restrictions, e.g.

```
help fmad.plus
```

4.5 Additional Functions

We have also coded a small number of additional functions which are not part of MATLAB but application specific. For example, for a matrix `A`, `logdet(A)` calculates `log(abs(det(A)))` a function used in statistical parameter estimation. This composition of functions may be differentiated very efficiently by considering it in this composite form [Kub94, For01]. Note that a version of the function valid for arguments of classes `double` and `sparse` is provided in the `MAD/madutil`.

4.6 Differentiating User Defined Functions

In section 4.1 we saw how simple MATLAB expressions could be differentiated using MAD. It is usually straightforward to apply the same methodology to differentiating user defined functions.

Example 4.5 (Brusselator ODE Function) *Consider the function which defines the Brusselator stiff ODE problem from the* MATLAB *ODE toolbox, supplied as* `brussode_f.m` *and shown in Figure 1. For a given scalar time* `t` *(unused) and vector* `y` *of* `2N` *rows, this function* `brussode_f` *will supply the right-hand-side function for an ODE of the form,*

$$\frac{d\mathbf{y}}{dt} = \mathbf{F}(t, \mathbf{y}).$$

The Jacobian of the function is required to solve the ODE efficiently via an implicit stiff ODE solver (e.g., `ode15s` *in* MATLAB*). The function has actually been written in so-called* vectorised *form, i.e. if* `y` *is a matrix then the* `dydt` *supplied will also be a matrix with each column of* `dydt` *corresponding to the appropriate column of* `y`*. This approach allows for the efficient approximation of the function's Jacobian via finite-differences.*

It is trivial to calculate the Jacobian of the function of figure 1 using MAD*. The commands to do so are given in the script* `MADEXbrussodeJac.m`*. The script also includes a simple one-sided finite-difference (FD) calculation of the Jacobian.*

MADEXbrussodeJac: Full Jacobian of Brusselator Problem

This example shows how to use the `fmad` *class to calculate the Jacobian of the Brusselator problem and compare the result with those from finite-differencing. More efficient AD approaches are available as shown in* `MADEXbrussodeJacSparse`*,* `MADEXbrussodeJacComp`

See also: brussode_f, brussode_S, MADEXbrussodeJacSparse, MADEXbrussodeJacComp, brussode

Contents

Initialise variables

We define the input variables.

```
N=3;
t=0;
y0=ones(2*N,1);
```

Use fmad to calculate the Jacobian

We initialise y with the value y0 and derivatives given by the identity matrix of size equal to the length of y0. We may then evaluate the function and extract the value and Jacobian.

```
y=fmad(y0,eye(length(y0)));
F=brussode_f(t,y,N);
F_fmad=getvalue(F); % grab value
JF_fmad=getinternalderivs(F) % grab Jacobian
```

```
JF_fmad =
   -2.6400    1.0000    0.3200         0         0         0
    1.0000   -1.6400         0    0.3200         0         0
    0.3200         0   -2.6400    1.0000    0.3200         0
         0    0.3200    1.0000   -1.6400         0    0.3200
         0         0    0.3200         0   -2.6400    1.0000
         0         0         0    0.3200    1.0000   -1.6400
```

Using finite-differencing

We make 2N extra copies of y, add a perturbation to the copies, perform all function evaluations taking advantage of the vectorization of the function `brussode_f`*, and approximate the derivatives using 1-sided finite-differences.*

```
y=repmat(y0,[1 2*N+1]);
y(:,2:end)=y(:,2:end)+sqrt(eps)*eye(2*N);
F=brussode_f(t,y,N);
F_fd=F(:,1);
JF_fd=(F(:,2:end)-repmat(F_fd,[1 2*N]))./sqrt(eps)
```

```
JF_fd =
   -2.6400    1.0000    0.3200         0         0         0
    1.0000   -1.6400         0    0.3200         0         0
    0.3200         0   -2.6400    1.0000    0.3200         0
         0    0.3200    1.0000   -1.6400         0    0.3200
         0         0    0.3200         0   -2.6400    1.0000
         0         0         0    0.3200    1.0000   -1.6400
```

Comparing the AD and FD Jacobian

The function values computed are, of course, identical. The Jacobians disagree by about 1e-8, this is due to the truncation error of the finite-difference approximation.

```
disp(['Function values norm(F_fd-F_fmad)  = ',num2str(norm(F_fd-F_fmad,inf))])
disp(['Function Jacobian norm(JF_fd-JF_fmad)= ',num2str(norm(JF_fd-JF_fmad,inf))])
```

```
Function values norm(F_fd-F_fmad)  = 0
Function Jacobian norm(JF_fd-JF_fmad)= 2.861e-008
```

```
function dydt=brussode_f(t,y,N)
% brussode_f: defines ODE for the Brusselator problem.
%             Taken from matlab 6.5 brussode.
%
% See also MADEXbrussode, brussode_S, brussode

% AUTHOR: S.A.Forth
% DATE: 30/5/06
% Copyright 2006: S.A.Forth, Cranfield University
%
% REVISIONS:
% DATE  WHO   WHAT
c = 0.02 * (N+1)^2;
dydt = zeros(2*N,size(y,2));      % preallocate dy/dt

% Evaluate the 2 components of the function at one edge of the grid
% (with edge conditions).
i = 1;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + c*(1-2*y(i,:)+y(i+2,:));
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + c*(3-2*y(i+1,:)+y(i+3,:));

% Evaluate the 2 components of the function at all interior grid points.
i = 3:2:2*N-3;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
    c*(y(i-2,:)-2*y(i,:)+y(i+2,:));
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
    c*(y(i-1,:)-2*y(i+1,:)+y(i+3,:));

% Evaluate the 2 components of the function at the other edge of the grid
% (with edge conditions).
i = 2*N-1;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + c*(y(i-2,:)-2*y(i,:)+1);
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + c*(y(i-1,:)-2*y(i+1,:)+3);
```

Figure 1: Brusselator function supplied with MATLAB ODE Toolbox

We should note that the above approach is not efficient either for FD or AD for large `N`. The Jacobian is sparse and exploiting the sparsity greatly increases the efficiency of both approaches. Exploiting sparsity will be covered in Sections 5.3 and 5.4.

4.7 Possible Problems Using the Forward Mode

In many cases `fmad` will be able to differentiate fairly involved functions as shown in Figure 1. However some problems in terms of robustness and efficiency may occur.

1. Assigning an `fmad` object to a variable of class `double`. For example,

   ```
   >> x=fmad(1,1);
   >> y=zeros(2,1);
   >> y(1)=x;
   ??? Conversion to double from fmad is not possible.
   ```

   See Section 5.2 and Example 5.4 for a solution to this problem.

2. Differentiation for a particular function is not supported. For example,

   ```
   >> x=fmad(1,1);
   >> y=tanh(x)
   ??? Error using ==> tanh
   Function 'tanh' not defined for variables of class 'fmad'.
   ```

   Please contact the author to arrange for the `fmad` class to be extended for your case.

3. Derivatives are not accurate when differentiating an iteratively defined function. See section 5.5 for an approach to this issue.

4. Differentiation is slow compared to finite-differences. The `fmad` and `derivvec` class are designed to be competitive with finite-differencing in terms of speed of execution. They will of course return derivatives correct to within machine rounding. If this is not the case then please contact the author.

5. Derivatives are incorrect. Then you have discovered a bug, please contact the author.

5 Advanced Use of Forward Mode for First Derivatives

We have seen in section 4 how MAD via overloaded operations on variables of the `fmad` class may calculate first derivatives. In this section we explore more advanced use of the `fmad` class.

5.1 Accessing The Internal Representation of Derivatives

We saw in section 4 that `fmad` returns multiple derivatives as N-D arrays of one dimension higher than the corresponding value. Since MATLAB variables are by default matrices, i.e. arrays of dimension 2, then we see that by default for multiple directional derivatives `fmad` will return 3-D (or higher dimension arrays). Although this approach is systematic it can be inconvenient. Frequently we require the Jacobian matrix of a vector valued function. We can trivially `reshape` or `squeeze` the 3-D array obtained by `fmad` to get the required matrix. However, and as we shall see in Section 5.3, this approach will not work if we wish to use MATLAB's sparse matrix class to propagate derivatives. We should bear in mind that the `derivvec` class stores arbitrary dimension derivative arrays internally as matrices. This *internal representation* of the derivatives may be accessed directly through use of the function `getinternalderivs` as the following example shows.

Example 5.1 MADEXIntRep1: Internal Representation

This example illustrates MAD's internal representation of derivatives.

See also fmad, MADEXExtRep1, MADEXExtRep2

Contents

- *Define a vector [1;1] with derivatives given by the sparse identity*
- *Getderivs returns the external form in full storage*
- *Getinternalderivs returns the internal form*

Define a vector [1;1] with derivatives given by the sparse identity

```
format compact
a=fmad([1;1],speye(2))
```

```
value =
     1
     1
Derivatives
Size = 2  1
No. of derivs = 2
d =
   (1,1)        1
d =
   (2,1)        1
```

Getderivs returns the external form in full storage

```
da=getderivs(a)
```

```
da(:,:,1) =
     1
     0
da(:,:,2) =
     0
     1
```

Getinternalderivs returns the internal form

`getinternalderivs` maintains the sparse storage used internally by MAD's `derivvec` class.

```
da=getinternalderivs(a)
```

```
da =
   (1,1)        1
   (2,2)        1
```

See Section 5.3 for more information on this topic.

5.2 Preallocation of Matrices/Arrays

It is well-known in MATLAB that a major performance overhead is incurred if an array is frequently increased in size, or *grown*, to accommodate new data as the following example shows.

Example 5.2 MADEXPrealloc1: importance of array preallocation

This example illustrates the importance of preallocation of arrays when using MAD.

Contents

Simple loop without preallocation

Because the array a continuously grows, the following loop executes slowly.

```
format compact
clear a
n=5000;
x=1;
a(1)=x;
tic; for i=1:n; a(i+1)=-a(i);end; t_noprealloc=toc
```

```
t_noprealloc =
    0.0563
```

Simple loop with preallocation

To improve performance we preallocate `a`*.*

```
clear a
a=zeros(n,1);
a(1)=x;
tic; for i=1:n;a(i+1)=-a(i);end;t_prealloc=toc
```

```
t_prealloc =
    0.0089
```

Fmad loop without preallocation

With an fmad variable performance is even more dramatically reduced when growing an array.

```
clear a
x=fmad(1,[1 2]);
a(1)=x;
tic;for i=1:n;a(i+1)=-a(i);end;t_fmadnoprealloc=toc
```

```
t_fmadnoprealloc =
   15.1634
```

Fmad loop with preallocation

To improve performance we preallocate `a`*.*

```
clear a
a=zeros(n,length(x));
a(1)=x;
tic; for i=1:n;a(i+1)=-a(i);end;t_fmadprealloc=toc


t_fmadprealloc =
    4.5518
```

Note that we have preallocated `a` using `zeros(n,length(x))`. In `fmad` we provide functions `length`, `size` and `numel` that return, instead of the expected integers, `fmad` object results with zero derivatives. This then allows for them to be used in functions such as `zeros` (here) or `ones` to preallocate `fmad` objects. Also note that it is not possible to change the class of the object that is being assigned to in MATLAB so if we wish to assign `fmad` variables to part of an matrix/array then that array must already be preallocated to be of `fmad` class. This technique has previously been used by Verma [Ver98a, Ver98c] in his ADMAT package.

Example 5.3 (Preallocating `fmad` Arrays)
In example 4.5 the input argument `y` *to the function will be of* `fmad` *class. In the third non-comment line,*

```
dydt = zeros(2*N,size(y,2));      % preallocate dy/dt
```

of the function of Figure 1 we see that `size(y,2)` *is used as an argument to* `zeros` *to preallocate storage for the variable* `dydt`*. Since* `y` *is of class* `fmad` *then so will be the result of the* `size` *function, and hence so will be the result of the* `zeros` *function and hence* `dydt` *will be of class* `fmad`*. This will then allow for* `fmad` *variables to be assigned to it and derivatives will be correctly propagated.*

We now see how to avoid problem 1 noted in Section 4.7.

Example 5.4 (Solution to Possible Problem 1)
We make use of `length`*, acting on an* `fmad` *variable to preallocate the array* `y` *to be of* `fmad` *class.*

```
>> x=fmad(1,1);
>> y=zeros(2,length(x(1)));
>> y(1)=x;
```

5.3 Dynamic Propagation of Sparse Derivatives

Many practical problems either possess sparse Jacobians or are partially separable so that many intermediate expressions have sparse local Jacobians. In either case it is possible to take advantage of this structure when calculating $\mathbf{F}'(\mathbf{x})\mathbf{V}$ in the cases when $\mathbf{V}$ is sparse. In particular using $\mathbf{V} = \mathbf{I}_n$, the identity matrix, facilitates calculation of the Jacobian $\mathbf{F}'(\mathbf{x})$. In order to do this we need to propagate the derivatives as sparse matrices. MATLAB matrices (though not N-D arrays) may be stored in an intrinsic `sparse` format for efficient handling of sparse **matrices** [GMS91]. Verma's ADMAT tool [Ver98a] can use MATLAB's sparse matrices to calculate Jacobians of functions which are defined only by scalars and vectors. In `fmad` careful design [For06] of the `derivvec` class for storage of derivatives has allowed us to propagate sparse derivatives for variables of arbitrary dimension (N-D arrays). In this respect `fmad` has a similar capability to the Fortran 90 overloading package ADO1 [PR98], and the Fortran source-text translation tool ADIFOR [BCH+98] which may make use of the SparseLinC library to form linear combinations of derivative vectors. See [Gri00, Chap. 6] for a review of this AD topic.

5.3.1 Use of Sparse Derivatives

In order to use sparse derivatives we initialise our independent variables with a sparse matrix. We use the feature of the `fmad` constructor (section 4.3) that the derivatives supplied must be of the correct number of elements and in column major order to avoid having to supply the derivative as an N-D array. As the following example shows, `getinternalderivs` must be used to access derivatives in their sparse storage format.

Example 5.5 Sparse1: Dynamic sparsity for Jacobians with fmad

This example illustrates use of dynamic sparsity when calculating Jacobians with fmad.

Contents

A function with sparse Jacobian

As a simple, small example we take,

```
y = F(x_1, x_2)=[ x_1^2 + x_2
                  x_2^2   ]
```

for which

```
DF/Dy = [ 2x_1  1
            0   2x_2]
```

and at (x_1,x_2)=(1,2)' we have,

```
DF/Dy = [ 2  1
          0  4]
```

Initialising sparse derivatives

We set the derivatives using the sparse identity matrix ***speye****. Note that fmad's* ***deriv*** *component stores the derivatives as a* ***derivvec*** *object with sparse storage as indicated by the displaying of only non-zero entries.*

```
x=fmad([1 2],speye(2))
```

```
value =
     1     2
Derivatives
Size = 1  2
No. of derivs = 2
d =
   (1,1)        1
d =
   (1,2)        1
```

Using sparse derivatives

We perform any calculations in the usual way - clearly **y** *has sparse derivatives.*

```
y=[x(1)^2+x(2); x(2)^2]
```

```
value =
     3
     4
Derivatives
Size = 2  1
No. of derivs = 2
d =
   (1,1)        2
d =
   (1,1)        1
   (2,1)        4
```

Extracting the derivatives

If we use **getderivs** *then, as usual MAD insists on returning a 3_D array and sparsity information is lost.*

```
dy=getderivs(y)
dy=reshape(dy,[2,2])
```

```
dy(:,:,1) =
     2
     0
dy(:,:,2) =
     1
     4
dy =
     2     1
     0     4
```

Accessing the internal derivative storage

By accessing the internal derivative storage used by `fmad` *in its calculation, the sparse Jacobian can be extracted.*

```
J=getinternalderivs(y)
```

```
J =
   (1,1)        2
   (1,2)        1
   (2,2)        4
```

The above example, though trivial, shows how sparse derivatives may be used in MAD. Of course there is an overhead associated with the use of sparse derivatives and the problem must be sufficiently large to warrant their use.

Example 5.6 MADEXbrussodeJacSparse: Brusselator Sparse Jacobian

This example shows how to use the `fmad` *class to calculate the Jacobian of the Brusselator problem using sparse propagation of derivatives. The compressed approache of* `MADEXbrussodeJacComp` *may be more efficient.*

See also: brussode_f, brussode_S, MADEXbrussodeJac, MADEXbrussodeJacComp, brussode

Contents

Initialise Variables

We define the input variables.

```
N=3;
t=0;
y0=ones(2*N,1);
```

Use fmad with sparse storage to calculate the Jacobian

We initialise `y` *with the value* `y0` *and derivatives given by the sparse identity matrix of size equal to the length of* `y0`*. We may then evaluate the function and extract the value and Jacobian.*

```
y=fmad(y0,speye(length(y0))); % Only significant change from MADEXbrussodeJac
F=brussode_f(t,y,N);
F_spfmad=getvalue(F); % grab value
JF_spfmad=getinternalderivs(F) % grab Jacobian
```

```
JF_spfmad =
   (1,1)      -2.6400
   (2,1)       1.0000
   (3,1)       0.3200
   (1,2)       1.0000
   (2,2)      -1.6400
   (4,2)       0.3200
   (1,3)       0.3200
   (3,3)      -2.6400
   (4,3)       1.0000
   (5,3)       0.3200
   (2,4)       0.3200
   (3,4)       1.0000
   (4,4)      -1.6400
   (6,4)       0.3200
   (3,5)       0.3200
   (5,5)      -2.6400
   (6,5)       1.0000
   (4,6)       0.3200
   (5,6)       1.0000
   (6,6)      -1.6400
```

Using finite-differencing

```
y=repmat(y0,[1 2*N+1]);
y(:,2:end)=y(:,2:end)+sqrt(eps)*eye(2*N);
F=brussode_f(t,y,N);
F_fd=F(:,1);
JF_fd=(F(:,2:end)-repmat(F_fd,[1 2*N]))./sqrt(eps)
```

```
JF_fd =
   -2.6400    1.0000    0.3200         0         0         0
    1.0000   -1.6400         0    0.3200         0         0
    0.3200         0   -2.6400    1.0000    0.3200         0
         0    0.3200    1.0000   -1.6400         0    0.3200
         0         0    0.3200         0   -2.6400    1.0000
         0         0         0    0.3200    1.0000   -1.6400
```

Comparing the AD and FD Jacobian

The function values computed are, of course, identical. The Jacobians disagree by about 1e-8, this is due to the truncation error of the finite-difference approximation.

```
disp(['Function values norm(F_fd-F_spfmad)  = ',num2str(norm(F_fd-F_spfmad,inf))])
disp(['Function Jacobian norm(JF_fd-JF_spfmad)= ',num2str(norm(JF_fd-JF_spfmad,inf))])
```

```
Function values norm(F_fd-F_spfmad)  = 0
Function Jacobian norm(JF_fd-JF_spfmad)= 2.861e-008
```

5.3.2 User Control of Sparse Derivative Propagation

Some AD tools, for example ADO1 [PR98], ADIFOR [BCH+98], allow for switching from sparse to full storage data structures for derivatives depending on the number of non-zeros in the derivative. There may be some user control provided for this. The sparse matrix implementation of MATLAB [GMS91] generally does not convert from sparse to full matrix storage unless a binary operation of a sparse and full matrix is performed. For example a sparse-full multiplication results in a full matrix. In the `derivvec` arithmetic used by `fmad` we presently provide no facilities for switching from sparse to full storage once objects have been created. Since MAD version 1.2, the `derivvec` class is designed to ensure that once calculations start with sparse derivatives, they continue with sparse derivatives. Any behaviour other than this is a bug and should be reported.

5.4 Sparse Jacobians via Compression

It is well known [Gri00, Chap. 7] that, subject to a favourable sparsity pattern of the Jacobian, the number of directional derivatives that need to be determined in order to form all entries of the Jacobian may be greatly reduced. This technique, known as *Jacobian (row) compression*, is frequently used in finite-difference approximations of Jacobians [DS96, Sec. 11.2], [NW99, p169-173]. Finite-differencing allows Jacobian-vector products to be approximated and can be used to build up columns, or linear combinations of columns, of the Jacobian. Use of sparsity and appropriate coloring algorithms frequently enables many columns to be approximated from one approximated directional derivative. This technique is frequently referred to as *sparse finite-differencing* however we shall use the term *compressed finite-differencing* to avoid confusion with techniques using sparse storage. The forward mode of AD calculates Jacobian-vector products exactly and, using the same colouring techniques as in compressed finite-differencing, may therefore calculate many columns of the Jacobian exactly by propagating just one directional derivative. Thus row compression may also be used by the `fmad` class of MAD.

Coleman and Verma [CV96] and, independently, Hossain and Steihaug [HS98] showed that since reverse mode AD may calculate vector-Jacobian products it may be used to calculate linear combinations of rows of the Jacobian. This fact is frequently utilised in optimisation in which an objective function with single output has its gradient, or row Jacobian calculated by the propagation of a single adjoint. These authors showed that by utilising both forward and reverse AD, and specialised colouring algorithms they could efficiently calculate sparse Jacobians which could not be calculated by sparse finite-differences. Such techniques are not available in MAD.

For a **given sparsity pattern** the row compression calculation proceeds as follows:

1. Calculate a good coloring from the given sparsity pattern using `MADcolor`.
2. Construct an appropriate seed matrix to initialise derivatives using `MADgetseed`.
3. Calculate the **compressed Jacobian** by using `fmad` to propagate derivatives, initialised by the seed matrix, through the function calculation to yield a compressed Jacobian.
4. The compressed Jacobian is uncompressed using `MADuncompressJac` to yield the required sparse Jacobian.

We use the Brusselator example again to illustrate this.

Example 5.7 MADEXbrussodeJacComp: Brusselator Compressed Jacobian

This example shows how to use the `fmad` *class to calculate the Jacobian of the Brusselator problem using compressed storage. If a good coloring can be found then this approach is usually more efficient than use of sparse or full storage as in* `MADEXbrussodeJac` *and* `MADEXbrussodeJacSparse`*.*

See also: brussode_f, brussode_S, MADEXbrussodeJac, MADEXbrussodeJacSparse, brussode

Contents

Initialise variables

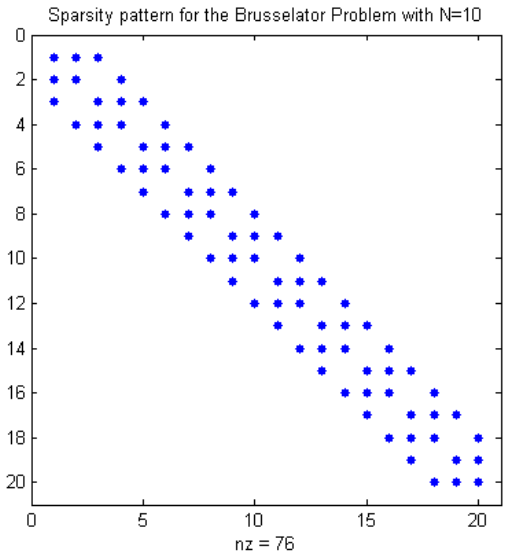
We define the input variables, setting N to 10 for this example.

```
format compact
N=10;
t=0;
y0=ones(2*N,1);
```

Obtain the sparsity pattern

The function brussode_S returns the sparsity pattern, the matrix with an entry 1 when the corresponding Jacobian entry is nonzero and entry 0 when the corresponding Jacobian entry is always zero.

```
sparsity_pattern=brussode_S(N);
figure(1)
spy(sparsity_pattern)
title(['Sparsity pattern for the Brusselator Problem with N=',num2str(N)])
```



Determine the coloring

The function MADcolor takes the sparsity pattern and determines a color (or group number) for each component of the input variables so that perturbing one component with that color affects different dependent variables to those affected by any other variable in that group.

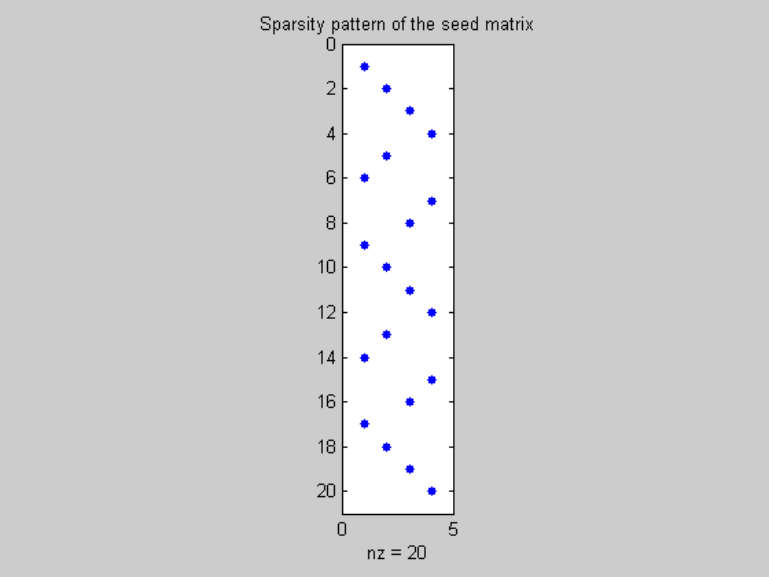
```
color_groups=MADcolor(sparsity_pattern);
ncolors=max(color_groups) % number of colors used
```

```
ncolors =
     4
```

Determine the seed matrix

The coloring is used by `MADgetseed` *to determine a set of* `ncolors` *directions whose directional derivatives may be used to construct all Jacobian entries.*

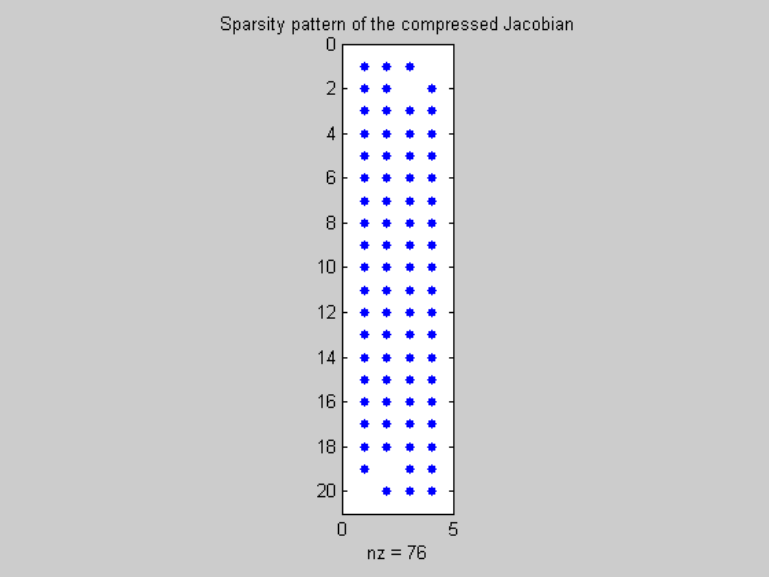
```
seed=MADgetseed(sparsity_pattern,color_groups);
figure(2);
spy(seed);
title('Sparsity pattern of the seed matrix')
```



Calculate the compressed Jacobian using fmad

We initialise y with the value y0 and derivatives given by the seed matrix. We then evaluate the function and extract the value and compressed Jacobian.

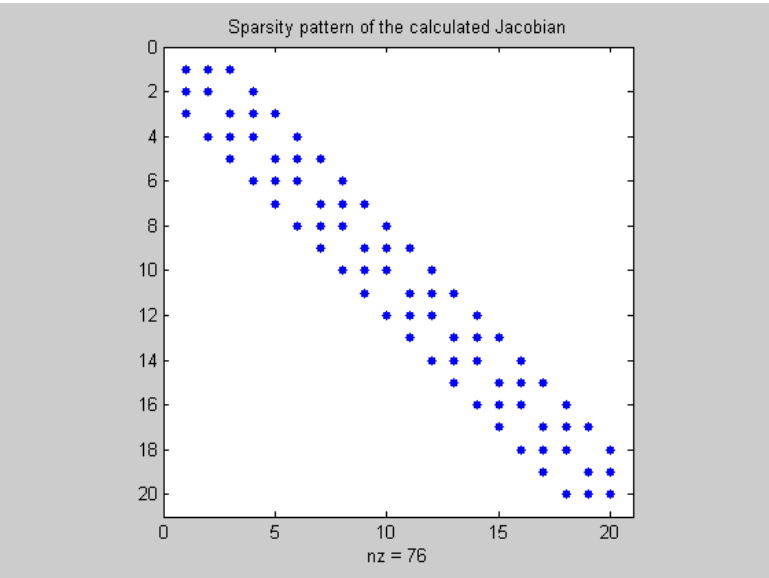
```
y=fmad(y0,seed);
F=brussode_f(t,y,N);
F_compfmad=getvalue(F); % grab value
Jcomp=getinternalderivs(F); % grab compressed Jacobian
figure(3);
spy(Jcomp);
title('Sparsity pattern of the compressed Jacobian')
```



Uncompress the Jacobian

`MADuncompressJac` is used to extract the Jacobian from its compressed form.

```
JF_compfmad=MADuncompressJac(Jcomp,sparsity_pattern,color_groups);
figure(4);
spy(JF_compfmad);
title('Sparsity pattern of the calculated Jacobian')
```



Using finite-differencing

```
y=repmat(y0,[1 2*N+1]);
y(:,2:end)=y(:,2:end)+sqrt(eps)*eye(2*N);
F=brussode_f(t,y,N);
F_fd=F(:,1);
JF_fd=(F(:,2:end)-repmat(F_fd,[1 2*N]))./sqrt(eps);
```

Comparing the AD and FD Jacobians

The function values computed are, of course, identical. The Jacobians disagree by about 1e-8, this is due to the truncation error of the finite-difference approximation.

```
disp(['Function values norm(F_fd-F_compfmad)  = ',num2str(norm(F_fd-F_compfmad,inf))])
disp(['Function Jacobian norm(JF_fd-JF_compfmad)= ',num2str(norm(JF_fd-JF_compfmad,inf))])
```

```
Function values norm(F_fd-F_compfmad)  = 0
Function Jacobian norm(JF_fd-JF_compfmad)= 4.2915e-008
```

Using compressed finite-differencing

We may also use compression with finite-differencing.

```
y=repmat(y0,[1 ncolors+1]);
y(:,2:end)=y(:,2:end)+sqrt(eps)*seed;
F=brussode_f(t,y,N);
F_compfd=F(:,1);
Jcomp_fd=(F(:,2:end)-repmat(F_fd,[1 ncolors]))./sqrt(eps);
JF_compfd=MADuncompressJac(Jcomp_fd,sparsity_pattern,color_groups);
```

Comparing the AD and compressed FD Jacobians

The function values computed are, of course, identical. The Jacobians disagree by about 1e-8, this is due to the truncation error of the finite-difference approximation.

```
disp(['Function values norm(F_compfd-F_compfmad)  = ',num2str(norm(F_compfd-F_compfmad,inf))])
disp(['Function Jacobian norm(JF_compfd-JF_compfmad)= ',num2str(norm(JF_compfd-JF_compfmad,inf))])
```

```
Function values norm(F_compfd-F_compfmad)  = 0
Function Jacobian norm(JF_compfd-JF_compfmad)= 4.2915e-008
```

5.5 Differentiating Iteratively Defined Functions

Consider a function $\mathbf{x} \rightarrow \mathbf{y}$ defined implicitly as a solution of,

$$\mathbf{g}(\mathbf{y}, \mathbf{x}) = 0, \tag{1}$$

and for which we require $\partial \mathbf{y} / \partial \mathbf{x}$. Such systems are usually solved via some fixed point iteration of the form,

$$\mathbf{y}^{n+1} = \mathbf{h}(\mathbf{y}^n, \mathbf{x}). \tag{2}$$

There have been many publications regarding the most efficient way to automatically differentiate such a system [Azm97, BB98b, Chr94, Chr98] and reviewed in [Gri00]. A simple approach termed *piggy-back* [Gri00], is to use forward mode AD through the iterative process of equation 2. It is necessary to check for convergence of both the value and derivatives of $\mathbf{y}$.

Example 5.8 MADEXBBEx3p1 - Bartholomew-Biggs Example 3.1 [BB98a]

This example demonstrates how to find the derivatives of a function defined by a fixed-point iteration using `fmad`*. The example is taken from (M.C. Bartholomew-Biggs, "Using Forward Accumulation for Automatic Differentiation of Implicitly-Defined Functions", Computational Optimization and Applications 9, 65-84, 1998). The example concerns solution of*

$$g_1 = y_1 - u_1 + u_3 * y_1 * \sqrt{y_1^2 + y_2^2} = 0$$

$$g_2 = y_2 - u_2 + u_3 * y_2 * \sqrt{y_1^2 + y_2^2} = 0$$

with

$$u_1 = x_1 * cos(x_2), u_2 = x_1 * sin(x_2), u_3 = x_3$$

via the fixed-point iteration,

$$y_{new} = y_{old} - g(y_{old}).$$

See also BBEx3p1_gfunc

Contents

Setting up the iteration

We set the initial value `y0`*, the value of the independent variable* `x`*, the tolerance* `tolg` *and the maximum number of iterations itmax.*

```
format compact
y0=[0; 0];
x0=[0.05; 1.3; 1]
tolg=1e-8;
itmax=20;
```

```
x0 =
    0.0500
    1.3000
    1.0000
```

Analytic solution

An analytic solution is easily found by setting,

$$y_1 = r\cos\theta, y_2 = r\sin\theta,$$

giving

$$\theta = x_2, r = \frac{-1+R}{2x_3}, R = \sqrt{1+4x_1x_2},$$

which may be differentiated to give the required sensitivities.

```
x=x0;
theta=x(2);
R=sqrt(1+4*x(3)*x(1));
```

```
r=(-1+R)/(2*x(3));
y_exact=[r*cos(theta); r*sin(theta)]
dthetadx=[0, 1, 0];
dRdx=[2*x(3)/R, 0, 2*x(1)/R];
drdx=dRdx/(2*x(3))-(-1+R)/(2*x(3)^2)*[0,0,1];
Dy_exact=[-r*sin(theta)*dthetadx+cos(theta)*drdx; ...
    r*cos(theta)*dthetadx+sin(theta)*drdx]
```

```
y_exact =
    0.0128
    0.0460
Dy_exact =
    0.2442   -0.0460   -0.0006
    0.8796    0.0128   -0.0020
```

Undifferentiated iteration

For the nonlinear, undifferentiated iteration we have:

```
y=y0;
x=x0;
u=[x(1)*cos(x(2));x(1)*sin(x(2));x(3)];
disp('Itns. |g(y)|  ')
for i=1:itmax
    g=BBEx3p1_gfunc(y,u);
    y=y-g;
    disp([num2str(i,'% 4.0f'),' ',num2str(norm(g),'%10.4e')])
    if norm(g)<=tolg
        break
    end
end
y_undif=y
Error_y_undif=norm(y_undif-y_exact,inf)
```

```
Itns. |g(y)|
1 5.0000e-002
2 2.5000e-003
3 2.4375e-004
4 2.3216e-005
5 2.2163e-006
6 2.1153e-007
7 2.0189e-008
8 1.9270e-009
y_undif =
    0.0128
    0.0460
Error_y_undif =
  1.6178e-010
```

Naive differentiation

Simply initialising x with the `fmad` *class gives a naive iteration for which the lack of convergence checks on the derivatives here results in only slight inaccuracy.*

```
y=y0;
x=fmad(x0,eye(length(x0)));
u=[x(1)*cos(x(2));x(1)*sin(x(2));x(3)];
disp('Itns. |g(y)|  ')
for i=1:itmax
    g=BBEx3p1_gfunc(y,u);
    y=y-g;
    disp([num2str(i,'% 4.0f'),' ',num2str(norm(g),'%10.4e'),' ',...
        num2str(norm(getinternalderivs(g)),'%10.4e')])
    if norm(g)<=tolg
        break
    end
end
y_naive=getvalue(y)
Dy_naive=getinternalderivs(y)
Error_y_naive=norm(y_naive-y_exact,inf)
Error_Dy_naive=norm(Dy_naive-Dy_exact,inf)
```

```
Itns. |g(y)|
1 5.0000e-002 1.0000e+000
2 2.5000e-003 1.0003e-001
3 2.4375e-004 1.4508e-002
4 2.3216e-005 1.8246e-003
5 2.2163e-006 2.1665e-004
6 2.1153e-007 2.4729e-005
7 2.0189e-008 2.7469e-006
8 1.9270e-009 2.9909e-007
y_naive =
    0.0128
    0.0460
Dy_naive =
    0.2442   -0.0460   -0.0006
    0.8796    0.0128   -0.0020
Error_y_naive =
  1.6178e-010
Error_Dy_naive =
  2.9189e-008
```

Problems with naive differentiation

A major problem with naive iteration is if we start close to the solution of the nonlinear problem - then too few differentiated iterations are performed to converge the derivatives. Below only one iteration is performed and we see a large error in the derivatives.

```
y=y_naive;
x=fmad(x0,eye(length(x0)));
u=[x(1)*cos(x(2));x(1)*sin(x(2));x(3)];
disp('Itns. |g(y)|  ')
for i=1:itmax
    g=BBEx3p1_gfunc(y,u);
    y=y-g;
    disp([num2str(i,'% 4.0f'),' ',num2str(norm(g),'%10.4e'),' ',...
        num2str(norm(getinternalderivs(g)),'%10.4e')])
    if norm(g)<=tolg
        break
    end
end
y_naive2=getvalue(y)
Dy_naive2=getinternalderivs(y)
Error_y_naive2=norm(y_naive2-y_exact,inf)
Error_Dy_naive2=norm(Dy_naive2-Dy_exact,inf)
```

```
Itns. |g(y)|
1 1.8392e-010 1.0000e+000
y_naive2 =
    0.0128
    0.0460
Dy_naive2 =
    0.2675   -0.0482   -0.0006
    0.9636    0.0134   -0.0022
Error_y_naive2 =
  1.5441e-011
Error_Dy_naive2 =
    0.0848
```

Adding a derivative convergence test

The naive differentiation may be improved by adding a convergence test for the derivatives.

```
y=y_naive;
x=fmad(x0,eye(length(x0)));
u=[x(1)*cos(x(2));x(1)*sin(x(2));x(3)];
disp('Itns. |g(y)|  ')
for i=1:itmax
    g=BBEx3p1_gfunc(y,u);
    y=y-g;
    disp([num2str(i,'% 4.0f'),' ',num2str(norm(g),'%10.4e'),' ',...
        num2str(norm(getinternalderivs(g)),'%10.4e')])
    if norm(g)<=tolg & norm(getinternalderivs(g))<=tolg % Test added
        break
    end
end
```

```
y_improved=getvalue(y)
Dy_improved=getinternalderivs(y)
Error_y_improved=norm(y_improved-y_exact,inf)
Error_Dy_improved=norm(Dy_improved-Dy_exact,inf)
```

```
Itns. |g(y)|
1 1.8392e-010 1.0000e+000
2 1.7554e-011 9.5445e-002
3 1.6755e-012 9.1098e-003
4 1.5992e-013 8.6949e-004
5 1.5264e-014 8.2988e-005
6 1.4594e-015 7.9208e-006
7 1.4091e-016 7.5600e-007
8 1.6042e-017 7.2157e-008
9 4.4703e-019 6.8870e-009
y_improved =
    0.0128
    0.0460
Dy_improved =
    0.2442   -0.0460   -0.0006
    0.8796    0.0128   -0.0020
Error_y_improved =
  4.1633e-017
Error_Dy_improved =
  5.7952e-010
```

Conclusions

Simple fixed-point iterations may easily be differentiated using MAD's `fmad` *class though users should add convergence tests for derivatives to ensure they, as well as the original nonlinear iteration, have converged. Note that* `getinternalderivs` *returns an empty array, whose norm is 0, for variables of class double, so adding such tests does not affect the iteration when used for such variables.*

5.6 User Control of Activities/Dependencies

Example 5.9 ***MADEXDepend1: User control of activities/dependencies***

This example shows how a user may modify their code to render inactive selective variables which MAD automatically assumes are active.

See also: depend, depend2

Contents

Function depend

The function **depend** *defines two variables u,v in terms of independents x,y,*

$$u = x + y, v = -x$$

Using the fmad class it is trivial to calculate the derivatives **du/dx**, **du/dy**, **dv/dx**, **dv/dy**. *We associate the first directional derivative with the* **x***-derivatives and the second with those of variable* **y**.

```
format compact
x=fmad(1,[1 0]); y=fmad(2,[0 1]);
[u,v]=depend(x,y);
Du=getinternalderivs(u);
du_dx=Du(1)
du_dy=Du(2)
Dv=getinternalderivs(v);
dv_dx=Dv(1)
dv_dy=Dv(2)
```

```
du_dx =
     1
du_dy =
     1
dv_dx =
    -1
dv_dy =
     0
```

One independent variable

Dealing with one independent, say x, is trivial and here requires just one directional derivative.

```
x=fmad(1,1); y=2;
[u,v]=depend(x,y);
du_dx=getinternalderivs(u)
dv_dx=getinternalderivs(v)
```

```
du_dx =
     1
dv_dx =
    -1
```

Dealing with one dependent variable

When we are only interested in the derivatives of one dependent variable, say u, then things are more tricky since ***fmad*** *automatically calculates derivatives of all variables that depend on active inputs. We could, of course, calculate v's derivatives and simply discard them and in many cases (including that here) the performance penalty paid would be insignificant. In some cases a calculation may be significant shortened by ensuring that we calculate derivatives only for those dependent variables we are interested in, or which are involved in calculating those dependents. To illustrate this see function* ***depend2*** *in which we ensure* ***v*** *is not active by using MAD's* ***getvalue*** *function to ensure that only* ***v****'s value and not its derivatives, is calculated. u=x+y; v=-getvalue(x); Note that in this case an empty matrix is returned by* ***getinternalderivatives****.*

```
x=fmad(1,[1 0]); y=fmad(2,[0 1]);
[u,v]=depend2(x,y);
Du=getinternalderivs(u);
du_dx=Du(1)
du_dy=Du(2)
Dv=getinternalderivs(v)
```

```
du_dx =
     1
du_dy =
     1
Dv =
     []
```

The last case of just one active output is more difficult to deal with because derivatives are propagated automatically by `fmad`'s overloaded library. Automatically dealing with a restricted number of active outputs requires a reverse dependency or activity analysis [Gri00]. This kind of analysis requires compiler based techniques and is implemented in source transformation AD tools such as ADIFOR, TAF/TAMCand Tapenade. In the present overloading context such an effect can only be achieved as above by the user **carefully** ensuring that unnecessary propagation of derivatives is stopped by judicious use of the `getvalue` function to propagate values and not derivatives. Of course such coding changes as those shown above, should only be performed if the programmer is sure they know what they are doing. Correctly excluding some derivative propagation and hence calculations may make for considerable run-time savings for large codes. However incorrect use will result in incorrectly calculated derivative values.

5.7 Adding Functions to the `fmad` Class

Because MATLAB has such a large, and growing, range of builtin functions it is not possible to commit to providing differentiated `fmad` class functions for them all. Our strategy is to add new functions as requested by users; most such requests are dealt with within a working week (depending on staff availability and complexity of the task). Some users may wish to add differentiated functions to the `fmad` class themselves, this section describes how this may be done. We request that users who do extend MAD in such a way forward their `fmad` class functions to TOMLAB so that they may be tested, optimised, and added to the MAD distribution to the benefit of all users.

In order for users to extend the `fmad` class they must add functions directly to the `@fmad` folder of the MAD distribution. In order to simplify this, and since MAD is distributed in p-code format preventing users looking at existing code, a non-p-coded *template* function `@fmad/userfunc.m` (shown in Figure 2) is included in the MAD distribution for users to modify. To illustrate how a user can create their own `fmad` functions consider the example

```
function y=userfunc(x)
% userfunc: y=userfunc(x) taemplate for user-defined function of fmad variable
%
% See also

y=x;      % deep copy of x so y has x's class and components
y.value= ; % add value here written in terms of x.value
y.deriv=.*x.deriv; % add multiplier of x's derivative written
                   % in terms of x.value and perhaps y.value
                   % to give y's derivative
```

Figure 2: The template function `userfunc.m` that may be copied and modified to extend MAD's functionality. Some comments have been omitted for brevity

of `y=sinh(x)`[1]. First the user would make a copy of `userfunc.m` named `sinh.m` in the `fmad` class folder `@fmad` of their installed MAD distribution. They would then modify their function `sinh.m` to be as in Figure 3. We see

```
function y=sinh(x)
% sinh: y=sinh(x) hyperbolic sine of fmad variable
%
% See also sinh, fmad/asinh, fmad/cosh, fmad/tanh

y=x;
y.value=sinh(x.value);
y.deriv=cosh(x.value).*x.deriv;
```

Figure 3: A possible user created `sinh.m` function for the `fmad` class created from the template `userfunc.m` of Figure 2

that just three changes have been made:

1. The function name has been changed appropriately to `sinh`.
2. The `value` component of the function's output `y.value` has been set to a value calculated using the value component of the function's input `x.value`. In this case the output's `value` component is the `sinh` of the input's `value` component.
3. The `deriv` component of the function's output `y.deriv` has been set to a value calculated using the value component of the function's input `x.value` multiplied by the `deriv` component of the function's input `x.deriv`. In this case the output's `deriv` component is `cosh(x.value)` times the function's input `derivs` component because the derivative of $\sinh(x)$ is $\cosh(x)$. In all cases the multiplier of `x.deriv` is the *local derivative* of the function, i.e., the derivative of the function's output with respect to its input in the mathematical analysis sense.

The user should then test their coding on simple problems for which they know the solution. For example, in the `sinh(x)` case above we might consider the derivatives when `x` is the vector `x=[ 1 2 ]`. Then to get derivatives of the output with respect to both components of `x` we would perform the following:

```
>> x=[1 2]
```

[1] `sinh(x)` is already part of the `fmad` class.

```
x =
     1     2
>> xfmad=fmad(x,eye(length(x)))
fmad object xfmad
value =
     1     2
derivvec object derivatives
Size = 1  2
No. of derivs = 2
derivs(:,:,1) =
     1     0
derivs(:,:,2) =
     0     1
>> yfmad=sinh(xfmad)
fmad object yfmad
value =
    1.1752    3.6269
derivvec object derivatives
Size = 1  2
No. of derivs = 2
derivs(:,:,1) =
    1.5431         0
derivs(:,:,2) =
         0    3.7622
>> yvalue=getvalue(yfmad)
yvalue =
    1.1752    3.6269
>> y=sinh(x)
y =
    1.1752    3.6269
>> Dy=getinternalderivs(yfmad)
Dy =
    1.5431         0
         0    3.7622
>> derivy=cosh(x)
derivy =
    1.5431    3.7622
```

in which we see that the calculated value of `y`'s value using `fmad` is `yvalue=[1.1752 3.6269]` agreeing with the directly calculated `y=sinh(x)`; and the calculated value of `y`'s derivatives,

```
Dy =
    1.5431         0
         0    3.7622
```

agrees with the analytic derivative `derivy=cosh(x)`[2].

As stated before, users should send their coding of such `fmad` functions to TOMLAB for performance optimisation and incorporation into the general distribution.

[2]Only the derivatives $\partial y_1/\partial x_1$ and $\partial y_2/\partial x_2$ have been calculated analytically while the zero derivatives $\partial y_1/\partial x_2$ and $\partial y_2/\partial x_1$ have been additionally calculated by `fmad`.

6 High-Level Interfaces to Matlab's ODE and Optimization Solvers

In many cases users wish to use AD for calculating derivatives necessary for ODE or optimisation solvers. When using the TOMLAB optimisation solvers the use of AD is easily achieved by setting the flags, *Prob.NumDiff* and *Prob.ConsDiff* as noted in Section 3. When solving stiff ODE's using MATLAB's ODE solvers or solving optimization problems using MATLAB's Optimization Toolbox functions the user can either:

- Directly use calls to MAD's functions to calculate required derivatives as detailed in Sections 4-5.
- Or use the facilities of MAD's high-level interfaces [FK04] to the ODE and optimization solvers as detailed in this section.

For a vector-valued function $\mathbf{F}(\mathbf{x})$, $\mathbf{x} \in I\!R^n$, $\mathbf{F}(\mathbf{x}) \in I\!R^m$ MAD currently provides 3 techniques for computing the Jacobian $\mathbf{JF}$,

Forward mode AD with full storage (see Section 4) - this is most likely to be efficient when n is small.

Forward mode AD with sparse storage (see Section 5.3) - this is most likely to be efficient for larger n, particularly if the Jacobian is sparse. Even if the Jacobian is dense, then provided n is sufficiently large this method is likely to outperform full storage because there is always sparsity present in the early stages of a Jacobian calculation. For small n this technique is likely to be slower than full storage because of the overhead of manipulating a sparse data structure.

Forward mode AD with compressed storage (see Section 5.4) - this is likely to be most efficient for large n when the Jacobian is sparse and has a favourable sparsity pattern (e.g., diagonal, banded) in which case the number of directional derivatives required to compute the Jacobian may be reduced to $p < n$ with p dependent on the sparsity pattern and bounded below by the maximum number of nonzeros in a row of the Jacobian. Note that sparse storage always uses fewer floating-point operations than compressed storage - but provided p is sufficiently small then compression outperforms sparse storage. This method requires the sparsity pattern of the Jacobian (or a safe over-estimate of the sparsity pattern) to be known. If the sparsity pattern is unknown but is known to be fixed then the sparsity pattern may be estimated using a Jacobian calculation with sparse storage and randomly perturbed $\mathbf{x}$ (to reduce the chance of unfortuitous cancellation giving a Jacobian entry of zero when in general it is non-zero). If the sparsity pattern changes between evaluations of $\mathbf{F}$ (perhaps because of branching) then either compression should not be used or the supplied Jacobian pattern must be an over-estimate to cover any possible pattern.

In general it is not possible to predict a priori which technique will be most efficient for any particular problem since this depends on many factors: sparsity pattern of $\mathbf{JF}$; relative size of n, m and when compression is possible p; efficiency of sparse data access; performance of the computer being used. A rational approach is to try each technique (except default to full storage for small n and reject full storage for large n), timing its evaluation and choosing the most efficient. To make this easy for the user this approach has been implemented within the function `MADJacInternalEval` and interfaces to `MADJacInternalEval` for ODE or optimisation solvers have been provided. Consequently user's need never access `MADJacInternalEval` directly and needn't bother themselves with the exact AD technique being used confident that a reasonably efficient technique for their problem has been automatically selected. Of course, this selection process has an associated overhead so the interface allows the user to mandate a technique if they so wish, perhaps based on experiments conducted previously.

6.1 Stiff ODE Solution

When solving the ordinary differential equation (ODE)

$$\frac{d\mathbf{y}}{dt} = \mathbf{F}(t, \mathbf{y}, p_1, p_2, \ldots), \ \mathbf{y}(t=0) = \mathbf{y}_0,$$

for $\mathbf{y} \in I\!R^n$ where $p_1, p_2, \ldots$ are fixed parameters using MATLAB's stiff [3] ODE solvers `ode15s` , `ode23s`, `ode23t`, and `ode23tb` then it is necessary to calculate, or at least approximate accurately, the Jacobian,

$$\mathbf{JF}(t, \mathbf{y}, p_1, p_2, \ldots) = \frac{\partial}{\partial \mathbf{y}} \mathbf{F}(t, \mathbf{y}, p_1, p_2, \ldots),$$

for the solvers to use a quasi-Newton solve within their implicit algorithms. By default such Jacobians are approximated by finite-differences. If the Jacobian is sparse and the sparsity pattern supplied then Jacobian compression is used [SR97] (c.f. Section 5.4). Using `MADODEJacEval` we may easily take advantage of AD.

Example 6.1 MADEXvdpode: Jacobian driver for the Van der Pol ODE.

This example shows how to use MAD's ODE Jacobian function `MADJacODE` to solve the stiff Van der Pol ODE problem.

See also: vdpode_f, vdpode_J, vdpode, ode15s, MADODERegister, MADODEJacEval, MADODEReport

Contents

Set up the ODE problem

First we define:

```
MU = 1000;     % default stiffness parameter for Van der Pol problem
tspan = [0; max(20,3*MU)]; % integration timespan of several periods
y0 = [2; 0]; % initial conditions
options=odeset('Stats','on'); % display ODE statistics
```

Integrate using finite-differencing for the Jacobian

Using the default finite-differencing a solution is easily found.

```
t1=cputime;
[t,y] = ode15s(@vdpode_f,tspan,y0,options,MU);
tfd=cputime-t1;
disp(['Jacobian by Finite-Differencing            - CPU time = ',num2str(tfd)])
```

[3] An ODE is stiff when the eigenvalues λ_i of the Jacobian matrix $\partial \mathbf{F}/\partial \mathbf{y}$ are such that $\text{real}(\lambda_i) < 0$ for all i and $\max |\text{real}(\lambda_i)| \gg \min |\text{real}(\lambda_i)|$ [Dor96, p. 125].

```
591 successful steps
225 failed attempts
1883 function evaluations
45 partial derivatives
289 LU decompositions
1747 solutions of linear systems
Jacobian by Finite-Differencing            - CPU time = 0.75108
```

Use the analytic Jacobian

Since the Jacobian is recalculated many times supplying an analytic Jacobian should improve performance.

```
options = odeset(options,'Jacobian',@vdpode_J);
t1=cputime;
[t,y] = ode15s(@vdpode_f,tspan,y0,options,MU);
tjac=cputime-t1;
disp(['Jacobian by hand-coding                    - CPU time = ',num2str(tjac)])
```

```
591 successful steps
225 failed attempts
1749 function evaluations
45 partial derivatives
289 LU decompositions
1747 solutions of linear systems
Jacobian by hand-coding                    - CPU time = 0.75108
```

Use the MAD Jacobian function MADODEJacEval

The automated MAD Jacobian calculation avoids the need for the user to supply the Jacobian. First we use `odeset` *to specify that the Jacobian will be calculated by* `MADODEJacEval`*. We then use* `MADODEDelete` *to first clear any existing settings for* `vdpode_f`*, then* `MADODERegister` *register the function and then integrate.*

```
options = odeset(options,'Jacobian',@MADODEJacEval);
MADODEDelete; % deletes all previous settings for ODE evaluation
MADODERegister(@vdpode_f); % all we must do is supply the ODE function handle
t1=cputime;
[t,y] = ode15s(@vdpode_f,tspan,y0,options,MU);
tmadjac=cputime-t1;
disp(['Jacobian by MADODEJacEval                      - CPU time = ',num2str(tmadjac)])
```

```
591 successful steps
225 failed attempts
1749 function evaluations
45 partial derivatives
289 LU decompositions
1747 solutions of linear systems
Jacobian by MADODEJacEval                     - CPU time = 0.88127
```

Repeat of MADODEJacEval use

*On a second use (**without** calling `MADODEDelete`) `MADODEJacEval` should be slightly faster since it "remembers" which technique to use to calculate the Jacobian.*

```
t1=cputime;
[t,y] = ode15s(@vdpode_f,tspan,y0,options,MU);
tmadjac2=cputime-t1;
disp(['Repeat of Jacobian by MADODEJacEval            - CPU time = ',num2str(tmadjac2)])
```

```
591 successful steps
225 failed attempts
1749 function evaluations
45 partial derivatives
289 LU decompositions
1747 solutions of linear systems
Repeat of Jacobian by MADODEJacEval            - CPU time = 0.91131
```

MADODEReport

Using `MADODEReport` we see which technique is used and why – here full storage is used due to the small size (2×2) of the Jacobian required.

```
MADODEReport % report on which technique used
```

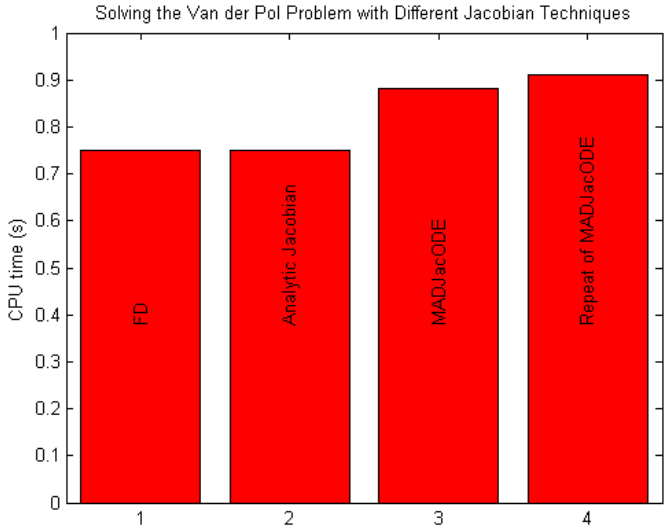
```
MADODEreport
Function name vdpode_f
Jacobian of size 2x2
Fwd AD - full       0.010014 s
Fwd AD - compressed Inf s
FWD AD - full selected
MADJacInternalEval: n< MADMinSparseN=10 so using forward mode full
```

Conclusions

Using MAD for such small ODE problems gives similar performance to finite-differencing. MAD is more useful for large ODE problems, particularly those with sparse Jacobians in, for example the Brusselator problem of `MADEXbrussode`.

```
bar([tfd,tjac,tmadjac,tmadjac2])
colormap hsv
ylabel('CPU time (s)')
title('Solving the Van der Pol Problem with Different Jacobian Techniques')
text(1,tfd/2,'FD','Rotation',90)
```

```
text(2,tfd/2,'Analytic Jacobian','Rotation',90)
text(3,tfd/2,'MADJacODE','Rotation',90)
text(4,tfd/2,'Repeat of MADJacODE','Rotation',90)
```



Using MAD for such small ODE problems gives similar performance to finite-differencing. MAD is more useful for large ODE problems, particularly those with sparse Jacobians in, for example the Brusselator problem of `MADEXbrussode`.

Example 6.2 MADEXbrussode: Jacobian driver for the Brusselator ODE.

This example of `MADEXbrussode` *shows how to use MAD's high-level ODE Jacobian function* `MADODEJacEval` *to solve the stiff ODE Brusselator problem.*

See also: brussode_f, brussode_S, brussode, ode15s, MADODERegister, MADODEFuncEval, MADODEJacEval, MADODEReport

Contents

Set up the ODE problem

First we define:

```
N = 80; % number of mesh points,
tspan = [0; 50]; % timespan of integration,
t=tspan(1);      % initial time,
y0 = [1+sin((2*pi/(N+1))*(1:N)); repmat(3,1,N)];
y0=reshape(y0,[2*N 1]); % initial conditions. and sparsity pattern
S=brussode_S(N); % find sparsity pattern.
```

Integrate Brusselator problem using finite-differencing for the Jacobian

Initially we use finite-differences to approximate the Jacobian ignoring any sparsity, timing and running the calculation.

```
options = odeset('Vectorized','on','Stats','on');
t1=cputime;
[tfd,yfd] = ode15s(@brussode_f,tspan,y0,options,N);
cpufd=cputime-t1;
disp(['FD-Jacobian              : cputime = ',num2str(cpufd)])
```

```
339 successful steps
54 failed attempts
2868 function evaluations
13 partial derivatives
110 LU decompositions
774 solutions of linear systems
FD-Jacobian              : cputime = 2.0329
```

Integrate Brusselator problem using compressed finite-differencing

Finite-differencing can take advantage of sparsity when approximating the Jacobian. We set options to: specify the Jacobian sparsity pattern, print out statistics and to turn on vectorization. We then time and run the calculation.

```
options = odeset('Vectorized','on','JPattern',S,'Stats','on');
t1=cputime;
[tfd,yfd] = ode15s(@brussode_f,tspan,y0,options,N);
cpucompfd=cputime-t1;
disp(['Compressed FD-Jacobian   : cputime = ',num2str(cpucompfd)])
```

```
339 successful steps
54 failed attempts
840 function evaluations
```

```
13 partial derivatives
110 LU decompositions
774 solutions of linear systems
Compressed FD-Jacobian   : cputime = 0.91131
```

Use the MAD ODE Jacobian function - MADODEJacEval

First delete any existing settings for MADODE using `MADODEDelete`*. Then register function* `brussode_f` *for* `MADODEJacEval` *using* `MADODERegister`*. Then use* `odeset` *to specify that* `MADODEJacEval` *will calculate the required Jacobian. Finally time and run the calculation and use* `MADODEReport` *to provide information on the Jacobian technique used.*

```
MADODEDelete % deletes any internal settings
MADODERegister(@brussode_f)               % supplies handle to ODE function
options=odeset(options,'Jacobian',@MADODEJacEval);
t1=cputime;
[tmadjac,ymadjac]=ode15s(@brussode_f,tspan,y0,options,N);
cpumadjac=cputime-t1;
disp(['MAD Jacobian       : cputime = ',num2str(cpumadjac)])
MADODEReport
```

```
339 successful steps
54 failed attempts
776 function evaluations
13 partial derivatives
110 LU decompositions
774 solutions of linear systems
MAD Jacobian       : cputime = 1.1416
MADODEreport
Function name brussode_f
Jacobian of size 160x160
Fwd AD - full       Inf s
Fwd AD - sparse     0.020029 s
Fwd AD - compressed Inf s
FWD AD - sparse selected
Sparsity not fixed so compression rejected. Large system reject full method.
```

Specifying that the sparsity pattern is fixed

For this problem the Jacobian is large and sparse but MAD does not "know" that the sparsity pattern is fixed and so can only use sparse forward mode and not compression. We first delete any internal settings, specify that the sparsity pattern is fixed, i.e., it does not change from one evaluation to the next due to branching; and then use the MAD Jacobian. The Jacobian sparsity pattern is calculated using fmad's sparse forward mode for a point randomly perturbed about `y0`*.*

```
MADODEDelete % delete any internal settings
MADODERegister(@brussode_f,'sparsity_fixed','true')% specify fixed sparsity
options=odeset(options,'Jacobian',@MADODEJacEval); % Use MAD Jacobian
t1=cputime;
[tmadjac,ymadjac]=ode15s(@brussode_f,tspan,y0,options,N);
cpumadjac_fixed=cputime-t1;
disp(['MAD Jacobian (sparsity fixed): cputime = ',num2str(cpumadjac_fixed)])
MADODEReport
% Compressed forward mode should now have been used.
```

```
339 successful steps
54 failed attempts
776 function evaluations
13 partial derivatives
110 LU decompositions
774 solutions of linear systems
MAD Jacobian (sparsity fixed): cputime = 1.1416
MADODEreport
Function name brussode_f
Jacobian of size 160x160
Fwd AD - full       Inf s
Fwd AD - sparse     0.020029 s
Fwd AD - compressed 0.020029 s
Percentage Sparsity    = 2.4844%
Maximum non-zeros/row  = 4 of 160
Maximum non-zeros/col  = 4 of 160
Size of compressed seed = 160x4
Percentage compressed to full seed    = 2.5%
FWD AD - compressed selected
Large system reject full method. Sparse is worse than compressed - discarding Sparse.
```

Specifying the sparsity pattern

If available the sparsity pattern can be specified directly within `MADsetupODE`.

```
MADODEDelete % delete any internal settings
MADODERegister(@brussode_f,'sparsity_pattern',S)
options=odeset(options,'Jacobian',@MADODEJacEval);
t1=cputime;
[tmadjac,ymadjac]=ode15s(@brussode_f,tspan,y0,options,N);
cpumadjac_S=cputime-t1;
disp(['MAD Jacobian (sparsity given) : cputime = ',num2str(cpumadjac_S)])
MADODEReport
```

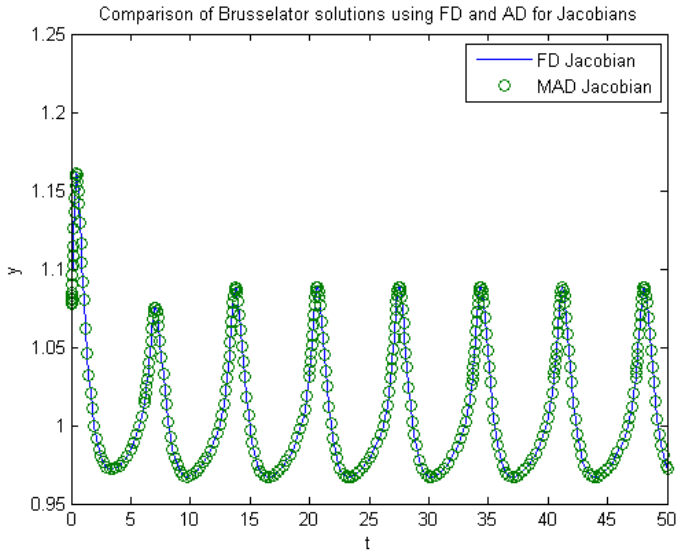
```
339 successful steps
54 failed attempts
776 function evaluations
```

```
13 partial derivatives
110 LU decompositions
774 solutions of linear systems
MAD Jacobian (sparsity given) : cputime = 1.1216
MADODEreport
Function name brussode_f
Jacobian of size 160x160
Fwd AD - full       Inf s
Fwd AD - sparse     0.030043 s
Fwd AD - compressed 0.010014 s
Percentage Sparsity    = 2.4844%
Maximum non-zeros/row  = 4 of 160
Maximum non-zeros/col  = 4 of 160
Size of compressed seed = 160x4
Percentage compressed to full seed    = 2.5%
FWD AD - compressed selected
Large system reject full method. Sparse is worse than compressed - discarding Sparse.
```

Difference in solutions

Of course, there should be no discernable difference between the finite difference and AD-enabled solutions.

```
plot(tfd,yfd(:,1),'-',tmadjac,ymadjac(:,1),'o')
xlabel('t')
ylabel('y')
legend('FD Jacobian','MAD Jacobian')
title('Comparison of Brusselator solutions using FD and AD for Jacobians')
```



Specifying a Jacobian technique

We can specify from the outset that compression is to be used.

```
MADODEDelete % delete any internal settings
MADODERegister(@brussode_f,'sparsity_pattern',S,'ad_technique','ad_fwd_compressed')
options=odeset(options,'Jacobian',@MADODEJacEval);
t1=cputime;
[tmadjac,ymadjac]=ode15s(@brussode_f,tspan,y0,options,N);
cpumadjac_comp=cputime-t1;
disp(['MAD Jacobian (compressed specified) : cputime = ',num2str(cpumadjac_S)])
MADODEReport
```

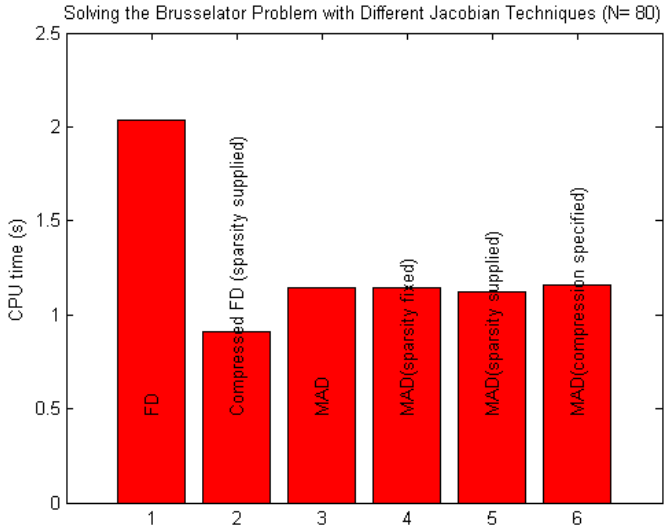
```
MADJacInternalRegister: AD technique selected is ad_fwd_compressed
339 successful steps
54 failed attempts
776 function evaluations
13 partial derivatives
110 LU decompositions
774 solutions of linear systems
MAD Jacobian (compressed specified) : cputime = 1.1216
MADODEreport
Function name brussode_f
Jacobian of size 160x160
Fwd AD - compressed 0.010014 s
Percentage Sparsity    = 2.4844%
Maximum non-zeros/row  = 4 of 160
Maximum non-zeros/col  = 4 of 160
Size of compressed seed = 160x4
Percentage compressed to full seed    = 2.5%
FWD AD - compressed selected
AD_FWD_COMPRESSED specified by call to MADJacInternalRegister
```

Conclusions

For this problem compressed finite-differencing is most efficient. If the sparsity pattern is not known then sparse forward mode AD outperforms finite-differencing. On supplying the sparsity pattern the performance of compressed forward mode AD approaches that of compressed finite-differencing. Simply using the MADODEJacEval interface yields acceptable performance.

```
bar([cpufd,cpucompfd,cpumadjac,cpumadjac_fixed,cpumadjac_S,cpumadjac_comp])
colormap hsv
ylabel('CPU time (s)')
title(['Solving the Brusselator Problem with Different Jacobian Techniques (N= ',num2str(N),')'])
text(1,cpucompfd/2,'FD','Rotation',90)
text(2,cpucompfd/2,'Compressed FD (sparsity supplied)','Rotation',90)
text(3,cpucompfd/2,'MAD','Rotation',90)
```

```
text(4,cpucompfd/2,'MAD(sparsity fixed)','Rotation',90)
text(5,cpucompfd/2,'MAD(sparsity supplied)','Rotation',90)
text(6,cpucompfd/2,'MAD(compression specified)','Rotation',90)
```



The example `MADEXpolluode` provides a third example for these techniques.

6.2 Unconstrained Optimization

High-level interfaces to MAD's functionality for MATLAB's Optimization Solvers have also been developed. In this section we cover unconstrained optimisation, in Section 6.3 we cover the constrained case.

Example 6.3 MADEXbanana: Unconstrained Optimization Using High-Level Interfaces

This example illustrates how the gradient for an unconstrained optimization problem, the banana problem of `bandem`*, may be calculated using the high-level interface function* `MADOptObjEval` *and used within* `fminunc`*.*

See also MADOptObjEval, MADOptObjDelete, MADOptObjRegister, MADOptObjReport, fminunc, banana_f, banana_fg, bandem

Contents

- *Default BFGS optimization without gradient*
- *Use of exact gradient*
- *Use of MAD's high-level interface MADOptObjEval*
- *Information on the AD techiques used*

Default BFGS optimization without gradient

If the gradient is not supplied then `fminunc` *will approximate it using finite-differences.*

```
format compact
x=[0 1];
options=optimset('LargeScale','off');
t1=cputime;
[xsol,fval,exitflag,output] = fminunc(@banana_f,x,options);
xsol
disp(['Default with no gradient  took ',num2str(output.iterations),...
    ' iterations and CPU time = ',num2str(cputime-t1)])
```

```
Optimization terminated: relative infinity-norm of gradient less than options.TolFun.
xsol =
    1.0000    1.0000
Default with no gradient  took 20 iterations and CPU time = 0.030043
```

Use of exact gradient

Since the function is just quadratic it may easily be differentiated by hand to give the analytic gradient in `banana_fg`*.*

```
options=optimset(options,'GradObj','on');
t1=cputime;
[xsol,fval,exitflag,output] = fminunc(@banana_fg,x,options);
xsol
disp(['Default with analytic gradient took ',num2str(output.iterations),...
    ' iterations and CPU time = ',num2str(cputime-t1)])
```

```
Optimization terminated: relative infinity-norm of gradient less than options.TolFun.
xsol =
    1.0000    1.0000
Default with analytic gradient took 20 iterations and CPU time = 0.020029
```

Use of MAD's high-level interface MADOptObjEval

Using `MADOptObjEval` *is easy, we first delete any previous settings, register the objective function* `banana_f` *and then use* `MADOptObjEval` *as the objective function which will automatically calculate the gradient when needed.*

```
MADOptObjDelete
MADOptObjRegister(@banana_f)
options=optimset(options,'GradObj','on');
t1=cputime;
```

```
[xsol,fval,exitflag,output] = fminunc(@MADOptObjEval,x,options);
xsol
disp(['Jacobian by MADFandGradObjOpt  took ',num2str(output.iterations),...
    ' iterations and CPU time = ',num2str(cputime-t1)])


Optimization terminated: relative infinity-norm of gradient less than options.TolFun.
xsol =
    1.0000    1.0000
Jacobian by MADFandGradObjOpt  took 20 iterations and CPU time = 0.09013
```

Information on the AD techiques used

The function ***MADOptObjReport*** *displays information on the techniques used. In this case since there are only 2 independent variables forward mode with full storage is used.*

```
MADOptObjReport


MADOptObjReportODE
Function name banana_f
Jacobian of size 1x2
Fwd AD - full       0.010014 s
Fwd AD - compressed Inf s
FWD AD - full selected
MADJacInternalEval: n< MADMinSparseN=10 so using forward mode full
```

6.3 Constrained Optimization

If we have a constrained optimization problem then we use `MADOptObjEval` for the objective, as in Example 6.3, and `MADOptConstrEval` for the constraint as Example 6.4 demonstrates.

Example 6.4 MADEXconfun: Constrained Optimization Using High-Level Interfaces.

This example shows how to use MAD's high-level interface functions to obtain both the objective function gradient and the constraint function Jacobian in a constrained optimisation problem solved using MATLAB's ***fmincon*** *function.*

See also MADEXbanana, MADOptConstrRegister, MADOptConstrEval, MADOptConstrReport, MADOptConstrDelete, MADOptObjRegister, MADOptObjEval, MADOptObjReport, MADOptObjDelete

Contents

Default optimization without gradients

Here we simply use the medium-scale algorithm with gradients evaluated by default finite-differencing.

```
options = optimset('LargeScale','off');
x0 = [-1 1];
t1=cputime;
[x,fval,exitflag,output] = fmincon(@objfun,x0,[],[],[],[],[],[],@confun,options);
t2=cputime-t1;
disp(['Default fmincon took ',num2str(output.iterations),...
    ' iterations and cpu= ',num2str(t2)])
```

```
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                                      1
                                      2
Default fmincon took 8 iterations and cpu= 0.040058
```

Using analytic gradients

The Mathworks supply functions `objfungrad` *and* `confungrad` *which return the objective's gradient and constraint's Jacobian when needed can be used for analytic evaluation of the required derivatives.*

```
options = optimset(options,'GradObj','on','GradConstr','on');
t1=cputime;
[x,fval,exitflag,output] = fmincon('objfungrad',x0,[],[],[],[],[],[],...
    'confungrad',options);
t2=cputime-t1;
disp(['Hand-coded derivs in fmincon took ',num2str(output.iterations),...
    ' iterations and cpu= ',num2str(t2)])
```

```
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
```

```
                                    1
                                    2
Hand-coded derivs in fmincon took 8 iterations and cpu= 0.040058
```

Using MAD's high-level interfaces

Calculating the gradients and Jacobians using MAD's high-level interfaces is easy, we first use `MADOptObjDelete` *and* `MADOptConstrDelete` *to delete any old settings, then register the objective and constraint functions using* `MADOptObjRegister` *and* `MADOptConstrRegister`*, then optimize using* `MADOptObjEval` *and* `MADOptConstrEval` *to supply the objective and constraints, and when needed their derivatives.*

```
MADOptObjDelete
MADOptConstrDelete
MADOptObjRegister(@objfun)
MADOptConstrRegister(@confun)
t1=cputime;
[x,fval,exitflag,output] = fmincon(@MADOptObjEval,x0,[],[],[],[],[],[],...
    @MADOptConstrEval,options);
t2=cputime-t1;
disp(['MAD driver in fmincon took ',num2str(output.iterations),...
    ' iterations and cpu= ',num2str(t2)])
```

```
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                                    1
                                    2
MAD driver in fmincon took 8 iterations and cpu= 0.10014
```

Seeing which AD techniques are used

The functions `MADOptObjReport` *and* `MADOptConstrReport` *report back which techniques have been used and why. Here full mode is used since the number of derivatives is small.*

```
MADOptObjReport
MADOptConstrReport
```

```
MADOptObjReportODE
Function name objfun
Jacobian of size 1x2
Fwd AD - compressed Inf s
FWD AD - full selected
MADJacInternalEval: n< MADMinSparseN=10 so using forward mode full
```

```
MADOptConstrReport
Function name confun
Jacobian of size 2x2
Fwd AD - full       0.010014 s
Fwd AD - compressed Inf s
FWD AD - full selected
MADJacInternalEval: n< MADMinSparseN=10 so using forward mode full
```

7 Black Box Interfaces

In some cases it is necessary or desirable for a user to provide code directly to provide the derivatives of a function that forms part of the target function that is being differentiated with MAD. Examples of this are when analytic derivatives for a function are available, or perhaps when the function is coded in another language (e.g., Fortran or C) and interfaced to MATLAB via a mex file or some other interface. We refer to such a function as a *black box function* since, as far as MAD is concerned it is a black box that should not be differentiated directly but for which derivative information will be supplied by the user. In order for MAD to differentiate through such black box functions three steps are necessary:

1. Use the function `MADBlackBoxRegister` to register the black box function with MAD supplying information on the number of input and output arguments, which of these are active (i.e., have derivatives), and to register a second function which calculates the necessary derivatives of output variables with respect to inputs.
2. Provide a function to calculate the required derivatives of the black box function.
3. Change the line of code in which the black box function is called so that the function call is made via the function `MADBlackBoxEval`.

The example of `MADEXAERBlackBox` will make this clearer.

Example 7.1 MADEXAERBlackBox: Black Box Interface Example for AER Problem

Demonstration of using black box interface to calculate the gradient of the AER (analysis of an enzyme reaction) problem from MINPACK-2 collection.

The AER problem is to find x to minimise the sum of squares,

```
f(x)=sum(F(x).^2)
```

where the residual vector F is supplied by MinpackAER_F. Here we look at how the gradient of f(x) may be calculated by direct use of `fmad` *or by making use of the Jacobian of F(x) via a Black Box Interface.*

See also: MADBlackBoxRegister, MinpackAER_Prob, TomlabAERBlackBox, ToolboxAERBlackBox

Contents

Function definition

The function we wish to differentiate is `AERObj_f` *which consists of the following code:*

```
function f=AERObj_f(x,Prob)

R=MinpackAER_F(x,Prob);

f=R'*R;
```

the Jacobian `DR/Dx` *of the function* `MinpackAER_F` *is available in the function* `MinpackAER_J`*. We may differentiate* `AERObj_f` *in one of two ways:*

- *Using* `fmad` *throughout.*
- *Using the Jacobian information from* `MinpackAER_J` *within a black box interface with* `fmad`*.*

Direct use of fmad

By using the `fmad` *class in the usual way we calculate the function's gradient.*

```
Prob=MinpackAER_Prob(zeros(4,1)); % set up structure Prob
x0=Prob.x_0;
x=fmad(x0,eye(length(x0)));
f=AERObj_f(x,Prob);
gradient=getinternalderivs(f)
```

```
gradient =
    0.1336   -0.0007   -0.0090    0.0111
```

Setting up the black box interface

We'll go through the 3 stages step-by-step.

1) Register the black box function

First we use `MADBlackBoxRegister` *to specify that* `MinpackAER_F` *is to be treated as a black box so that* `fmad` *needn't differentiate it and that the Jacobian of this function will be returned by* `MinpackAER_J`*. Note that we specify that* `MinpackAER_F` *has* `Nin`*=2 input variables,* `NOut=1` *output variable and that only the first input (*`ActiveIn=1`*) and first output (*`ActiveOut=1`*} are active. If more than one input or output is active then ActiveIn and ActiveOut are vector lists of the position of the active variables in the argument list. As a result of this* `MinpackAER_J` *should calculate the Jacobian of the first output with respect to the first input.*

```
MADBlackBoxRegister(@MinpackAER_F,'NIn',2,'NOut',1,'ActiveIn',1,'Activeout',1,...
    'JHandle',@MinpackAER_J)
```

2) Provide a function to calculate required derivatives

This is the aforementioned `MinpackAER_J`.

3) Prepare a modified objective function

We must now modify the code of the objective function in `AERObj_f` *so that the call to* `MinpackAER_F` *is wrapped within a call to* `MADBlackBoxEval` *- we have coding such a change in* `AERObjBlackBox_f` *replacing the line,*

```
R=MinpackAER_F(x,Prob);
```

of `AERObj_f` *with*

```
R=MADBlackBoxEval(@MinpackAER_F,x,Prob);
```

in `AERObjBlackBox_f`. `MADBlackBoxEval` *has a similar syntax to MATLAB's* `feval` *function, i.e. its arguments are the handle to the original function followed by the arguments passed to the original function. When evaluated with non-* `fmad` *arguments* `MADBlackBoxEval` *will simply use* `feval(@MinpackAER_F,x,Prob)` *so* `AERObjBlackBox_f` *may be used with arguements of class double.*

Calculate derivatives using black box interface

It is now easy to calculate the required derivatives since `MADBlackBoxEval` *takes care of all the interfacing of* `fmad` *variables with those supplied by* `MinpackAER_J`.

```
x=fmad(x0,eye(length(x0)));
f=AERObjBlackBox_f(x,Prob);
gradient=getinternalderivs(f)
```

```
gradient =
    0.1336   -0.0007   -0.0090    0.0111
```

Conclusions

Using MAD's black box interface facilities it is straightforward to make use of analytic expressions, hand-coding, or external Fortran or C programs for calculating derivatives of functions.

For an example using such an interface in conjunction with the TOMLAB optimisers see `MADEXTomlabAERBlackBox`; MATLAB Optimization Toolbox users should see `MADEXToolboxAERBlackBox`.

MAD stores all data about black box functions in the global variable `MADBlackBoxData`. In addition to `MADBlackBoxRegister` and `MADBlackBoxEval` further black box interface functions to manipulate the data in `MADBlackBoxData` are:

- `MADBlackBoxDisplay`: which displays a list of all registered black box functions. For example, after running Example 7.1 `MADBlackBoxDisplay` produces the following output.

```
>> MADBlackBoxDisplay
Function : MinpackAER_F
  Type      : simple
  File      : C:\ShaunsDocuments\matlab\MAD\EXAMPLES\BlackBox\MinpackAER_F.m
  NIn       : 2
  NOut      : 1
  ActiveIn  : 1
  ActiveOut : 1
  J function  : MinpackAER_J
    Type      : simple
    File      : C:\ShaunsDocuments\matlab\MAD\EXAMPLES\BlackBox\MinpackAER_J.m
```

This shows that `MinpackAER_F` has been registered as a black box interface function, gives the file that defines the function, information on the input and output variables, and information on the Jacobian function.

- `MADBlackBoxIndex`: Given the function handle of a registered black box function, this function returns the index of that function (and optionally the function name) in MAD's internal list of black box functions. For example,

```
>> i=MADBlackBoxIndex(@MinpackAER_F)
i =
     1

>> [i,name]=MADBlackBoxIndex(@MinpackAER_F)
i =
     1
name =
MinpackAER_F
```

  It is not anticipated that users will use this function.

- `MADBlackBoxDelete`: Given the function handle of a registered black box function, this function will delete that function from MAD's list of black box functions. For example,

```
>> MADBlackBoxDelete(@MinpackAER_F)
>> MADBlackBoxDisplay
MADBlackBoxDisplay: No functions registered
```

  Note that `MADBlackBoxDelete` with no function handle argument deletes details of **all** registered black box functions.

8 Conclusions & Future Work

In this document we have introduced the user to the automatic differentiation (forward mode) of MATLAB expressions and functions using the `fmad` class. We have included examples that illustrate:

- Basic usage for expressions
- Basic usage for functions
- Accessing the internal, 2-D matrix representation of derivatives.
- Preallocation of arrays to correctly propagate derivatives
- Calculating sparse Jacobians via dynamic sparsity or compression.
- Differentiating implicitly defined functions.
- Controlling dependencies
- The use of `fmad` in MATLAB toolboxes (ODEs and Optimization)
- Differentiating black box functions

Future releases of this document will feature reverse mode, second derivatives, FAQ's and anything else users find useful. The function coverage of `fmad` will be increased and present restrictions decreased with user demand. We are interested in receiving user's comments and feedback, particularly regarding cases for which `fmad` fails, gives incorrect results or is not competitive (in terms of run times) with finite-differences.

References

[aut05] autodiff.org. List of automatic differentiation tools. http://www.autodiff.org/?module=Tools, 2005.

[Azm97] Yousry Y. Azmy. Post-convergence automatic differentiation of iterative schemes. *Nuclear Science and Engineering*, 125:12–18, 1997.

[BB98a] M.C. Bartholomew-Biggs. Using forward accumulation for automatic differentiation of implictly-defined functions. *Computational Optimization and Applications*, 9:65–84, 1998.

[BB98b] Michael C. Bartholomew-Biggs. Using forward accumulation for automatic differentiation of implicitly-defined functions. *Computational Optimization and Applications*, 9:65–84, 1998.

[BCH+98] Christian H. Bischof, Alan Carle, Paul Hovland, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0 users' guide (revision D). Technical Report ANL/MCS-P263-0991, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439 USA, 1998. Available via `http://www.mcs.anl.gov/adifor/`.

[BCH+05] H. Martin Bücker, George F. Corliss, Paul D. Hovland, Uwe Naumann, and Boyana Norris, editors. *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*. Springer, New York, NY, 2005.

[BCKM96] Christian H. Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[BRM97] Christian H. Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[Chr94] Bruce Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3:311–326, 1994.

[Chr98] Bruce Christianson. Reverse accumulation and implicit functions. *Optimization Methods and Software*, 9(4):307–322, 1998.

[CV96] Thomas F. Coleman and Arun Verma. Structure and efficient Jacobian calculation. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, chapter 13, pages 149–159. SIAM, Philadelphia, PA, 1996.

[CV00] Thomas F. Coleman and Arun Verma. ADMIT-1: Automatic differentiation and MATLAB interface toolbox. *ACM Transactions on Mathematical Software*, 26(1):150–175, 2000.

[Dor96] John R. Dormand. *Numerical Methods for Differential Equations*. Library of Engineering Mathematics. CRC Press, 1996.

[DS96] J.E. Dennis, Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. SIAM Classice in Applied Mathematics. SIAM, Philadelphia, 1996.

[FK04] Shaun A. Forth and Robert Ketzscher. High-level interfaces for the MAD (Matlab Automatic Differentiation) package. In P. Neittaanmäki, T. Rossi, S. Korotov, E. Oñate, J. Périaux, and D. Knörzer, editors, *4th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS)*, volume 2. University of Jyväskylä, Department of Mathematical Information Technology, Finland, Jul 24–28 2004. ISBN 951-39-1869-6.

[For01] Shaun A. Forth. Notes on differentiating determinants. Technical Report AMOR 2001/1, Applied Mathematics & Operational Research, Cranfield University (RMCS Shrivenham), 2001.

[For06] Shaun A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software*, 32(2), June 2006.

[GJU96] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.

[GK96] Ralf Giering and Thomas Kaminski. Recipes for adjoint code construction. Technical Report 212, Max-Planck-Institut für Meteorologie, Hamburg, Germany, 1996.

[GMS91] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. Technical paper, The Mathworks, 1991. Available via `www.mathworks.com`.

[Gri00] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, PA, 2000.

[HS98] A.K.M. Shahadat Hossain and Trond Steihaug. Computing a sparse Jacobian matrix by rows and columns. *Optimization Methods and Software*, 10:33–48, 1998.

[INR05] INRIA Tropics Project. TAPENADE 2.0. http://www-sop.inria.fr/tropics, 2005.

[Kub94] K. Kubota. Matrix inversion algorithms by means of automatic differentiation. *Applied Mathematics Letters*, 7(4):19–22, 1994.

[mat06a] The MathWorks Inc., 3 Apple Hill Drive, Natick MA 01760-2098. *MATLAB Mathematics*, March 2006. http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/math.pdf.

[Mat06b] The Mathworks Inc., 3 Apple Hill Drive, Natick MA 01760-2098. *Optimization Toolbox User's Guide*, 2006. http://www.mathworks.com/access/helpdesk/help/pdf_doc/optim/optim_tb.pdf.

[NW99] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer series in operational research. Springer-Verlag, New York, 1999.

[PR98] J.D. Pryce and J.K. Reid. ADO1, a Fortran 90 code for automatic differentiation. Technical Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England, 1998. Available via `ftp://matisa.cc.rl.ac.uk/pub/reports/ prRAL98057.ps.gz`.

[RH92] Lawrence C. Rich and David R. Hill. Automatic differentiation in MATLAB. *App. Num. Math.*, 9:33–43, 1992.

[SR97] L.F. Shampine and M.W. Reichelt. The MATLAB ODE suite. *SIAM J. Sci. Comput.*, 18:1–22, 1997.

[Ver98a] Arun Verma. ADMAT: Automatic differentiation in MATLAB using object oriented methods. In *SIAM Interdisciplinary Workshop on Object Oriented Methods for Interoperability*, pages 174–183, Yorktown Heights, New York, Oct 21-23 1998. SIAM, National Science Foundation.

[Ver98b] Arun Verma. *Structured Automatic Differentiation*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1998.

[Ver98c] Arun Verma. *Structured Automatic Differentiation*. PhD thesis, Cornell University, 1998.

[Ver99] Arun Verma. ADMAT: Automatic differentiation in MATLAB using object oriented methods. In M. E. Henderson, C. R. Anderson, and S. L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop*, pages 174–183, Philadelphia, 1999. SIAM.