# ADMAT™: Automatic Differentiation Toolbox

**For use with MATLAB**

## User's Guide

*Version 2.0*

***Cayuga Research***

# Contents

# Chapter 1

# Introduction

Many scientific computing tasks require the repeated computation of derivatives. Hand-coding of derivative functions can be tedious, complex, and error-prone. Moreover, the computation of first and second derivatives, and sometimes the Newton step, is often a dominant expense in a scientific computing code. Derivative approximations such as finite-differences involve additional errors.

This toolbox is designed to help a MATLAB user compute first and second derivatives and related structures efficiently, accurately, and automatically. ADMAT employs many sophisticated techniques, exploiting sparsity and structure, to gain efficiency in the calculation of derivative structures (e.g., gradients, Jacobians, and Hessians). Moreover, ADMAT can directly and effectively calculate Newton steps for nonlinear systems.

To use ADMAT to evaluate a smooth nonlinear 'objective function' at a given argument, a MATLAB user need only supply an M-file. On request and when appropriate, ADMAT will ensure that in addition to the objective function evaluation, the Jacobian matrix, the Hessian matrix, and possibly the Newton step will also be evaluated at the supplied argument. The user need not supply derivative codes or approximation schemes.

*ADMAT 2.0 Features:*

- Efficient gradient computation (by 'reverse mode').
- Efficient evaluation of sparse Jacobian and Hessian matrices.
- A template design for the efficient calculation of 'structured' Jacobian and Hessian matrices.

- Efficient direct computation of Newton steps (In some cases avoiding the full computation of the Jacobian and/or Hessian matrix).
- Mechanisms and procedures for combining automatic differentiation of M-files with the finite differencing approximation for MEX files (for C and Fortran subfunctions).

### *.... and for the aficionado*

- " Forward" mode of automatic differentiation: A new MATLAB class "deriv" which overloads more than 100 MATLAB built-in functions.
- "Reverse" mode of automatic differentiation: A new MATLAB class "derivtape" which uses a virtual tape to record all functions and overloads more than 100 MATLAB built-in functions.
- MATLAB interpolation function INTERP1 is available.

### *Limitations of ADMAT*

ADMAT supports most frequently used computations for the first and second derivatives. However, the limitation of the current version of ADMAT is that it does not support the second derivation computation for a matrix. In other words, the matrix in a function for the second derivation computation has to be a constant, as opposed to a variable.

### *Guide Organization*

- Chapter 2: ADMAT installation.
- Chapter 3: Computing gradients, Jacobians and Hessians.
- Chapter 4: Sparse and structured computations.
- Chapter 5: Advanced use of ADMAT; Additional sparsity computations, etc.
- Chapter 6: Newton computations.
- Chapter 7: Using ADMAT with the MATLAB Optimization Toolbox.
- Chapter 8: Connecting with MEX files (Fortran, C) and finite-differencing.
- Chapter 9: Errors that may occur. Troubleshooting.
- Appendix A : ADMAT application examples.

- Bibliography.

## *Acknowledgement*

ADMAT 2.0 builds on the original work of Thomas Coleman and Arun Verma (ADMAT [12], ADMIT-2[14]). ADMAT 2.0 has increased functionality and is more efficient than those pioneering efforts. The technology behind ADMAT 2.0 is derived from research published by Coleman and colleagues over a number of years; many of the most relevant publications are listed in the Bibliography.

We thank Arun Verma for several illuminating discussions over the past years.

# Chapter 2

# Installation of ADMAT

In this chapter, the installation of ADMAT on a Unix (Linux) and Windows platform is discussed.

## 2.1 Requirements

ADMAT belongs to the "operator overloading" class of AD tools and uses object oriented programming features. Thus, ADMAT requires MATLAB 6.5 or above.

## 2.2 Obtaining ADMAT

The complete ADMAT package, *ADMAT 2.0 Professional*, can be licensed from Cayuga Research. See *www.cayugaresearch.com* for details. *ADMAT 2.0 Professional* can be evaluated free of charge for a period of up to 3 weeks.

A package of reduced functionality, *ADMAT 2.0 Student*, can be obtained from Cayuga Research and evaluated free of charge for a period of up to 3 weeks. Note that *ADMAT 2.0 Student* computes only the first derivatives by forward and reverse modes of automatic differentiation. Hence a user of *ADMAT 2.0 Student* can only use the functionalities described in §5.3, 5.4 and 5.5 and test the corresponding demos in the Demos\ Chapter 5 directory.

## 2.3 Installation Instructions for Windows Users

ADMAT is supplied in the zipped file ADMAT-2.0.zip. Please follow these installation instructions.

1. Place the ADMAT package in an appropriate directory and unzip the package using any unzip software.

2. There are two ways to set the search path in MATLAB.

   ********** Method 1. **********

   (a) Click "File" in the MATLAB window.

   (b) Choose the "Set Path" option.

   (c) Click the "Add with Subfolders" button.

   (d) Find the targetted directory for the ADMAT package in the"Browse for Folders" window and click "OK".

   (e) Click the "Save" button to save the path for ADMAT and click the "Close" button.

   (f) Type "startup" in the MATLAB prompt or log out of MATLAB and relog in.

   ********** Method 2. **********

   (a) Access the ADMAT directory.

   (b) Edit startup.m file.

   (c) Add ALL subdirectories of ADMAT search paths in the file manually;

   (d) Save the file and type startup in MATLAB prompt to set up the paths for the package.

3. There is a "success" message as ADMAT is correctly installed. Users can type " help ADMAT-2.0" in the MATLAB prompt to get a list of main functions in the package.

## 2.4 Installation Instructions for Unix (Linux) Users

1. Unzip ADMAT-2.0.zip using "unzip ADMAT-2.0.zip" in the Unix (Linux) prompt.

2. Follow Method 2 listed above.

# Chapter 3

# Computing Gradients, Jacobians and Hessians

## 3.1 Computing Gradients

One of the most common computations in scientific computing is the calculation of the gradient of a scalar-valued function. That is, if $f$ is a differentiable function that maps $n$-vectors $x$ to scalars, i.e., $f : R^n \rightarrow R^1$, the gradient of $f$ at a point $x$ is the vector of first derivatives:

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}.$$

ADMAT computes the gradient $\nabla f(x)$ automatically given both the MATLAB M-file to evaluate $f(x)$ and the value of the argument $x$ (an $n$-vector). ADMAT can compute the gradient using the overloaded MATLAB function 'feval'.

If "forward mode" is chosen, the gradient is computed in time comparable to that required by the forward finite-differencing. However, ADMAT does not require a differencing parameter and incurs no truncation error. The advantage of forward mode, compared to reverse mode, is that there are no 'extra' space requirements.

"Reverse mode" computes the gradient in time proportional to the time required to evaluate the function $f$ itself. This is optimal; reverse-mode can be approximately $n$-times faster than forward mode when applied to the computation of gradients. The downside of using reverse mode is that the space requirements can be quite large since the entire computational graph required to evaluate $f$ must be saved and then accessed in reverse order. See §5.2 for advice on how to use the inherent structure in

the program that evaluates $f$ to significantly reduce the space requirements.

The overloaded ADMAT function 'feval' provides a universal interface to compute gradients. It can be overloaded through the use of the function 'ADfun'.

We first illustrate with an example and then describe the general situation, followed by a second example.

## Example 3.1.1. Compute the gradient of the Brown function.

*See DemoGrad.m*

This example shows how to compute the gradient of the Brown function in ADMAT in three different ways: default (reverse), forward, and reverse. The definition of the Brown function is as follows:

$$y = \sum_{i=1}^{n-1} \{(x_i^2)^{x_{i+1}^2+1} + (x_{i+1}^2)^{x_i^2+1}\}.$$

A MATLAB function to evaluate the brown function is given below:

```
function value = brown(x,Extra)
% Evaluate the problem size.
n = length(x);
% Initialize intermediate variable y.
y=zeros(n,1);
i=1:(n-1);
y(i)=(x(i) .^ 2) .^ (x(i+1) .^ 2+1);
y(i)=y(i)+(x(i+1) .^ 2) .^ (x(i) .^ 2+1);
value=sum(y);
```

The following is an illustration of the use of ADMAT with the Brown function:

1. Set the Brown function as the function to be differentiated.
   >> myfun = ADfun('brown', 1);

   Note: the second input argument in ADfun, '1', is a flag indicating a scalar mapping, $f : R^n \to R^1$; more generally, the second argument is set to '$m$' for a vector-valued function, $F : R^n \to R^m$.

2. Set the dimension of x.

```
>> n = 5;
```

3. Initialize vector $x$.

```
>> x = ones(n,1)
x =
   1
   1
   1
   1
   1
```

4. Call feval to get the function value and the gradient of Brown function, allowing ADMAT to choose the mode (by default ADMAT chooses to use reverse mode for computing the gradients).

```
>> [f, grad] = feval(myfun, x)
f =
    8
grad =
    4    8    8    8    4
```

5. Use the forward mode to compute the gradient of $f$. In this case the third input argument of feval is set to the empty array, '[ ]' since no parameters are stored in the input variable Extra.

```
% Compute the gradient by forward mode
%
% Set options to forward mode AD; set input n as the problem size.
>> options = setgradopt('forwprod', n);
% compute gradient
>> [f,grad] = feval(myfun, x, [ ], options)
f =
    8
grad =
    4    8    8    8    4
```

6. Use the reverse mode to compute the gradient $g$. As the above case, the input Extra is '[]'.

```
% Compute the gradient by reverse mode
%
% set options to reverse mode AD. Input n is the problem size.
>> options = setgradopt('revprod', n);
% compute gradient
>> [f, grad] = feval(myfun, x, [], options)
f =
     8
grad =
     4     8     8     8     4
```

Functions ADfun and feval can be summarized as follows.

**Description of** ADfun **and** feval

fun= ADfun(infun,scalar)

**Input arguments**

'infun' is the function to be differentiated; a string representing the function name.
'scalar' is the dimension of the objective function value; a scalar

**Output arguments**

'fun ' is the function to overload feval; a MATLAB cell structure.

[f, g] = feval(fun, x, Extra, options).

**Input arguments**

'fun' is the function to be differentiated; an ADMAT ADfun class object.
'x' is a vector of the independent variables; either a row or column vector.
vector x is the 'point' at which the function and its gradient will be evaluated.
'Extra' stores parameters required in function fun; a MATLAB cell function.
'options' allows the user to choose the forward mode or reverse mode of automatic differentiation. It has to be defined through ADMAT function setgradopt.
The default mode for computing gradients for feval is the reverse mode.

**Output arguments**

'f ' is the function value at point x; a scalar.
'g ' is the gradient of $f$; a row vector.

options = setgradopt(mode, siz)

**Input arguments**

'mode' sets up the 'mode' of automatic differentiation, a string.

- mode = 'forwprod': compute gradients by the forward mode.
- mode = 'revprod': compute gradients by the reverse mode.

'siz ' is the size of the problem, a scalar.

There is a requirement on defining the function to be differentiated when invoking ADMAT in this manner: the function interface must contain just one output and two input arguments. For example,

y = functionName(x, Extra),

where y is the output, x is the independent input variable and Extra is a MATLAB cell structure, which stores all other parameters required in functionName. When there are no parameters stored in Extra, the empty array, '[ ]' is passed. The definitions of the Broyden and Brown functions in this chapter satisfy the requirement.

What can be done when a function has more than two input arguments? For example, suppose y = f(x, mu, gamma) where $x$ is the independent variable. In this case the second and third input arguments can be encapsulated as Extra.mu, and Extra.gamma. Thus, the new function call is y = f(x, Extra). Users can store scalars, vectors and even matrices as fields in Extra, but currently, matrices stored in MATLAB sparse format are not supported. We illustrate how to rewrap the original function to satisfy the interface requirement in **Example 3.1.2**.

## Example 3.1.2. Compute the gradient of a weighted mean function.

*See DemoFeval.m*

This example illustrates the use of input parameter 'Extra'. The weighted mean of a vector is computed.

```
function val = mean_weighted(x, mu, n)
%
%      Compute the mean of the n vector x
%      with the weight mu.
%
%      INPUT
%          x – a vector x
%         mu – weight for mean computation
%          n – length of x
%
%      OUTPUT
%         val – weighted mean of x
%
y = mu .* x;
val = sum(y)/n;
```

Obviously, the above function does not satisfy the 2-input requirement mentioned above for function feval. So we revise the function as indicated to satisfy the requirement.

```
function val = mean_feval(x, Extra)
%
%      Compute the mean of the n vector x
%      with the weight mu.
%
%       Note that: this function satisfies
%        the requirement for feval.
%
%      INPUT
%          x – vector x
%        Extra – stores other parameters, mu and n
%
%      OUTPUT
%         val – weighted mean of x
%
mu = Extra.mu;
n = Extra.n;
y = mu .* x;
val = sum(y)/n;
```

Now, the function mean_feval satisfies the 2-input argument requirement, so we can use feval to compute its gradient.

1. Set problem size.

   ```
   >> n = 5;
   ```

2. Initialize the random seed.

   ```
   >> rand('seed',0);
   ```

3. Define a vector x.

   ```
   >> x = rand(n,1)
   x =
      0.2190
      0.0470
      0.6789
      0.6793
      0.9347
   ```

4. Define a weight vector mu.

   ```
   >> mu = rand(n,1)
   mu =
      0.3835
      0.5194
      0.8310
      0.0346
      0.0535
   >> mu = mu/sum(mu);
   mu =
      0.2105
      0.2851
      0.4561
      0.0190
      0.0293
   ```

5. Assign variables mu and n to Extra.

   ```
   >> Extra.mu = mu;
   >> Extra.n = n;
   ```

6. Set mean_feval as the function to be differentiated.

   ```
   >> myfun = ADfun('mean_feval', 1);
   ```

7. Compute the gradient of mean_feval

   ```
   >> [f, grad ] = feval(myfun, x, Extra);
   f =
     0.0819
   grad =
       0.0421    0.0570    0.0912    0.0038    0.0059
   ```

This 2-input argument requirement can limit the flexibility of ADMAT. For example, if users would like to compute the derivatives of function y = f(x, mu, gamma) with regard to x, mu and gamma, respectively, the function must be rewrapped three times to satisfy the requirement before calling feval. In this situation, users can use the advanced features of ADMAT in §5.3 and §5.5 to compute the derivatives by the forward or reverse mode without the need to rewrap the function interface.

The gradient is a special case of the first derivative of a function $F : R^n \to R^m$ (i.e., $m = 1$). In the next section we discuss the general situation when $m > 1$.

## 3.2 Computing Jacobians

The Jacobian is the first order derivative of a vector-valued function $F$. That is, if $F$ is a differentiable vector-valued function, $F : R^n \to R^m$, then its corresponding Jacobian, evaluated at a point $x$, is an $m$-by-$n$ matrix:

$$J(x) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & & & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix},$$

where $y = F(x)$, $y = (y_1,\ y_2,\ \cdots, y_m)^T$ and $x = (x_1,\ x_2,\ \cdots, x_n)^T$.

Given a user-supplied MATLAB function (M-file) to evaluate the function $F$, and the current vector $x$, ADMAT can automatically (and accurately) determine the Jacobian matrix $J(x)$ at the point $x$. In fact, ADMAT is more general than this. Given an arbitrary matrix $V$ with $n$ rows, ADMAT can directly and accurately compute the product $J(x) \cdot V$ using the 'forward mode' version of automatic differentiation. $F$ is

differentiated along the columns of $V$ to obtain the result $J(x) \cdot V$. So if a user chooses $V$ to be the $n$-by-$n$ identity matrix $I$, then the forward-mode result is the Jacobian matrix $J(x)$. Alternatively, given an arbitrary matrix $W$ with $m$ columns, the 'reverse-mode' of ADMAT will produce, directly and accurately, the product $J^T(x) \cdot W$. So if the user chooses $W$ to be the $m$-by-$m$ identity matrix, then the reverse-mode of ADMAT produces the transpose of the Jacobian matrix, $J^T(x)$.

When the 'forward mode' in ADMAT is used, given the MATLAB function to evaluate $F$, the 'current point' $x$, and a matrix $V$ (with $n$ rows), then the product $J(x) \cdot V$ is determined automatically in time proportional to $c_V \cdot \omega(F)$, where $c_V$ is the number of columns of $V$ and $\omega(F)$ is the time taken for a single evaluation of $F$. There are no significant additional space requirements when the 'forward mode' version of ADMAT is used. Alternatively, when the 'reverse mode' in ADMAT is used, given the MATLAB function to evaluate $F$, the 'current point' $x$, and a matrix $W$ (with $m$ columns), then the product $J^T(x) \cdot W$ is determined automatically in time proportional to $c_W \cdot \omega(F)$, where $c_W$ is the number of rows of $W$ and $\omega(F)$ is the time taken for a single evaluation of $F$. The 'reverse mode' accesses the computational graph representing the evaluation of $F$ in reverse order; therefore, 'reverse mode' requires that the computational graph be saved - this can result in serious additional space demands. Thus, from a strict time complexity point of view, when $m << n$, the reverse mode is preferable to the forward mode; however, this advantage can sometimes be mitigated when the space requirements become excessive. See Chapters 4 and 5 to see how sparsity and structure can be used to reduce the space requirements for the 'reverse mode' option.

The following example illustrates how to compute Jacobians and their products.

### Example 3.2.1. Compute the Jacobian of the Broyden function by feval.

*See DemoJac.m*

The Broyden function is derived from a chemical engineering application [7]. Its definition is as follows.

```
function fvec = broyden(x, Extra)
% Evaluate the length of input
n = length(x);
% Initialize fvec. It has to be allocated in memory first in
ADMAT .
fvec=zeros(n,1);
i=2:(n-1);
fvec(i)= (3-2.*x(i)).*x(i)-x(i-1)-2*x(i+1) + 1;
fvec(n)= (3-2.*x(n)).*x(n)-x(n-1)+1;
fvec(1)= (3-2.*x(1)).*x(1)-2*x(2)+1;
```

1. Set the problem size.

   ```
   >> n = 5
   ```

2. Set 'broyden' as the function to be differentiated.

   ```
   >> myfun = ADfun('broyden', n);
   ```

   Note that the Broyden function is a vector-valued function which maps $R^n$ to $R^n$. Thus, the second input argument in ADfun is set to $n$ corresponding to the column dimension.

3. Set the independent variable x as an 'all ones' vector.

   ```
   >> x = ones(n,1)
   x =
      1
      1
      1
      1
      1
   ```

4. Call feval to compute the function value and the Jacobian matrix at $x$. We omit the input argument (Extra) when calling feval, given that it is empty.

   ```
   >> [F, J] = feval(myfun, x)
   F =
   ```

```
     0
    -1
    -1
    -1
     1
J =
    -1    -2     0     0     0
    -1    -1    -2     0     0
     0    -1    -1    -2     0
     0     0    -1    -1    -2
     0     0     0    -1    -1
```

5. Using the forward mode AD, compute the product $J(x) \times V$, where $V$ is an $n \times 3$ 'all ones' matrix. Set the third input argument to '[ ]', since no parameters are stored in Extra.

```
>> V = ones(n,3);
>> options = setopt('forwprod', V);     % set options to forward mode AD
>> [F,JV] = feval(myfun, x, [ ], options)
F =
     0
    -1
    -1
    -1
     1
JV =
    -3    -3    -3
    -4    -4    -4
    -4    -4    -4
    -4    -4    -4
    -2    -2    -2
```

6. Using the reverse mode AD compute the product of $J^T(x) \times W$, where $W$ is an $n \times 3$ 'all ones' matrix. We pass '[ ]' to the third input argument of feval since no parameters are stored in Extra.

```
>> W = ones(n,3);
>> options = setopt('revprod', W);     % set options to reverse mode AD
>> [F, JTW] = feval(myfun, x, [], options)
F =
```

$$
\begin{array}{r}
0 \\
-1 \\
-1 \\
-1 \\
1
\end{array}
$$

JTW =

$$
\begin{array}{rrr}
-2 & -2 & -2 \\
-4 & -4 & -4 \\
-4 & -4 & -4 \\
-4 & -4 & -4 \\
-3 & -3 & -3
\end{array}
$$

## 3.3 Computing Hessians

The Hessian matrix, $H$, is the symmetric matrix of second derivatives of a twice continuously-differentiable scalar-valued function $f : R^n \to R^1$:

$$
H = \begin{pmatrix}
\frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
\frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\
\vdots & & & \vdots \\
\frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2}
\end{pmatrix}.
$$

ADMAT can compute the Hessian matrix, using feval, given the M-file for defining the scalar-valued function $f$ and an argument $x$.

Function feval has two interfaces for Hessian computation. One interface returns the function value, gradient, and Hessian matrix; the other returns the product of the Hessian, $H$, and a matrix $V$.

- [f, grad, H] = feval(fun, x, Extra)
  **Input arguments**
  'fun' is the function to be differentiated; a string representing the function name.
  'x' is the vector of the independent variables; either a row or column vector.
  'Extra' stores parameters required in function fun; a MATLAB cell structure.

  **Output arguments**
  'f ' is the function value at point x; a scalar.
  'grad' is the gradient at point x; a row vector.
  'H' is the Hessian matrix evaluated at point x; a matrix.

- HV = feval(fun, x, Extra, options)
  **Input arguments**
  'fun' is the function to be differentiated; a string representing the function name.
  'x' is a vector of the independent variables (can be either a column or row vector).
  'Extra' stores parameters required in function fun.
  'options' is the flag for computing $H \times V$.
  - options = 'HtimesV': compute $H \times V$, where $H$ is a Hessian. Matrix $V$ is one of entries of options; a MATLAB cell structure.

  **Output Argument**
  'HV ' is the product $H \times V$, a matrix.

## Example 3.3.1. Compute the gradient and Hessian of Brown function using function "feval".

*See DemoHess.m*

1. Set the Brown function as the function to be differentiated.
   >> myfun = ADfun('brown', 1);

2. Set the problem size.

   >> n = 5;

3. Set the independent variable x to be an 'all ones' vector.

   ```
   >> x = ones(n,1)
   x =
      1
      1
      1
      1
      1
   ```

4. Call feval to get the function value, gradient and Hessian of the Brown function. We omit the third input argument, Extra, since it is empty.

```
>> [v, grad, H] = feval(myfun, x)
v =
     8
grad =
     4     8     8     8     4
H =
    12     8     0     0     0
     8    24     8     0     0
     0     8    24     8     0
     0     0     8    24     8
     0     0     0     8    12
```

5. Compute the product of $H$ and $V$, where $V$ is an $n \times 3$ 'all ones' matrix. We pass '[ ]' to the third input argument of feval since no parameters are stored in Extra.

```
>> V = ones(n,3);
>> options = setopt('htimesv', V);     % set options for H times V
>> HV = feval(myfun, x, [], options)
HV =
    20    20    20
    40    40    40
    40    40    40
    40    40    40
    20    20    20
```

In summary, function feval provides a common interface to compute gradients, as well as Jacobian and Hessian matrices. When there is sparsity or structure present the use of feval in the manner described above may not be efficient. In Chapter 4, we will discuss the efficient computation of sparse Hessian and Jacobian matrices, especially when evaluating Hessian or Jacobian matrices at different points with the same structure. Efficiency in the presence of more general structure, beyond sparsity, can be achieved using the techniques described in §5.2.

## 3.4 Data Type Consistency in Assignment Operation

Due to automatic data type conversion in Matlab 7.3 and above, we *highly recommend* users call function cons after initializing the dependent variable in their own function

definition. This will ensure that the data type of the dependent variable is consistent with that of the input independent variable. Otherwise, undetected errors in the derivative computation may occur.

## Descriptions of cons function

y = cons(y, x),

Make the data type of y consistent with that of x.

### Input arguments

"y" is the dependent variable, whose data type should be consistent with x in data type.
"x" is the independent variable.

### Output arguments

"y" is the dependent variable, consistent with x in data type.

The following example illustrates the benefit of using function cons.

## Example 3.4.1 Compare the results with and without calling cons.

*See DemoCons.m*

First, we define two similar functions. Sample2 calls the data type consistent function, cons while sample1 does not.

```
function y = sample1(x,m)

y=zeros(m,1);

for i = 1:m
    y(i) = x(i)*x(i);
end
```

```
function y = sample2(x,m)

y = zeros(m,1);
y = cons(y,x);
for i = 1:m
    y(i) = x(i)*x(i);
end
```

In the above function definitions, the only difference is that function sample2 calls

cons to make the data type of y consistent with that of x. Now we compare the results from these two functions.

1. Set problem size.
   >> n = 3;

2. Set independent variable.
   % Define a 'deriv' tape variable x, which is the ADMAT forward
   % mode type. Please refer to Section 5.3 for details
   >> x = deriv([1;2;3], eye(n));

3. Call sample1 function
   >> y1 = sample1(x,n)
   y1 =
   1
   4
   9

4. Call sample2 function
   >> y2 = sample2(x, n)
   val =
   1
   4
   9
   deriv =
   2 0 0
   0 4 0
   0 0 6

The result from function sample1 is the function value (only) even through the input x is an ADMAT forward mode type. This occurs because the MATLAB assignment operation in the for loop automatically converts the result of x(i)*x(i) into a double type due to the data type of y(i). Thus, ADMAT did nothing in this function call. However, with the use of function cons in sample2, the data type of y is consistent with x, which is the ADMAT forward mode type, deriv. Sample2 returns the function value and the derivative simultaneously as desired. Therefore, we *highly recommend* users to call cons after initializing the dependent variables in order to avoid some undetectable errors in the derivative computation by ADMAT.

# Chapter 4

# Sparse Jacobian and Hessian Matrices

Efficient computation of sparse Jacobian and Hessian matrices is one of the key features of ADMAT. The overall strategy is based on graph coloring techniques to exploit matrix sparsity [9, 10, 13].

Informally, a matrix is viewed as sparse if most of its entries are zero. There is no precise definition of a sparse matrix; however, the pragmatic view, which we adopt, is that a matrix is regarded as sparse if it is cost-effective to do so. That is, the use of sparse techniques can result in significant time savings.

## 4.1 Sparse Jacobian Matrix Computation

In this section, we illustrate how to evaluate a sparse Jacobian matrix. We give detailed descriptions of two popular sparsity functions "getjpi" and "evalj".

### Description of getjpi

Function "getjpi" computes the sparsity information for the efficient computation of a sparse Jacobian matrix. It is invoked as follows:

[JPI, SPJ] = getjpi(fun, m, n, Extra, method, SPJ)

#### Input arguments

"fun" is the function to be differentiated; a string variable representing the function

name.
"n" is the number of columns of the Jacobian matrix; a scalar.
"m" is the number of rows of the Jacobian matrix; by default, m = n; a scalar.
"Extra" stores parameters required in function fun apart from the independent variable; a MATLAB cell structure.
"method" sets techniques used to get the coloring Information:

- method = 'd': direct bi-coloring (the default),
- method = 's': substitution bi-coloring,
- method = 'c': one-sided column method,
- method = 'r': one-sided row method,
- method = 'f': sparse finite-difference.

Detailed background on the various methods is given in [12, 14].

"SPJ" is the user specified sparsity pattern of the Jacobian, in MATLAB sparse format.

**Output arguments**

"JPI" includes the sparsity pattern, coloring information and other information required for efficient Jacobian computation.
"SPJ" is the sparsity pattern of the Jacobian, represented in the MATLAB sparse format.

Note that ADMAT computes the sparsity pattern, 'SPJ', of the Jacobian first before calculating 'JPI'. This is an expensive operation. If the user already knows the sparsity pattern 'SPJ', then it can be passed to the function 'getjpi' as an input argument, so that ADMAT will calculate 'JPI' based on the user-specified sparsity pattern. There is no need to recalculate the sparsity pattern, 'SPJ'; it is inefficient to do so.

## Description of evalj

Function evalj computes the sparse Jacobian matrix based on the sparsity pattern and coloring information obtained from getjpi. It is invoked as follows:

[F, J] = evalj(fun, x, Extra, m, JPI, verb, stepsize)

**Input arguments**

"fun" is the function to be differentiated; a string representing the function name.
"x" is the vector of independent variables; a vector.
"Extra" contains parameters required in function fun; a MATLAB cell structure.
"m" is the row dimension of the vector mapping, that is $F : R^n \to R^m$; a scalar.
"JPI" is the sparsity pattern and coloring information recorded for the current sparse Jacobian Jacobian.
"verb" holds flags for display level. (See §4.3 for details)

- verb = 0: no display (the default).
- verb = 1: display the number of groups used.
- verb $\geq$ 2: display in graph upon termination.

"stepsize" is the step size for finite-differencing and is only needed when JPI is computed by the finite-difference method, 'f'. By default, stepsize = 1e-5.

**Output arguments**

"F " is the function value at point x; a vector.
"J" is the Jacobian matrix at point x; a sparse matrix.

## Example 4.1.1. Compute the Jacobian matrix of an arrowhead function.

*See DemoSprJac.m*

Let $y = F(x),\ F : R^n \to R^n$, where

$$\begin{aligned} y(1) &= 2x(1)^2 + \textstyle\sum_{i=1}^{n} x(i)^2 \\ y(i) &= x(i)^2 + x(1)^2, \quad i = 2 : n \end{aligned}$$

The following is the arrowhead function in MATLAB.

```
function y= arrowfun(x,Extra)
y = x.*x;
y(1) = y(1)+x'*x;
y = y + x(1)*x(1);
```

The corresponding Jacobian matrix has an arrowhead sparsity structure, as shown in Figure 4.1 for $n = 50$ with 148 nonzeros. The procedure to evaluate the Jacobian $J$ at an "all ones" vector for $n = 5$, is as follows.

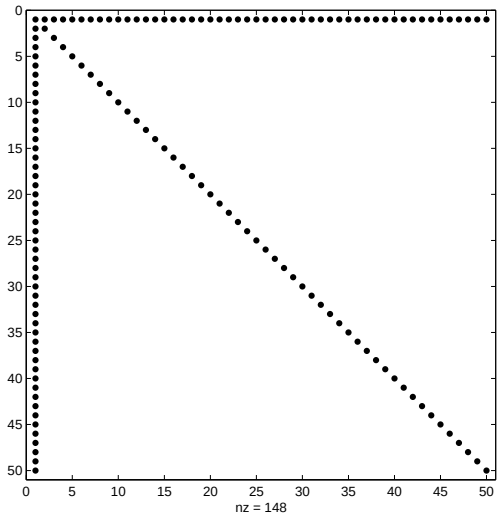


Figure 4.1: Jacobian Sparsity pattern of arrowfun.

1. Set problem size.
   >> n = 5;

2. Initialize x.
   >> x = ones(n,1)
   x =
   1
   1
   1
   1
   1

3. Compute "JPI" corresponding to function arrowfun.

   >> m = length(arrowfun(x));
   >> JPI = getjpi('arrowfun', m);

4. Compute the function value and the Jacobian (in MATLAB sparse format) based on the computed "JPI". Set the input argument, Extra, to '[ ]' since it is empty.

   >> [F, J] = evalj('arrowfun', x, [], n, JPI)

   F =

```
   7
   2
   2
   2
   2
J =
   (1,1)  6
   (2,1)  2
   (3,1)  2
   (4,1)  2
   (5,1)  2
   (1,2)  2
   (2,2)  2
   (1,3)  2
   (3,3)  2
   (1,4)  2
   (4,4)  2
   (1,5)  2
   (5,5)  2
```

The function "getjpi" determines the sparsity pattern of the Jacobian of arrowfun. It only needs to be executed once for a given function. In other words, once the sparsity pattern is determined by "getjpi", ADMAT calculates the Jacobian at a given point based on the pattern.

## 4.2 Sparse Hessian Computation

The process for determining sparse Hessian matrices is similar to the process for sparse Jacobians.

### Description of gethpi

Function gethpi computes the sparsity pattern of a Hessian matrix, and corresponding coloring information. It is invoked as follows:

[HPI, SPH] = gethpi(fun, n, Extra, method, SPH)

#### Input arguments

"fun" is the function to be differentiated; a string representing the function name.

"n" is the order of the Hessian; a scalar.
"Extra" stores parameters required in function fun (apart from the independent variable); a MATLAB cell structure.
"method" sets techniques used to get the coloring information.

- method= 'i-a': the default, ignore the symmetry. Computing exactly using AD.
- method= 'd-a': direct method, using AD.
- method= 's-a': substitution method using AD.
- method= 'i-f': ignore the symmetry and use finite differences (FD)
- method= 'd-f': direct method with FD.
- method= 's-f': substitution method with FD.

Note that details for the different methods are given in [12, 14].

"SPH" is the user-specified sparsity pattern of Hessian in MATLAB sparse format.

**Output arguments**

"HPI" includes the sparsity pattern, coloring information and other information required for efficient Hessian computation.
"SPH" is sparsity pattern of Hessian, represented in the MATLAB sparse format.

Similar to 'getjpi', users can pass 'SPH' to function 'gethpi' when the sparsity pattern of Hessian is already known. The computation of 'HPI' will be based on the user-specified sparsity pattern.

## Description of evalh

[v, grad, H] = evalh(fun, x, Extra, HPI, verb)

**Input arguments**

"fun" is function to be differentiated; a string representing the function name.
"x" is the independent variable; a vector.
"Extra" stores parameters required in function fun.
"HPI" is the sparsity pattern and coloring information of Hessian.
"verb" is flag for display level. (See §4.3)

- verb = 0: no display (the default).
- verb = 1: display the number of groups used.
- verb $\geq$ 2: display in graph upon termination.

**Output arguments**

"v " is the function value at point x; a scalar.
"grad" is the gradient at point x; a vector.
"H" is Hessian at point x; a sparse matrix.

## Example 4.2.1. Compute the Hessian of the Brown function at point $x' = [1,\ 1,\ 1,\ 1,\ 1]$.

*See DemoSprHess.m*

1. Set problem size.

   ```
   >> n = 5
   ```

2. Set independent variable.

   ```
   >> x = ones(n,1)
   x =
      1
      1
      1
      1
      1
   ```

3. Compute relevant sparsity information encapsulated in "HPI".

   ```
   >> HPI = gethpi('brown', n);
   ```

4. Evaluate the function value, gradient and Hessian at x. We set the input argument Extra to'[ ]' since it is empty.

   ```
   >> [v, grad, H] = evalh('brown', x, [], HPI)
   v =
        8
   ```

grad =

| | | | | |
|---|---|---|---|---|
| 4 | 8 | 8 | 8 | 4 |

H =

| | |
|---|---|
| $(1,1)$ | 12 |
| $(2,1)$ | 8 |
| $(1,2)$ | 8 |
| $(2,2)$ | 24 |
| $(3,2)$ | 8 |
| $(2,3)$ | 8 |
| $(3,3)$ | 24 |
| $(4,3)$ | 8 |
| $(3,4)$ | 8 |
| $(4,4)$ | 24 |
| $(5,4)$ | 8 |
| $(4,5)$ | 8 |
| $(5,5)$ | 12 |

Similar to the sparse Jacobian situation, function gethpi encapsulates the sparsity structure and relevant coloring information for efficient calculation of the sparse Hessian $H$. Function gethpi only needs to be executed once for a given function. In other words, once the sparsity pattern is determined by "gethpi", ADMAT calculates the Hessian at a given point based on the pattern.

## 4.3 Reporting

For both evalj and evalh, two different levels of display output are possible. Users can choose the input argument verb to set up the display level. We illustrate the use of verb with evalj.

We will repeat Example 4.1.1 here, but with the different values of verb.

- verb = 0. There is no information displayed.
- verb = 1. Verbose mode. For example, below is the the output produced by the direct bi-coloring methods:

  Number of Row groups = 1
  Number of column groups = 2
  Total Number of groups = 3

- verb $\geq$ 2. Additional sparsity patterns are shown. For example, for the bi-coloring method, three subplots are shown in Figure 4.2. The upper left subplot

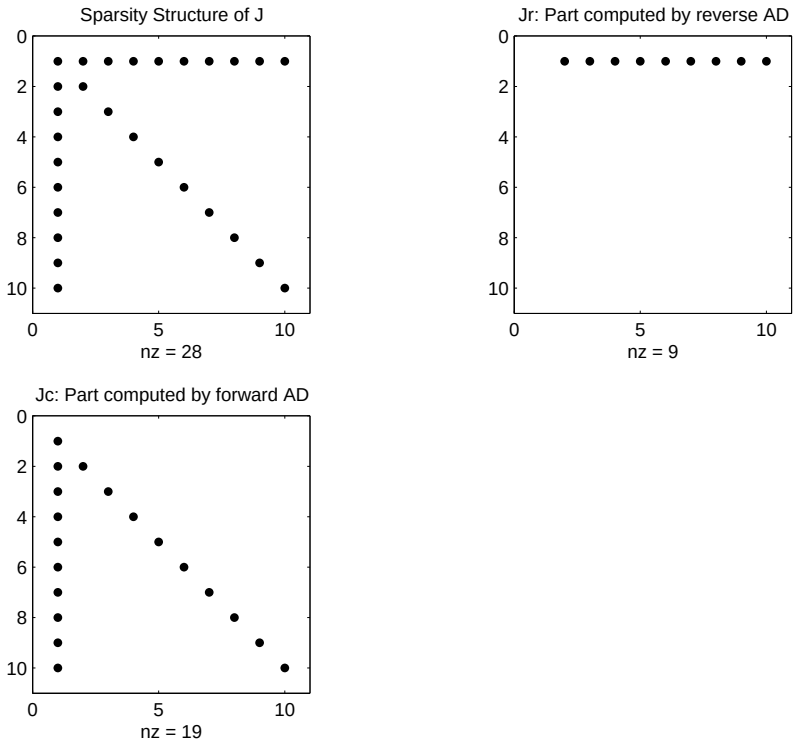


Figure 4.2: Jacobian Sparsity patterns of arrowfun computed by bi-coloring method with problem size 10.

shows the whole sparsity pattern of the Jacobian while the other two show the sparsity patterns computed by the forward mode AD and the reverse mode AD in the bi-coloring method, respectively.

# Chapter 5

# Advanced ADMAT

In the previous chapters, we introduced ADMAT fundamentals that allow for the computation of gradients, Jacobians, and Hessians. In this chapter we introduce a number of advanced features: Jacobian and Hessian sparsity pattern computation, efficient structured gradient computation, and derivative computation by forward and reverse mode with the direct use of the overloaded operations. These features can help users compute the sparsity patterns without forming the Jacobian or Hessian explicitly, reduce the space requirement for gradient computation by reverse mode, and avoid the function definition restrictions on "feval" calls.

## 5.1 Computing Sparsity Patterns of Jacobian and Hessian Matrices

ADMAT provides functions to compute the sparsity patterns of Jacobian and Hessian matrices.

### Sparsity Pattern of Jacobian

SPJ = jacsp(fun, m, n, Extra),

#### Input arguments

"fun" is the function to be differentiated; a string representing the function name.
"m" is the row dimension of the Jacobian; a scalar
"n" is the column dimension of the Jacobian; a scalar
"Extra" stores parameters required in fun; a MATLAB cell structure.

**Output argument**

"SPJ" is the sparsity pattern of the Jacobian; in MATLAB sparse format

## Example 5.1.1. Compute the sparsity pattern of the Broyden function.

*See DemoSP.m*

1. Set problem size.

   >> n = 5;

2. Compute the Jacobian sparsity pattern of the Broyden function.

   >> SPJ = jacsp('broyden', n,n)
   SPJ =
   $(1,1)$ 1
   $(2,1)$ 1
   $(1,2)$ 1
   $(2,2)$ 1
   $(3,2)$ 1
   $(2,3)$ 1
   $(3,3)$ 1
   $(4,3)$ 1
   $(3,4)$ 1
   $(4,4)$ 1
   $(5,4)$ 1
   $(4,5)$ 1
   $(5,5)$ 1

## Sparsity Pattern of Hessian

SPH = hesssp(fun, n, Extra)

**Input arguments**

"fun" is the function to be differentiated; a string representing the function name.
"n" is the order of the Hessian matrix; a scalar.
"Extra" stores parameters required in fun; a MATLAB cell structure.

**Output argument**

"SPH" is the sparsity pattern of the Hessian; in MATLAB sparse format.

### Example 5.1.2. Compute the sparsity pattern of the Hessian of the Brown function.

*See DemoSP.m*

1. Set problem size.

   ```
   >> n = 5;
   ```

2. Compute the sparsity pattern of the Hessian of the Brown function.

   ```
   >> SPH = hesssp('brown', n)
   SPH =
      (1, 1)   1
      (2, 1)   1
      (1, 2)   1
      (2, 2)   1
      (3, 2)   1
      (2, 3)   1
      (3, 3)   1
      (4, 3)   1
      (3, 4)   1
      (4, 4)   1
      (5, 4)   1
      (4, 5)   1
      (5, 5)   1
   ```

## 5.2 Efficient Computation of Structured Gradients

As mentioned in Chapter 3, the reverse mode is preferable to the forward mode when computing gradients in that the required computing time, in theory, is proportional to the time required to compute the objective function $f$ itself. This is optimal. However, reverse mode requires a large amount of memory since all intermediate results are required to be saved. Thus, on occasion the memory requirement of the reverse mode surpasses the internal (fast) memory available and the effective performance is significantly degraded. Fortunately, many practical computing problems exhibit 'structure',

and this structure can be exploited to reduce the practical computing time [8].

First we define a structured function. Given a scalar-valued function $z = f(x),\ f:\ R^n \to R$, a structured computation is defined as follows:

$$\begin{array}{l}\text{Solve for } y_1\text{: } F_1(x, y_1) = 0,\\ \text{Solve for } y_2\text{: } F_2(x, y_1, y_2) = 0,\\ \qquad\qquad \vdots\\ \text{Solve for } y_p\text{: } F_p(x, y_1, \cdots, y_p) = 0,\\ \text{Solve for } z\text{: } z - \bar{f}(x, y_1, \cdots, y_p) = 0,\end{array}$$

where $\bar{f}$ is a scalar-valued function. If we define the "extended" function $\tilde{F}_E^T = (F_1^T, F_2^T, \cdots, F_p^T)$, then the program to evaluate $f$ can be simply rewritten as,

$$\begin{array}{l}\text{Solve for } y\text{: } \tilde{F}_E(x, y) = 0\\ \text{"Solve" for output } z\text{: } z - \bar{f}(x, y_1, y_2, \cdots, y_p) = 0.\end{array}$$

Thus, the corresponding Jacobian of $F_E$ is in the form

$$J_E = \begin{pmatrix} (\tilde{F}_E)_x & (\tilde{F}_E)_y \\ \nabla_x \bar{f}^T & \nabla_y \bar{f}^T \end{pmatrix}.$$

The gradient of $f$, $\nabla_x f$, can be obtained from $J_E$ through a Schur-decomposition: eliminate the (2,2)-block, $\nabla_y \bar{f}^T$, using a block Gaussian transformation and then the transformed (2,1)-block will hold the desired result, i.e., $\nabla_x f^T$ [8].

The key point is that reverse mode automatic differentiation can be applied to the various structured steps indicated above, in turn. This can result in considerable space savings, and in practice, a shorter running time.

Next, we illustrate how to use this technique to reduce the memory requirements of reverse mode AD.

## Example 5.2.1. Computing the gradient while exploiting the structure

*See DemoStct.m*

Consider the autonomous ODE

$$y' = F(y),$$

where $F$ is the Broyden function. Suppose for an initial state $y_0 = x$, we employ an explicit Euler's method to compute the approximation $y_k$ to a desired final state $y(T)$. Then we estimate the error $z = f_0(y_k - y(T))$, where $f_0$ is the 2-norm.

The function $z = f(x)$ is defined as follows:

$$\begin{aligned} &y_0 = x, \\ &y_i = S(y_{i-1}) \qquad \text{for } i = 1, 2, \cdots, k, \\ &z = f_0(y_k), \end{aligned}$$

where $S: R^n \to R^n$ is a one step of Euler's method and $f_0 : R^n \to R$ is the 2-norm. The gradient of $f(x)$ is

$$[\nabla f(x)]^T = [\nabla f_0(y_k)]^T J_{k-1} J_{k-2} \cdots J_1 J_0,$$

where $J_i$ is the Jacobian of $S$ at $y_i$. There are at least three different methods to compute the gradient.

1. **Straightforward use of reverse mode**. The reverse mode is used to calculate the gradient of $f(x)$(refer to §5.5).

2. **Straightforward use of forward mode**. The forward mode is used to calculate the gradient (refer to §5.3).

3. **Sparse block computation with reverse mode**. Since $J_i$ is tridiagonal, the efficient sparse Jacobian computation introduced in Chapter 4 is used to compute each matrix $J_i$. Reverse mode is used to compute $\nabla f_0(y_k)$; finally, we calculate the product, $[\nabla f(x)V]^T$. The corresponding source code for the implementation is as follows.

```
%
%    compute the gradient of f(x) by sparse blocks
%
%
%   compute the sparsity of function S
JPI = getjpi('funcS', n, n, Extra);
%   evaluate J1, J2, ...., Jp, each with the same sparsity.
J = sparse(eye(n));
y = x;
for i = 1 : Extra.p
    [y, J1] = evalj('funcS', y, Extra, n, JPI);
    J = J1*J;
end
```

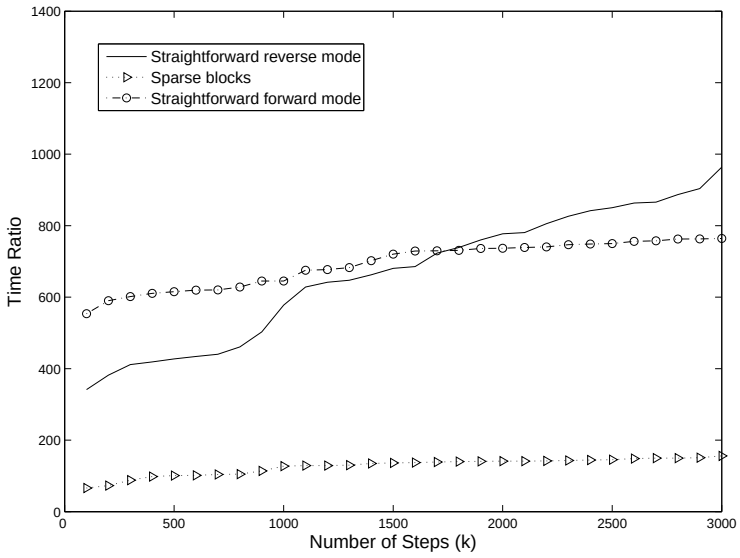


Figure 5.1: The time ratio of $\omega(\nabla f)/\omega(f)$ for Euler's method.

```
%  compute the gradient of function f0, where f0 is 2-norm
myfun = ADfun('funcF0',1);
options = setopt('revprod', 1);
[f, gSpr]= feval(myfun, y, Extra, options);
gSpr = gSpr*J;
```

In the above source code, ADMAT is used to calculate $J_0, J_1, \cdots, J_{k-1}$ through 'evalj'. Since the matrices $J_0, J_1, \cdots, J_{k-1}$ have the same structure, the sparsity 'JPI' only needs to be computed once. Then, the gradient, $\nabla f_0(y_k)$, is evaluated by the reverse mode AD through 'feval'.

Figure 5.1 plots the ratio of $\omega(\nabla f)/\omega(f)$, where $\omega(f)$ is the execution time of evaluating function $f$, on a fixed problem size $n = 800$ with different numbers of Euler's steps. The vertical axis is the time $\omega(\nabla f)$ taken to compute the gradient divided by the time $\omega(f)$ taken to evaluate the function. All calculations are done through ADMAT . The horizontal axis is the number of steps for Euler's method.

Figure 5.1 illustrates that the straightforward reverse mode performs very well for small problems, but the computing time spikes upward when the internal memory limitation is reached. The straightforward forward mode can outperform the reverse

mode when the memory requirements of reverse mode become excessive. The third approach exploits the sparsity of $J_i$. This approach is the fastest of the three procedures. For more details about the efficient computation of structured gradients, please refer to [8].

## 5.3 Forward Mode AD

Using forward mode AD, ADMAT can easily compute the first derivatives of functions which are defined using arithmetic operations and intrinsic functions of MATLAB. The forward mode AD provides users more flexibility with ADMAT than just using 'feval'. Users can define their own functions as usual, no restriction on the number of input arguments. When there is more than one input argument, the derivative with regard to any input argument can be computed by the forward mode without any change to the function definition. In this section, we will give several examples on how to use the forward mode AD in ADMAT .

### Descriptions of forward mode AD functions

- y = deriv(x, V),

  Define y as a deriv class object for the forward mode. Each object of the deriv class is a MATLAB struct array with two fields: val and deriv.

  **Input arguments**

  "x" is the value of the independent variable. y.val is initialized to be x.
  "V" is the value of the matrix $V$ which is used to compute $J \times V$ directly in forward mode. The dimension of V must be compatible with $J$ to compute the product. y.deriv is initialized to be $J * V$, which is essentially $V$ as the Jacobian of y respect to y is identity matrix.

  **Output arguments**

  "y" is an initialized object of the deriv class.

- val = getval(y),

  Get the value of the deriv class object y.

  **Input arguments**

"y" is the object of the deriv class.

**Output arguments**

"val" is the value of y, that is y.val.

- ydot = getydot(y),

  Get the product $J * V$ where $J$ is the Jacobian.

  **Input arguments**

  "y" is the object of class deriv.

  **Output arguments**

  "ydot" is the value of $J * V$, that is the field y.deriv.

## Example 5.3.1. Compute the first derivative of $f(x) = x^2$ at $x = 3$.

*See DemoFwd1.m*

This example shows how to compute the function value and the first order derivative of $y = x^2$ by using forward mode AD.

1. Define input argument $x$ to be a deriv object of which the value is 3 and the matrix $V$ is 1.

   ```
   >> x = deriv(3,1)     % create a deriv object of which the value is 3 and the matrix V is 1
   val =
        3
   deriv =
        1
   ```

2. Compute $y = x^2$.

   ```
   y = x ^ 2
   val =
   ```

```
    9
deriv =
    6
```

3. get the value of $y = x^2$

```
>> yval = getval(y)
yval =
    9
```

4. get the first order derivative of $y = x^2$.

```
>> ydot = getydot(y)
ydot =
    6
```

**Example 5.3.2. Compute the first order derivative of matrix-vector multiplication $A * x$.** (Of course the answer is matrix $A$ itself!)

*See DemoFwd1.m*

Differentiating a matrix-vector multiplication:

1. Size of matrix $A$.

```
>> n = 5;
```

2. $A$ is a $5 \times 5$ random matrix.

```
>> A = rand(n)
A =
   0.9501   0.7621   0.6154   0.4057   0.0579
   0.2311   0.4565   0.7919   0.9355   0.3529
   0.6068   0.0185   0.9218   0.9169   0.8132
   0.4860   0.8214   0.7382   0.4103   0.0099
   0.8913   0.4447   0.1763   0.8936   0.1389
```

3. $x$ is an all ones vector.

```
>> x = ones(n,1)
x =
   1
   1
   1
   1
   1
```

4. Initialization

```
>> x = deriv(x, eye(n))
val =
   1
   1
   1
   1
   1
deriv =
   1   0   0   0   0
   0   1   0   0   0
   0   0   1   0   0
   0   0   0   1   0
   0   0   0   0   1
```

5. Compute the multiplication of $y = Ax$.

```
>> y = A * x
val =
   2.7913
   2.7679
   3.2772
   2.4657
   2.5448
deriv =
   0.9501   0.7621   0.6154   0.4057   0.0579
   0.2311   0.4565   0.7919   0.9355   0.3529
   0.6068   0.0185   0.9218   0.9169   0.8132
   0.4860   0.8214   0.7382   0.4103   0.0099
   0.8913   0.4447   0.1763   0.8936   0.1389
```

6. Get the value of $y = Ax$.

```
>> yval = getval(y)
yval =
   2.7913
   2.7679
   3.2772
   2.4657
   2.5448
```

7. Get the first order derivative of $y$, i.e., the Jacobian matrix.

```
>> ydot = getydot(y)
ydot =
   0.9501   0.7621   0.6154   0.4057   0.0579
   0.2311   0.4565   0.7919   0.9355   0.3529
   0.6068   0.0185   0.9218   0.9169   0.8132
   0.4860   0.8214   0.7382   0.4103   0.0099
   0.8913   0.4447   0.1763   0.8936   0.1389
```

Note that the forward mode AD computes a product of the Jacobian matrix and $V$. Users can specify their own matrix $V$ when defining a deriv object. We set $V$ as an identity matrix in the above example so that the Jacobian matrix, $J$, is obtained. If $V$ is not an identity matrix, then the forward mode returns the product $J \times V$ rather than $J$ itself. If $V$ is not specified in defining a deriv object, say x = deriv(a), then $V$ is set to be zero by default. The storage of the deriv class will be discussed in §5.4.

## Example 5.3.3. Compute the Jacobian matrix of Broyden function at $x = [1, 1, 1]$.

*See DemoFwd2.m*

This example illustrates how to use the ADMAT forward mode on the Broyden function.

**Compute the user-specified function**

1. Create a deriv class object

```
>> x = deriv([1,1,1], eye(3))
val =
  1 1 1
```

deriv =

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

2. Call Broyden function at x

   >> y = broyden(x)

   val =

   $$\begin{matrix} 0 \\ -1 \\ 1 \end{matrix}$$

   deriv =

   $$\begin{matrix} -1 & -2 & 0 \\ -1 & -1 & -2 \\ 0 & -1 & -1 \end{matrix}.$$

3. Get the value of the Broyden function

   >> yval = getval(y)

   yval =

   $$\begin{matrix} 0 \\ -1 \\ 1 \end{matrix}$$

4. Get the Jacobian matrix of the Broyden function

   >> J = getydot(y)

   J =

   $$\begin{matrix} -1 & -2 & 0 \\ -1 & -1 & -2 \\ 0 & -1 & -1 \end{matrix}$$

## 5.4 Storage of deriv Class

After users specify matrix $V$, the number of columns of $V$ is assigned to a global variable globp, whose initial value is one.

- If x.val is a scalar, then its x.deriv field is a row vector, whose length equals globp;
- If x.val is a row/column vector, then its x.deriv field is a matrix, whose number of columns equals globp and the number of rows equals the length of x.val;
- If x.val is a matrix, then its x.deriv field is a 3-D matrix, e.g., $A(i, j, k)$. The value of $k$ varies from 1 to globp.

- ADMAT does not support multi-dimensional arrays above 2D, so it returns an error when the dimension of x.val is 3 or above.

### Example 5.4.1. Storage examples for the deriv class objects.

*See DemoStorage.m*

This example illustrates the storage of a deriv class object without a user-specified second input argument, V, when the input value is a scalar, vector or matrix with different values of globp.

- globp = 1

```
>> x = deriv(1)
   val =
        1
   deriv =
        0
>> x = deriv(ones(3,1))
   val =
        1
        1
        1
   deriv =
        0
        0
        0
>> x = deriv(ones(3))
   val =
        1   1   1
        1   1   1
        1   1   1
   deriv =
        0   0   0
        0   0   0
        0   0   0
```

- globp = 2

```
>> x = deriv(1)
   val =
        1
```

```
    deriv =
         0     0
>>  x = deriv(ones(3,1))
    val =
         1
         1
         1
    deriv =
         0     0
         0     0
         0     0
>>  x = deriv(ones(3))
    val =
         1     1     1
         1     1     1
         1     1     1
    deriv(:,:,1) =
         0     0     0
         0     0     0
         0     0     0
    deriv(:,:,2) =
         0     0     0
         0     0     0
         0     0     0
```

## 5.5 Reverse Mode AD

In the reverse mode, ADMAT uses a virtual tape to record all the intermediate values and operations performed in the function evaluation. Computation of starts from the end of the tape, a MATLAB global variable and goes backward through the tape. The requested derivative is finally recorded at the beginning of the tape. In this section, we will give examples of computing the first order derivative by the reverse mode AD.

### Descriptions of reverse mode AD functions

- y = derivtape(x, flag),

  Define a derivtape class for the reverse mode. Each derivtape object is a MATLAB struct array with two fields: val and varcount.

**Input arguments**

"x" is the value of the independent variable, value for the field val.
"flag" is a flag for the beginning of tape.

- "1" - create a derivtape object x , save x on the tape and set the number of the cell storing x as the beginning of the tape.
- "0" - create a derivtape object and save the value to the tape;
- None - create a derivtape object, but do not save the value on the tape.

**Output arguments**

"y" is an initialized derivtape class object.

- val = getval(y),

  Get the value of a derivtape class object.

  **Input arguments**

  "y" is a derivtape class object.

  **Output arguments**

  "val" is the value of y, that is y.val.

- JTW = parsetape(W),

  Compute the product of $J^T \times W$, where $J^T$ is the transpose of the first derivative of the differentiation function.

  **Input arguments**

  "W" is the user input value.

  **Output arguments**

  "JTW" is the product of $J^T \times W$.

### Example 5.5.1. Compute the first order derivative of MATLAB operation $y = x^2$.

*See DemoRvs1.m*

1. Define a derivtape object and record data on a tape.
   ```
   >> x = derivtape(3,1)      % record value 3 at the beginning of the tape
   val =                      <-- value recorded on the tape.
        3
   varcount =                 <-- place where the value is stored on the tape
        1
   ```

2. Compute $y = x^2$.
   ```
   >> y = x ^ 2
   val =
        9
   varcount =
        2
   ```

3. Get value of $y$.
   ```
   >> yval = getval(y)
   yval =
        9
   ```

4. Get the first order derivative of $y$.
   ```
   >> ydot = parsetape(1)
   ydot =
        6
   ```

### Example 5.5.2. Compute the first order derivative of matrix-vector multiplication $Ax$.

*See DemoRvs1.m*

1. Set the problem size.

   ```
   >> n = 5                   % size of the matrix
   ```

2. Get a random matrix.

   ```
   >> A = rand(n)
   ```

```
A =
   0.9501   0.7621   0.6154   0.4057   0.0579
   0.2311   0.4565   0.7919   0.9355   0.3529
   0.6068   0.0185   0.9218   0.9169   0.8132
   0.4860   0.8214   0.7382   0.4103   0.0099
   0.8913   0.4447   0.1763   0.8936   0.1389
```

3. Define $x$ as a derivtape object and set $x$ as the beginning of the tape.

```
>> x = derivtape(ones(n,1),1)
val =
   1
   1
   1
   1
   1
varcount =
     1
```

4. Compute $Ax$.

```
>> y = A*x
val =
   2.7913
   2.7679
   3.2772
   2.4657
   2.5448
varcount =
     2
```

5. Get the value of $y$

```
>> yval = getval(y)
yval =
   2.7913
   2.7679
   3.2772
   2.4657
   2.5448
```

6. Get the transpose of the first order derivative of $y$

```
>> JT = parsetape(eye(n))
JT =
   0.9501   0.2311   0.6068   0.4860   0.8913
   0.7621   0.4565   0.0185   0.8214   0.4447
   0.6154   0.7919   0.9218   0.7382   0.1763
   0.4057   0.9355   0.9169   0.4103   0.8936
   0.0579   0.3529   0.8132   0.0099   0.1389
```

Note that the reverse mode AD computes a product of the transpose of Jacobian matrix with $W$, that is $J^T W$. Users can specify their own $W$ when parsing the tape by "parsetape(W)". We set $W$ as an identity matrix in above example so that $J^T$ is obtained.

**Example 5.5.3. Compute the transpose of Jacobian matrix, $J^T$, of the Broyden function at $x = [1, 1, 1]$ in the reverse mode.**

*See DemoRvs2.m*

1. Define the Broyden function fvec = broyden(x, Extra) as in §3.1.

2. Create a derivtape class object.

```
>> x = derivtape([1,1,1],1)
val =
  1 1 1
varcount =
     1
```

3. Call function Broyden function with x

```
>> y = broyden(x)
val =
     0
    -1
     1
varcount =
    40
```

4. Get the value of $y$

```
>> yval = getval(y)
```

```
yval =
     0
    -1
     1
```

5. Get the transpose of Jacobian of $y$

```
>> JT = parsetape(eye(3))
JT =
    -1    -1     0
    -2    -1    -1
     0    -2    -1
```

Note that if a user-specified function is a mapping from $R^n$ to $R$, the Jacobian matrix reduces to the gradient. Theoretically, the reverse mode AD computes the gradient much faster than the forward mode AD does [9], but the reverse mode requires a large amount of memory space since it records each operation on a tape. This drawback sometimes leads to accessing low speed storage media which slows down the computation significantly.

## 5.6 Computing Second-Order Derivatives

Both the forward and reverse modes are used in the second order derivatives computation. For a scalar-valued function $f(x) : R^n \to R$, the forward mode is used to compute $w = (\nabla f)^T V$, then the reverse mode computes $(\frac{\partial w}{\partial x})^T$, that is $HV \cdot W$, where $H$ is the Hessian, since $W$ usually has fewer columns than the number of variables of $x$.

### Descriptions of functions for computing second order derivatives

- y = derivtapeH(x, flag, V),

  Define a derivtapeH class for computing the second order derivative.

  **Input arguments**

  "x" is the value of the independent variable.
  "flag" is a flag for the beginning of tape.

  - "1" - create a derivtapeH object x , save x on the tape and set the cell storing x as the beginning of the tape.

- "0" - create a derivtapeH object and save the value to the tape;
- None - create a derivtapeH object, but do not save the value on the tape.

"V" is the matrix for computing the product $g^T \times V$, where $g$ is the first derivative of the function to be differentiated.

**Output argument**

"y" is an initialized derivtapeH class object, which takes two cells on tape. One is for the independent variable x, the other is for V.

- val = getval(y),

  Get the value of the derivtapeH class object.

  **Input argument**

  "y" is a derivtapeH class object.

  **Output argument**

  "val" is the value of derivtapeH class object.

- ydot = getydot(y),

  Get the product $g^T \times V$ of a derivtapeH class object.

  **Input argument**

  "y" is a derivtapeH class object.

  **Output argument**

  "val" is the product $g^T \times V$ of the derivtapeH class object.

- HW = parsetape(W),

  Compute the product of $H \times W$, where $H$ is the second order derivative of the differentiation function.

  **Input argument**

"W" is the value of product $H \times W$.

**Output argument**

"HW is the product of $H \times W$.

## Example 5.6.1. Compute the first and second order derivatives of $y = x^2$.

*See Demo2nd1.m*

1. First define a derivtapeH object with value 3.

```
>> x = derivtapeH(3,1,1)
val =
     3
varcount =
     1
val =
     1
varcount =
     2
```

2. Compute $y = x^2$.

```
>> y = x ^ 2
val =
     9
varcount =
     3
val =
     6
varcount =
     6
```

3. Get the value of $y$.

```
>> yval = getval(y)
yval =
     9
```

4. Get the first order derivative of $y$.

```
>> y1d = getydot(y)
y1d =
     6
```

5. Get the second order derivative of $y$.

```
>> y2d = parsetape(1)
y2d =
     2
```

## Example 5.6.2. Compute the gradient and Hessian of Brown function.

*See Demo2nd2.m*

1. Set problem size.

```
>> n = 5          % problem size
```

2. 
```
>> x = ones(n,1)
x =
   1
   1
   1
   1
   1
```

3. Define a derivtapeH object.

```
>> x = derivtapeH(x,1,eye(n))
val =
   1
   1
   1
   1
   1
varcount =
     7
val =
```

```
    1   0   0   0   0
    0   1   0   0   0
    0   0   1   0   0
    0   0   0   1   0
    0   0   0   0   1
varcount =
     8
```

4. Call Brown function.

```
>> y = brown(x)
val =
     8
varcount =
     95
val =
     4   8   8   8   4
varcount =
     96
```

5. Get the value of $y$.

```
>> yval = getval(y)
yval =
     8
```

6. Get the gradient of the Brown function at $x$.

```
>> grad = getydot(y)
grad =
     4   8   8   8   4
```

7. Get the Hessian of the Brown function at $x$.

```
>> H = parsetape(eye(n))
H =
   12   8   0   0   0
    8  24   8   0   0
    0   8  24   8   0
    0   0   8  24   8
    0   0   0   8  12
```

## 5.7 1-D Interpolation in ADMAT

1-Dimension interpolation is available in the current version of ADMAT under 'interp1_AD', with the same input and output interfaces as the MATLAB 1-Dimension interpolation, 'interp1.m'. We did not overload the MATLAB interpolation function 'interp1'; instead, made a few changes to the Matlab function 'interp1' and its dependencies so that 'interp1_AD' is consistent for use with ADMAT. The following example illustrates the use of 'interp1_AD'.

### Example 5.7.1. 1-Dimension interpolation in ADMAT.

*See DemoInter1.m*

In this example, we will use two methods, linear interpolation method and cubic spline interpolation method, to estimate the function value and Jacobian at point x0 with the ADMAT 1-Dimension interpolation function, 'interp1_AD'.

1. Initial value for the interpolation.

   ```
   >> x = 0 : 10;
   >> y = sin(x);
   ```

2. Function points.

   ```
   >> x0 = 0.2 : 0.2 : 1;
   ```

3. Length of function points.

   ```
   >> n = length(x0);
   ```

4. Call 'interp1_AD' for interpolation.

   **Linear interpolation method**.

   (a) Linear interpolation at point x0 by forward mode AD.

   ```
   >> xi = deriv(x0, eye(n));
   >> yi = interp1_AD(x,y,xi);
   >> y0 = getval(yi)
   y0 =
   ```

```
    0.1683    0.3366    0.5049    0.6732    0.8415
>> J = getydot(yi)
J=
   0.8415         0         0         0         0
        0    0.8415         0         0         0
        0         0    0.8415         0         0
        0         0         0    0.8415         0
        0         0         0         0    0.0678
```

(b) Linear interpolation at point x0 by reverse mode AD.

```
>> xi = derivtape(x0,1);
>> yi = interp1_AD(x,y,xi);
>> y0 = getval(yi)
y0 =
    0.1683    0.3366    0.5049    0.6732    0.8415
>> J = parsetape(eye(n))
J=
   0.8415         0         0         0         0
        0    0.8415         0         0         0
        0         0    0.8415         0         0
        0         0         0    0.8415         0
        0         0         0         0    0.0678
```

**Cubic spline interpolation method.**

(a) Cubic spline interpolation at point x0 by forward mode AD.

```
>> xi = deriv(x0, eye(n));
>> yi = interp1_AD(x,y,xi, 'spline');
>> y0 = getval(yi)
y0 =
    0.2181    0.4134    0.5837    0.7270    0.8415
>> J = getydot(yi)
J=
   1.0351         0         0         0         0
        0    0.9155         0         0         0
        0         0    0.7859         0         0
        0         0         0    0.6462         0
        0         0         0         0    0.4965
```

(b) Cubic spline interpolation at point x0 by reverse mode AD.

```
>> xi = derivtape(x0,1);
>> yi = interp1_AD(x,y,xi,'spline');
>> y0 = getval(yi)
y0 =
    0.2181    0.4134    0.5837    0.7270    0.8415
>> J = parsetape(eye(n))
J=
   1.0351         0         0         0         0
        0    0.9155         0         0         0
        0         0    0.7859         0         0
        0         0         0    0.6462         0
        0         0         0         0    0.4965
```

Note that some other interpolation methods, such as the piecewise cubic Hermite interpolation method and the nearest neighbor interpolation method, are also supported in 'interp1_AD'. Users can refer to MATLAB 'interp1' help documentation for more details on the use of 'interp1_AD'.

# Chapter 6

# Newton Computations

ADMAT provides functions for optimization and the Newton step computation for nonlinear systems. There are two basic options for users. There is the 'traditional' Newton computation - the Jacobian (or Hessian) is first computed and then the Newton step is determined by solving the linear Newton system - and there is the use of an expanded Jacobian (Hessian) matrix formed through the use of structure [16]. The latter can yield significant cost benefits. Both approaches will be illustrated in this chapter.

## 6.1 Traditional Newton Computation

The Newton computation is widely used in solving nonlinear equations and optimization problems. For example, with respect to a nonlinear equation $F(x) = 0$, where $F : R^n \to R^n$, a Newton iteration defined at 'current' point $x$, is given by

$$\begin{aligned} &\text{Solve} \quad J(x)s_N = -F(x), \\ &\text{Update} \quad x = x + s_N, \end{aligned}$$

where $J(x)$ is the $n$-by-$n$ Jacobian of $F(x)$. ADMAT is a good tool to compute the Jacobian $J(x)$.

If the Jacobian matrix $J(x)$ is sparse, i.e., most of the entries of $J$ remain zero for all $x$, then ADMAT can be used both to determine this sparsity structure (once) and then efficiently determine the values of the non-zero components of $J(x)$ at each successive point $x$. Each Jacobian evaluation is followed by a sparse linear solver to determine the Newton step.

The following function, newton.m, is a pure Newton process. We do not recommend it

for general nonlinear systems and arbitrary starting points, since it may not converge, but we include it here because it illustrates the use of ADMAT in the context of a sparse nonlinear process. For general nonlinear systems, we recommend procedures that force convergence from arbitrary starting points (i.e., use line search or trust region strategy). MATLAB functions fsolve and fminunc are examples of such procedures.

Function newton requires the user to pass several arguments: the name of the function to be solved, starting point $x$, a solution tolerance, and a bound on the number of iterations. newton will then apply the pure Newton process and return the solution (with the given tolerance) or will return the iterate achieved upon exceeding the iteration bound.

```
function [x, normy, it] = newton(func, x0, tol, itNum, Extra)
%
%      A straight Newton process. Termination occurs when the norm of the vector
%     function func is less than the tolerance 'tol' or the iteration count reaches the
%     maximum number of iterations 'itNum'.
%
%      INPUT
%           func - nonlinear vector function
%           x0 – initial value of x
%           tol - stopping tolerance
%           itNum - iteration count limit
%           Extra – parameters for function 'func'
%
%     OUTPUT
%          x - approximate root of func as produced be the Newton process
%          normy – norm of function value at x
%          it – number of iterations
%
if (nargin < 2)
    error('At least two input arguments are required.');
end
switch (nargin)
    case 2
        tol = 1e-13;
        itNum = 100;
        Extra = [];
```

```
    case 3
        itNum = 100;
        Extra = [];
    case 4
        Extra = [];
        otherwise
end

x = x0; n = length(x0);
m = length(feval(str2func(func), x0));
% initialize the iteration counter
it = 0;

% determine sparse Jacobian structure
if nargin < 5
    JPI = getjpi(func, m);
else    % Extra is not empty
    JPI = getjpi(func, m, [], Extra);
end
normy = 1.0;

% Newton steps
while ( normy > tol) && (it < itNum)
    % evaluate the function value and the Jacobian matrix at x
    [y, J] = evalj(func, x, Extra, [],JPI);
    delta = -J\y;
    normy = norm(y);
    x = x + delta;
    it = it + 1;
end
```

**Example 6.1.1. Apply the Newton process to the nonlinear equation** arrowfun$(x) = 0$**, where** arrowfun **is defined in §4.1.**

*See DemoNewton.m*

1. Set problem size.

   >> n = 5;

2. Initialize the random seed.

```
>>  rand('seed',0);
```

3. Set starting point.

```
>> x = rand(n,1)
x =
   0.2190
   0.0470
   0.6789
   0.6793
   0.9347
```

4. Apply the Newton process to the system arrowfun($x$) = 0

```
>> [x, normy, it] = newton('arrowfun', x, 1e-8, 50)
x =
1.0e-004 *
   0.0668
   0.0144
   0.2072
   0.2073
   0.2852
normy =
    8.4471e-009
it =
     15
```

## 6.2 Structured Newton Computation

The main expense in the determination of a Newton step for a nonlinear system is often the evaluation of the Jacobian (Hessian) matrix. In particular, when the Jacobian is dense, evaluating the Jacobian matrix can be an expensive proposition (e.g., the cost of computing the Jacobian can be a factor of $n$ times the cost of evaluating the function $F$ itself, where $n$ is the number of columns in $J$.).

However, many (perhaps most) nonlinear systems with expensive dense Jacobians show a certain structure in their computations and if the code to evaluate $F$ is written to expose this structure (illustrated below) then it turns out that the Newton step can often be computed without fully computing the actual Jacobian matrix; this technique

can result in great cost savings [16].

Suppose that the computation $z = F(x)$ can be broken down into the following 'macro' steps, performed in top-down order:

$$\left.\begin{array}{rcl} \text{Solve for } y_1 & : & F_1^E(x, y_1) = 0 \\ \text{Solve for } y_2 & : & F_2^E(x, y_1, y_2) = 0 \\ \vdots & & \vdots \\ \text{Solve for } y_p & : & F_p^E(x, y_1, y_2, \cdots, y_p) = 0 \\ \text{"Solve" for output } z & : & z - F_{p+1}^E(x, y_1, y_2, \cdots, y_p) = 0 \end{array}\right\}. \tag{6.1}$$

For convenience define the 'extended' function,

$$F^E(x, y_1, \cdots, y_p) = \begin{pmatrix} F_1^E(x, y_1, \cdots, y_p) \\ F_2^E(x, y_1, \cdots, y_p) \\ \vdots \\ F_p^E(x, y_1, \cdots, y_p) \\ F_{p+1}^E(x, y_1, \cdots, y_p) \end{pmatrix}.$$

The Newton step for (6.1) can be written, as to solve

$$J^E \begin{pmatrix} \delta x \\ \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_p \end{pmatrix} = -F^E = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ -F \end{pmatrix}, \tag{6.2}$$

where the square (extended) Jacobian matrix $J^E$ is a block lower Hessenberg matrix:

$$J^E = \left(\begin{array}{c|cccc} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y_1} & & & \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y_1} & \frac{\partial F_2}{\partial y_2} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \frac{\partial F_p}{\partial x} & \frac{\partial F_p}{\partial y_1} & \cdots & \cdots & \frac{\partial F_p}{\partial y_p} \\ \hline \frac{\partial F_{p+1}}{\partial x} & \frac{\partial F_{p+1}}{\partial y_1} & \cdots & \cdots & \frac{\partial F_{p+1}}{\partial y_p} \end{array}\right) \tag{6.3}$$

Note that in (6.3) the pair $(F^E, J^E)$ is evaluated at the current point $x$, and the current vector $y$ is implicitly defined by (6.1). If we label $J^E$ consistent with the partition illustrated in (6.3),

$$J^E = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \tag{6.4}$$

then, assuming the uniqueness of the Newton step, matrix $B$ is nonsingular and the Newton step for the system $F(X) = 0$, at the current point, is

$$s_N = -(C - DB^{-1}A)^{-1}F. \tag{6.5}$$

Indeed, at point $x$, the Jacobian of $F$ is the (Schur-complement) matrix $J = (C - DB^{-1}A)$, where all quantities are evaluated at $x$. Note that despite any possible sparsity present in any of the matrices $A$, $B$, $C$, $D$ the Jacobian $J$ is almost surely dense due to the application of $B^{-1}$. However, in many real applications matrix $B$, especially, will be very sparse and it is cost effective in these cases to compute $s_N = \delta x$ by solving the larger (but very sparse) system (6.2), taking advantage of both sparsity and block-structure in $J^E$. Users can refer to [16] for more details.

We illustrate the 'structure' ideas with the following example.

Consider the autonomous ODE,

$$y' = f(y),$$

supposing $y(0) = f(x0)$, we use an explicit one-step Euler method to compute an approximation $y_k$ to a desired final state $y(T) = \phi(u0)$. Thus, it leads to a recursive function,

$$\begin{array}{l} y_0 = x \\ \text{for } i = 1, \cdots, p \\ \qquad \text{Solve for } y_i : \ y_i - F(y_{i-1}) = 0 \\ \text{Solve for } z : \ z - y_p = 0, \end{array} \tag{6.6}$$

where $F(y_i) = y_i + h \cdot f(y_i)$ and $h$ is the step size of the Euler method. The corresponding expanded function is

$$F^E(x, y_1, \cdots, y_p) = \begin{pmatrix} y_1 - F(y_0) \\ y_2 - F(y_1) \\ \vdots \\ y_p - F(y_{p-1}) \\ y_p \end{pmatrix}.$$

The subsequent Newton process can be written as,

$$J^E \begin{pmatrix} \delta x \\ \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_p \end{pmatrix} = -F^E = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ -G \end{pmatrix}, \tag{6.7}$$

where $J^E$ is the Jacobian matrix of $F^E(x, y_1, \cdots, y_p)$ with respect to $(x, y_1, \cdots, y_p)$ and $G$ is the function in nonlinear equation $G(x) = 0$, that is $y - \phi(u0)$ in this example.

Below is the description of newton_expand. The subsequent example illustrates the use of this function to solve the above ODE.

**Description of** newton_expand

[x, fval, it] = newton_expand (func, func_y, funG, x0, tol, itNum, Extra, p)

**Input arguments**

"func" is the expanded function, $F^E$, whose intermediate variables $y_1, ..., y_p$ are treated as independent variables; a string.
"func_y" is the function, $F$, which returns intermediate variables, $y_1, \cdots, y_p$ with input argument $x$; a string
"funG" is function $G$ (the right hand side of the Newton process); a string function
"x0" is the initial value of $x$; a vector
"tol" is the stopping tolerance; a scalar
"itNum" is the maximum number of iterations; a scalar
"Extra" stores parameters for function func; a MATLAB cell structure
"p" is the number of intermediate variables; a scalar

**Output arguments**

"x" is the computed solution; a vector
"fval" is function value at the solution; a scalar
"it" is the number of iterations; a scalar

## Example 6.2.1. Solve the ODE using newton_expand.

*See DemoNewton_Exp.m*

All functions called in DemoNewton_Exp.m can be found in Demos\Chapter 6 directory.

1. Set some initial values.

```
>> p = 5;     % number of intermediate variables
>> N = 8;     % size of original problem
```

```
>> tol = 1e-13;     % convergence tolerance
>> itNum = 40;     % maximum number of iterations
>> h = 1e-8;     % step size
```

2. Initialize the random seed.

```
>> rand('seed', 0);
```

3. Set the target function's input point $xT$.

```
>> xT = rand(N,1)
xT =
   0.2190
   0.0470
   0.6789
   0.6793
   0.9347
   0.3835
   0.5194
   0.8310
```

4. Set the expanded function $F^E$, $F$ and $G$.

```
>> func = 'Exp_DS';     % Expanded function with independent variables, x,
>>                      % y1, y2,...., yM
>> func_y = 'func_DS';     % function revealing relation between x, y1,y2,...,yM
>> funG = 'Gx';          % function on the right hand side of Newton process
```

5. Set parameters required in func = 'Exp_DS' and func_y.

```
>> Extra.N = N;        % size of original problem
>> Extra.M = p;        % number of intermediate variables
>> Extra.u0 = xT;      % input variable for the target function φ(xT)
>> Extra.fkt = @fx;    % function f on the right hand side of the ODE
>> Extra.phi = @exp;   % target function φ
>> Extra.h = h;     % step size of one step Euler method.
```

   Note that, we set the right hand side function of the ODE to be $f(x) = x$ so that the actual solution of the above ODE is $y = e^x$.

6. Set the starting point $x$.

```
>> x = ones(n,1);
```

7. Solve the ODE by the expanded Newton computation.

```
>> [x, fval, it] = newton_expand(func, func_y, x, tol, itNum, Extra, p)
x =

   1.2448
   1.0482
   1.9716
   1.9725
   2.5464
   1.4674
   1.6810
   2.2955
fval =
    1.0415e-015
it =
    2
```

8. Compare the computed solution with the target function value.

```
>>  The difference between computed solution with the target function value
norm(x-exp(xT)) = 2.605725e-007
```

## Example 6.2.2. Newton step comparisons.

*See DemoRawExp.m*

Consider the composite function, $F(x) = \bar{F}(A^{-1}\tilde{F}(x))$, where $\bar{F}$ and $\tilde{F}$ are Broyden functions [7] (their Jacobian matrices are tridiagonal) and the structure of $A$ is based on 5-point Laplacian defined on a square $(\sqrt{n}+2)$-by-$(\sqrt{n}+2)$ grid. For each nonzero element of $A$, $A_{ij}$ is defined as a function of $x$, specifically, $A_{ij} = x_j$. Thus, the nonzero elements of $A$ depend on $x$; the structure of $A_x \cdot v$, for any $v$, is equal to the structure of $A$.

The evaluation of $z = F(x)$ is a structured computation, $F^E(x, y_1, y_2)$, defined by the following three steps:

$$\left.\begin{array}{lll} (1) \quad \text{Solve for } y_1 & : & y_1 - \tilde{F}(x) = 0 \\ (2) \quad \text{Solve for } y_2 & : & Ay_2 - y_1 = 0 \\ (3) \quad \text{Solve for } z & : & z - \bar{F}(y_2) = 0 \end{array}\right\}. \tag{6.8}$$

Differentiating $F^E$ defined by (6.8) with respect to $x,\ y_1,\ y_2$, yields

$$J^E = \begin{bmatrix} -\tilde{J} & I & 0 \\ A_x y_2 & -I & A \\ 0 & 0 & \bar{J} \end{bmatrix}. \tag{6.9}$$

The Newton step can be obtained by solving

$$J^E \begin{pmatrix} \delta x \\ \delta y_1 \\ \delta y_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -F(x) \end{pmatrix}. \tag{6.10}$$

In this example, we consider two approaches to the Newton step computation.

- **Straight Newton computation**. In this approach, the structure is not exploited. Specifically, the Jacobian matrix $J$ is formed by differentiating $F$ using the forward-mode automatic differentiation(AD) (equivalent in cost to obtaining $J$ column-by-column using forward finite-differences)[4]. Finally, the dense system $Js_N = -F$ by using the Matlab linear system solver '\'.
- **Structured Newton computation**. This approach involves forming $J^E$ (6.9) via structured automatic differentiation, i.e.. Then the expanded system (6.10) is solved by using the Matlab linear system solver '\'.

Figure 6.1 plots the running times in seconds of one single step of the two Newton approaches. The experiment was carried out using Matlab 6.5 (R13) on a laptop with Intel 1.66 GHz Duo Core CPU and 1GB RAM. All the matrices in the experiments are sparse except for the matrix $J$. Clearly, the Newton step computation is greatly accelerated by exploiting the structure of $F$.

The structured Newton concept can also be applied to solving minimization problems. Suppose that the (sufficiently) smooth minimization problem,

$$\min_x f(x) \tag{6.11}$$

yields a corresponding Newton step with respect to the gradient,

$$s_N = -H^{-1}(x)\nabla f(x). \tag{6.12}$$

The gradient, $\nabla f(x)$, is the $n$-by-1 matrix of first derivatives,

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix} \tag{6.13}$$

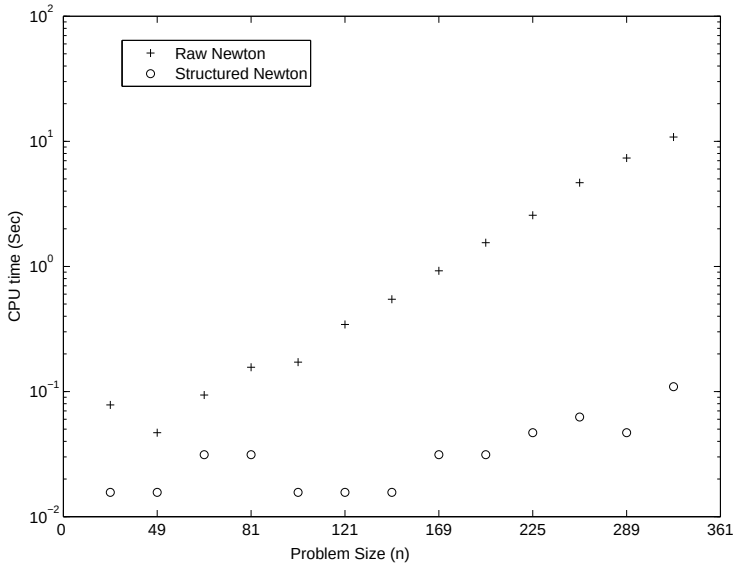


Figure 6.1: Computation times of one step of the straight Newton computation and one step of the structured Newton computation.

i.e., $\nabla f^T$ is the Jacobian of $f$; $H(x)$ is the Hessian matrix, i.e., the symmetric matrix of second derivatives of $f$:

$$H(x) = \nabla^2 f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix} \tag{6.14}$$

Following the form of (6.1) we define a structured scalar valued function: $z = f(x)$:

$$\left.\begin{array}{lcl} \text{Solve for } y & : & \tilde{F}^E(x, y) = 0 \\ \text{“Solve” for } z & : & z - \bar{f}(x, y_1, \cdots, y_p) = 0 \end{array}\right\} \tag{6.15}$$

where $\tilde{F}^E = \tilde{F}^E(x, y_1, y_2, \cdots, y_p) = \begin{pmatrix} \tilde{F}_1^E(x, y_1, \cdots, y_p) \\ \tilde{F}_2^E(x, y_1, \cdots, y_p) \\ \vdots \\ \tilde{F}_p^E(x, y_1, \cdots, y_p) \end{pmatrix}$ and $y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{pmatrix}$.
Note that each $y_i,\ i = 1, \cdots, p$, is itself a vector (varying lengths). By our structured assumption, $\tilde{F}^E$ represents a triangular computation:

$$\left.\begin{array}{rcl} \text{Solve for } y_1 & : & \tilde{F}_1^E(x, y_1) = 0 \\ \text{Solve for } y_2 & : & \tilde{F}_2^E(x, y_1, y_2) = 0 \\ & \vdots & \\ \text{Solve for } y_p & : & \tilde{F}_p^E(x, y_1, y_2, \cdots, y_p) = 0 \end{array}\right\} \tag{6.16}$$

Similar to the Newton step computation for systems of nonlinear equations, a larger but sparse system based on differentiating (6.15, 6.16) can be solved in order to obtain the Newton step (6.12). The analogy to system (6.2) is, solve

$$H^E \begin{pmatrix} \delta w \\ \delta y \\ \delta x \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -\nabla_x f \end{pmatrix} \tag{6.17}$$

where $s_N$, the Newton step defined in (6.12), satisfies $s_N = \delta x$, and $H^E$ is a symmetric Hessian matrix,

$$H^E = \begin{pmatrix} 0 & \tilde{F}_y^E & \tilde{F}_x^E \\ (\tilde{F}_y^E)^T & (\tilde{F}_{yy}^E)^T w + \nabla_{yy}^2 \bar{f} & (\tilde{F}_{yx}^E)^T w + \nabla_{yx}^2 \bar{f} \\ (\tilde{F}_x^E)^T & (\tilde{F}_{xy}^E)^T w + \nabla_{xy}^2 \bar{f} & (\tilde{F}_{xx}^E)^T w + \nabla_{xx}^2 \bar{f} \end{pmatrix} \tag{6.18}$$

and $w$ is the (vector) solution to the (typically) sparse system,

$$(\tilde{F}_y^E)^T w = -\nabla_y \bar{f}. \tag{6.19}$$

Using (6.19), $H^E$ in (6.18) can be rewritten

$$H^E = \begin{pmatrix} 0 & \tilde{F}_y^E & \tilde{F}_x^E \\ (\tilde{F}_y^E)^T & (\tilde{F}_{yy}^E)^T w + \nabla_{yy}^2 \bar{f} & 0 \\ (\tilde{F}_x^E)^T & 0 & (\tilde{F}_{xx}^E)^T w + \nabla_{xx}^2 \bar{f} \end{pmatrix}.$$

If we permute the columns and rows of $H^E$

$$H_P^E = \begin{bmatrix} \tilde{F}_x^E & \tilde{F}_y^E & 0 \\ 0 & (\tilde{F}_{yy}^E)^T w + \nabla_{yy}^2 \bar{f} & (\tilde{F}_y^E)^T \\ (\tilde{F}_{xx}^E)^T w + \nabla_{xx}^2 \bar{f} & 0 & (\tilde{F}_x^E)^T \end{bmatrix} = \left[\begin{array}{c|c} \text{A} & \text{L} \\ \hline \text{B} & \text{M} \end{array}\right],$$

then the Hessian matrix of $f$ is $H = B - ML^{-1}A$.

The key point is that a cost-effective alternative to forming the Hessian matrix $H$ and in turn solving $Hs_N = -\nabla_x f$ is to compute $H^E$ via *sparse* AD or finite-difference technology and then solve (6.17) with a sparse solver; $s_N = \delta x$.

## Example 6.2.3 Structured Newton computation for solving a minimization problem.

*See DemoExpHess.m*

We minimize $z = \bar{F}(\tilde{F}(x)) + x^T x$, where $\tilde{F}(x)$ is Broyden function, and $\bar{F}(y)$ is a scalar-valued function, $\bar{F}(y) = \sum_{i=1}^{n} y_i^2 + \sum_{i=1}^{n-1} 5 y_i y_{i+1}$. The triangular computation system (6.15) of $z$ can be constructed as follows,

$$\begin{cases} \text{Solve for } y & : \quad \tilde{F}^E(x,y) = y - \tilde{F}(x) = 0, \\ \text{“Solve” for } z & : \quad z - \bar{f}(x,y) = z - [\bar{F}(y) + x^T x] = 0. \end{cases}$$

The corresponding expanded Hessian matrix $H^E$ is,

$$H^E = \begin{pmatrix} 0 & -\tilde{J} & I \\ -\tilde{J}^T & \nabla_{yy}^2 \bar{f} & 0 \\ I & 0 & (\tilde{F}_{xx}^E)^T w + \nabla_{xx}^2 \bar{f} \end{pmatrix},$$

where $I$ is an $n$-by-$n$ identity matrix, and $w$ is equal to $-\nabla_y(\bar{f})$. Thus, the one step of structured Newton computation for this minimization problem can be written as follows.

1. Initial value of the problem size.

```
>> n = 25;      % problem size
>> I = eye(n);
```

2. Initialization. Variable 'tildeFw' represents the inner product, $(\tilde{F}^E)^T w$.

```
>> TildeF = 'broyden';
>> TildeFw = 'tildeFw';
>> BarF = 'barF';
```

3. Initialize x and values in Extra.

```
>> xstart = rand(n,1);
>> Extra.w = ones(n,1);
>> Extra.y = ones(n,1);
>> Extra.n = n;
>>  m = length(broyden(x));
```

4. Compute the sparsities of the constituent Jacobian and Hessian matrices.

```
>> JPI = getjpi(TildeF, m);
>> hpif = gethpi(BarF, n, Extra);
>> hpiFw = gethpi(TildeFw, n,Extra);
```

5. One step of structured Newton computations.

   (a) Compute $y$, $\tilde{F}_x^E$ and $\tilde{F}_y^E = I$.

   ```
   >> [y, FEx] = evalj(TildeF, x, [], [], JPI);
   ```

   (b) Compute $\nabla_y \bar{f}$ and $\nabla^2_{yy} \bar{f}$.

   ```
   >> Extra.y = x;
   >> [z, grady, H2]= evalH(BarF, y, Extra, hpif);
   ```

   (c) Set $w$ to $-\nabla_y \bar{f}$.

   ```
   >> Extra.w = -grady(:);
   ```

   (d) Compute the function value, gradient and Hessian of $(\tilde{F}^E)^T w$ with respect to $x$ .

```
>>  Extra.y = y;
>> [z, grad, Ht] = evalH(TildeFw, x, Extra, hpiFw);
```

(e) Construct the expanded Hessian matrix $H^E$.

```
>> HE(1:n, n+1: 3*n) = [I, -FEx];
>> HE(n+1:3*n, 1:n) = [I; -FEx'];
>> HE(n+1:2*n, n+1:2*n) = H2;
>> HE(2*n+1:3*n, 2*n+1:3*n) = Ht+2*I;
```

(f) Compute the function value and gradient of the original function.

```
>> myfun = ADfun('OptmFun', 1);
>> [nf , gradx] = feval(myfun, x, Extra);
```

(g) Solve the Newton system and update $x$.

```
>> HE = sparse(HE);
>> d = -HE  [zeros(2*n,1); gradx(:)];
>> x = x + d(2*n+1:3*n);
```

Note that this section only gives samples of structured Newton computation for solving nonlinear equations and minimization problems. Users can refer to [16] for details about its advantages, applications, performances and parallelism.

# Chapter 7

# Using ADMAT with the MATLAB Optimization Toolbox

The MATLAB optimization toolbox includes solvers for many nonlinear problems, such as multidimensional nonlinear minimization, nonlinear least squares with upper and lower bounds, nonlinear system of equations, and so on. Typically, these solvers use derivative information such as gradients, Jacobians, and Hessians. In this chapter, we illustrate how to conveniently use ADMAT to accurately and automatically compute derivative information for use with the MATLAB Optimization Toolbox.

## 7.1 Nonlinear Least Squares Solver 'lsqnonlin'

MATLAB provides a nonlinear least squares solver, "lsqnonlin ", for solving nonlinear least squares problems,

$$\min \|F(x)\|_2^2 \qquad \text{s.t.} \qquad l \leq x \leq u,$$

where $F(x)$ maps $R^m$ to $R^n$ and $\|\ \|$ is 2-norm. By default this solver employs the finite difference method to estimate gradients and Hessians (unless users specify an alternative). ADMAT, for example, can be specified as an alternative to finite differences. Users just need to set up a flag in input argument 'options' without changing any original codes. The following examples illustrate how to solve nonlinear least squares with ADMAT used to compute derivatives. For the details of input and output arguments of the MATLAB solver, 'lsqnonlin', please refer to MATLAB help documentation.

## Example 7.1.1. Solving a nonlinear least squares problem using ADMAT to compute the Jacobian matrix.

*See DemoLSq.m*

Solve the nonlinear least squares problem,

$$\min \|F(x)\|_2^2 \quad \text{s.t.} \quad l \leq x \leq u,$$

where $F(x)$ is the Broyden function defined in §3.2.

1. Set problem size.

   ```
   >> n = 5;
   ```

2. Initialize the random seed.

   ```
   >> rand('seed' ,0);
   ```

3. Initialize x.

   ```
   >> x0 = rand(n,1)
   x0 =
      0.2190
      0.0470
      0.6789
      0.6793
      0.9347
   ```

4. Set the lower bound for x.

   ```
   >> l = -ones(n,1);
   ```

5. Set the upper bound for x.

   ```
   >> u = ones(n,1);
   ```

6. Get the default value of 'options' of MATLAB 'lsqnonlin' solver.

   ```
   >> options = optimset('lsqnonlin');
   ```

7. Turn on the Jacobian flag in input argument 'options'. This means that the user will provide the method to compute Jacobians (In this case, the use of ADMAT).

```
>> options = optimset(options, 'Jacobian', 'on');
```

8. Set the function to be differentiated by ADMAT. The function call 'feval' is overloaded by the one defined in ADMAT, which returns the function value and Jacobian on each 'feval' call (See Chapter 3.2).

```
>> myfun = ADfun('broyden', n);
```

9. Call 'lsqnonlin' to solve the nonlinear least squares problem using ADMAT to compute derivatives.

```
>> [x, RNORM] = lsqnonlin(myfun, x0, [],[], options)
x =
   -0.1600
    0.2584
    0.9885
    1.0000
    0.2120
RNORM =
    0.7375
```

## 7.2 Multidimensional Nonlinear Minimization Solvers 'fmincon' and 'fminunc'

Consider a multidimensional constrained nonlinear minimization problem,

$$\begin{array}{lll} & \min f(x) & \\ \text{s.t.} & Ax \leq b, \quad Aeq * x = B & \text{(linear constraints)} \\ & c(x) \leq 0, \quad ceq(x) \leq 0 & \text{(nonlinear constraints)} \\ & l \leq x \leq u, & \end{array}$$

where $f(x)$ maps $R^n$ to a $R^1$. The MATLAB solver for this problem is 'fmincon'.

The unconstrained minimization problem is simply

$$\min f(x),$$

where $f(x)$ maps $R^n$ to a scalar. This unconstrained problem can be solved by 'fminunc'. ADMAT can be used to compute the gradient or both the gradient and the Hessian matrix.

### Example 7.2.1 Solve unconstrained nonlinear minimization problems using ADMAT.

*See DemoFminunc.m*

Solve the nonlinear minimization problem,

$$\min brown(x),$$

where $brown(x)$ is the Brown function defined in §3.1. In this example, we will solve the problem twice. In the first time, ADMAT is used to compute gradients only, and Hessians are estimated by the default finite difference method. In the second solution, ADMAT is used to compute both gradients and Hessians.

1. Set problem size.

   ```
   >> n = 5;
   ```

2. Initialize random seed.

   ```
   >> rand('seed', 0);
   ```

3. Initial value of x.

   ```
   >> x0 = rand(n,1)
   x0 =
      0.2190
      0.0470
      0.6780
      0.6793
      0.9347
   ```

4. Set the function to be differentiated by ADMAT. The function call 'feval' is overloaded by the one defined in ADMAT. It can return the function value, gradient and Hessian in each 'feval' call (See §3.1 for details).

   ```
   >> myfun = ADfun('brown',1);
   ```

5. Get the default value of 'options'.

   ```
   >> options = optimset('fminunc');
   ```

6. Solve the problem using ADMAT to determine gradients (but not Hessians).

   (a) Turn on the gradient flag in input argument 'options' (but not the Hessian flag). Thus, the solver will use ADMAT to compute gradients, but will estimate Hessians by finite difference method.

   ```
   >> options = optimset(options, 'GradObj', 'on');
   ```

   (b) Call the MATLAB constrained nonlinear minimization solver 'fmincon' with ADMAT used to determine gradients only.

   ```
   >> [x,FVAL] = fminunc(myfun,x0,options)
   x =
     1.0e − 004 *
       −0.1487
       −0.1217
         0.2629
       −0.0173
       −0.5416
   FVAL =
     4.8393e − 009
   ```

7. Solve the problem using ADMAT to compute both gradients and Hessians.

   (a) Turn on both gradient and Hessian flags in input argument 'options'. Thus, the solver will use the user specified method (ADMAT) to compute both gradients and Hessians.

   ```
   >> options = optimset(options, 'GradObj', 'on');
   >> options = optimset(options, 'Hessian', 'on');
   ```

   (b) Call the MATLAB constrained nonlinear minimization solver 'fmincon' using ADMAT to compute derivatives.

   ```
   >> [x,FVAL] = fminunc(myfun,x0,options)
   x =
     1.0e − 004 *
   ```

$-0.1487$
$-0.1217$
$0.2629$
$-0.0173$
$-0.5416$

FVAL =

$4.8393e-009$

**Example 7.2.2 Solve the constrained nonlinear minimization problems using ADMAT.**

*See DemoFmincon.m*

Solve the nonlinear minimization problem,

$$\min brown(x), \quad l \leq x \leq u,$$

where $brown(x)$ is the Brown function defined in Chapter 3.

1. Set problem size.

   >> n = 5;

2. Initialize the random seed.

   >> rand('seed', 0);

3. Set initial value of x.

   >> x0 = rand(n,1)
   x0 =
   0.2190
   0.0470
   0.6789
   0.6793
   0.9347

4. Set the lower bound for x.

   >> l = -ones(n,1);

5. Set the upper bound for x.

   >> u = ones(n,1);

6. Set the function to be differentiated by ADMAT.

   >> myfun = ADfun('brown',1);

7. Get the default value of 'options'.

   >> options = optimset('fmincon');

8. Set 'options' so that both gradients and Hessians are computed by ADMAT.

   >> options = optimset(options, 'GradObj', 'on');
   options = optimset(options, 'Hessian', 'on');

9. Call MATLAB constrained nonlinear minimization solver 'fmincon' with ADMAT to compute derivatives.

   >> [x,FVAL] = fmincon(myfun,x0,[],[],[],[],l,u,[], options)
   x =
   $1.0e-007$ *
   $-0.0001$
   $0.0000$
   $-0.1547$
   $-0.1860$
   $0.0869$
   FVAL =
   $1.2461e-015$

In summary, ADMAT can be conveniently linked to MATLAB nonlinear least squares solver 'lsqnonlin' and nonlinear minimization solvers 'fminunc' and 'fmincon' in two steps:

1. Set up the Jacobian, gradient and Hessian flags in the input argument 'options'.
2. Set the function to be differentiated by ADMAT using the 'ADfun' function call to overload 'feval'.

# Chapter 8

# Combining C/Fortran with ADMAT

ADMAT can differentiate any function defined in an M-file. ADMAT cannot be applied to any external files, such as MEX files. However, ADMAT can be combined with finite-differencing to enable M-file/external file combinations.

### Example 8.1.1 Compute the Jacobian of the Broyden function (which is programmed in C.)

*See mexbroy.c and CBroy.m*

The C file, mexbroy.c, for the Broyden function and its MEX function are as follows.

```
/*************************************************************
%
%              Evaluate the Broyden nonlinear equations test function.
%
%
%      INPUT:
%           x - The current point (row vector).
%
%
%      OUTPUT:
%           y - The (row vector) function value at x.
%
*************************************************************/
```

```
#include "mex.h"
#define x_IN 0
#define y_OUT 0

extern void mexbroy(int n, double *x, double *y)
{
     int i;
     y[0] = (3.0-2.0*x[0])*x[0]-2.0*x[1]+1.0;
     y[n-1] = (3.0-2.0*x[n-1])*x[n-1]-x[n-2]+1.0;

     for(i=1; i<n-1; i++)
          y[i]= (3.0 - 2.0*x[i])*x[i]-x[i-1]-2.0*x[i+1] + 1.0;

}

// MEX Interface function
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray, *prhs[])
{
     // define input and output variables
     double *x, *y;
     int n;
     if (nrhs != 1)
          mexErrMsgTxt("One input required");
     else if(nlhs >1)
          mexErrMsgTxt("Too many output argument");

     n = mxGetM(prhs[x_IN]);
     if (n == 1)
          n = mxGetN(prhs[x_IN]);

     plhs[y_OUT] = mxCreateDoubleMatrix(n, 1, mxREAL);

     x = mxGetPr(prhs[x_IN]);
     y = mxGetPr(plhs[y_OUT]);

     mexbroy(n, x, y);

     return;
}
```

Note that mexFunction is an interface to integrate C/Fortran functions with MATLAB. It transfers input and output arguments between MATLAB and C/Fortran. After defining the Broyden and its MEX function, type 'mex mexbroy.c' in the MATLAB prompt to compile the MEX file.

Once the compilation succeeds, the Broyden function can be called as a MATLAB function. File CBroy.m integrates mexbroy.c file into ADMAT via finite-differencing.

```
function y = CBroy(x, Extra)
% y = CBroy(x, Extra)
%
%       compute the broyden function at x by C subroutine, mexbroy.c
%       Mapping, CBoy : R^n - - - - > R^n
%
%
%       INPUT:
%            x - The current point (column vector). When it is an
%                 object of deriv class, the fundamental operations
%                 will be overloaded automatically.
%            Extra - Parameters required in CBroy.
%
%
%       OUTPUT:
%            y - The (vector) function value at x. When x is an object
%                 of deriv class, y will be an object of deriv class as well.
%                  There are two fields in y. One is the function value
%                  at x, the other is the Jacobian matrix at x.
%
%
%       This is an example of how to use the finite difference method to add
%       existing C subroutine into ADMAT package. Users can call this function
%       as any forward mode functions in ADMAT .
%
%
%                       Cayuga Research
%                            July 2008
%

global globp;
```

```
global fdeps;

n = length(x);
if isa(x, 'deriv')              % x is an objective of deriv class
     val = getval(x);           % get the value of x
     drv = getydot(x);          % get the derivative part of x
     y = mexbroy(val);          % compute the function value at x
     ydot = zeros(getval(n), globp);     % initialize the derivative of y
     % compute the derivative by finite difference method
     for i = 1 : globp
          tmp = mexbroy(val + fdeps*drv(:,i));
          ydot(:,i) = (tmp - y)/fdeps;
     end

     % set y as an objective of deriv class
     y =deriv(y, ydot);
else
     y = mexbroy(x);
end
```

The global variable fdeps is the step size used in finite differencing. Its default value is 1e-6. Users can specify their own step size.

The final finite-difference calculation is illustrated below.

1. Set problem size.

   ```
   >> n = 5;
   ```

2. Define a deriv input variable.

   ```
   >> x = ones(n,1);
   >> x = deriv(x, eye(n));
   ```

3. Compute the Jacobian by finite differencing.

   ```
   >> y = CBroy(x)
   val =
   ```

```
     0
    -1
    -1
    -1
     1
deriv =
   -1.0000   -2.0000         0         0         0
   -1.0000   -1.0000   -2.0000         0         0
         0   -1.0000   -1.0000   -2.0000         0
         0         0   -1.0000   -1.0000   -2.0000
         0         0         0   -1.0000   -1.0000
```

4. Extract Jacobian matrix from the finite differencing.

```
>> JFD = getydot(y)
JFD =
   -1.0000   -2.0000         0         0         0
   -1.0000   -1.0000   -2.0000         0         0
         0   -1.0000   -1.0000   -2.0000         0
         0         0   -1.0000   -1.0000   -2.0000
         0         0         0   -1.0000   -1.0000
```

# Chapter 9

# Troubleshooting

Below we list some potential problems that may occur in the use of ADMAT.

1. Conversion to double from deriv is not possible.

   This usually means a deriv class object is assigned to a double class variable. Check both sides of the assignment and make sure both sides are of the same type.

2. Error using ==> XXX
   Function 'XXX' is not defined for variables of class 'deriv'.

   Some MATLAB functions are not overloaded in ADMAT yet. Please contact Cayuga Research for extending ADMAT to the MATLAB function of your interest.

3. Undefined function or variable 'deriv'.

   ADMAT is not installed yet. Please refer to Chapter 2 to make sure ADMAT is properly installed.

4. Error using ==> deriv/deriv
   Please restart ADMAT. There may be a license problem.

   If you got the following message,

   "Please contact Cayuga Research for a license extension",

   it means the license for ADMAT 2.0 expired. Please contact us for renewing license.

5. Do not use Matlab command "clear all" to clear your workspace while using ADMAT. This will remove all ADMAT global variables from memory: unpredictable errors may then occur. Instead, use "clear" selectively as needed.

6. ADMAT cannot perform 3-D or higher operations. ADMAT only performs the 1-D and 2-D matrix operations.

7. Derivatives are incorrect. Please make sure following issues was checked.

   - "clear all" was not called before using the ADMAT.
   - Make the data type of the dependent variable consistent with that of the input independent variable in user-defined function (See §3.4 for details).

   If there is still an error, please contact Cayuga Research for further help.

# Appendix A

# Applications of ADMAT

In this chapter we illustrate two applications of ADMAT. First, we show how to trigger the quasi-Newton computation in MATLAB unconstrained nonlinear minimization solver 'fminunc' using ADMAT to determine gradients. Second, we present a sensitivity problem.

## A.1 Quasi-Newton Computation

The MATLAB multidimensional unconstrained nonlinear minimization solver, 'fminunc', uses the quasi-Newton approach when the user chooses the medium-scale option (to classify the size of the problem being solved). This method updates the inverse of Hessian directly by the Broyden-Fletcher-Goldfarb-Shanno (BFGS) formula and, by default, estimates gradients by finite differencing. However, it allows users to specify their own gradients computation method to replace the finite difference method. The following example shows how to trigger the quasi-Newton computation in 'fminunc' with ADMAT used to determine gradients.

### Example A.1.1 Find a minimizer of the Brown function using the quasi-Newton approach in fminunc (with gradients determined by ADMAT.)

*See DemoQNFminunc.m*

1. Set the problem size.

   >> n = 5;

2. Initialize the random seed

   >> rand('seed',0);

3. Initialize x.

   >> x0 = rand(n,1)
   x0=
   0.2190
   0.0470
   0.6789
   0.6793
   0.9347

4. Set the function to be differentiated by ADMAT.

   >> myfun = ADfun('brown',1);
   Note: the second input argument in ADfun, '1', is a flag indicating a scalar mapping, $f : R^n \to R^1$; more generally, the second argument is set to '$m$' for a vector-valued function, $F : R^n \to R^m$.

5. Get the default value of argument 'options' .

   >> options = optimset('fmincon');

6. Turn on the gradient flag of input argument 'options'. Thus, the solver uses user-specified method to compute gradient (i.e., ADMAT).

   >> options = optimset(options, 'GradObj', 'on');

7. Set the flag of 'LargeScale' to off, so that the quasi-Newton method will be used.

   >> options = optimset(options, 'LargeScale', 'off');

8. Solve the constrained nonlinear minimization by the quasi-Newton method in 'fmincon' with ADMAT used to compute gradients.

   >> [x,FVAL] = fminunc(myfun,x0, options)
   x =
   1.0e-004 *

```
    −0.1487
    −0.1217
     0.2629
    −0.0173
    −0.5416
FVAL =
   4.8393e-009
```

## A.2 A Sensitivity Problem

This example is concerned with sensitivity with respect to problem parameters at the solution point. Consider a nonlinear scalar-valued function $f(x, \mu)$, where $x$ is the independent variable and $\mu$ is a vector of problem parameters. This example illustrates that ADMAT can be used to both compute derivatives with respect to $x$, in order to minimize $f(x, \mu)$ with respect to $x$ (for a fixed value of $\mu$), and analyze the sensitivity of $f(x_{opt}, \mu)$ with respect to $\mu$ at the solution point.

**Example A.2.1. Analyze the sensitivity of brownv$(x, V)$ function with respect to $\mu$ at the optimal point $x_{opt}$ with $V = 0.5$.**

*See DemoSens.m*

The function brownv$(x, V)$ is similar to the Brown function defined in §3.1. Its definition is as follows.

```
function f = brownv(x,V)
% length of input x
n=length(x);
% if any input is a 'deriv' class object, set n to 'deriv' class as

if isa(x, 'deriv') || isa(V, 'deriv')
    n = deriv(n);
end
y=zeros(n,1);
i=1:(n-1);
y(i)=(x(i). ^ 2). ^ (V*x(i+1). ^ 2+1)+((x(i+1)+0.25). ^ 2). ^ (x(i). ^ 2+1) + ...
    (x(i)+0.2*V). ^ 2;
f = sum(y);
tmp = V'*x;
f = f - .5*tmp'*tmp;
```

1. Set the problem size.

   >> n = 5;

2. Set the initial value of $V$ to 0.5. We first solve the minimization problem, $\min \text{brownv}(x, V)$, at $V = 0.5$, then analyze the sensitivity of $\text{brownv}(x_{opt}, V)$ with respect to $V$.

   >> V = 0.5

3. Initialize the random seed.

   >> rand('seed',0);

4. Set the initial value of $x$.

   ```
   >> x0 = 0.1*rand(n,1)
   x0 =
      0.2190
      0.0470
      0.6789
      0.6793
      0.9347
   ```

5. Solve the unconstrained minimization problem, $\min \text{brownv}(x, V)$, using the MATLAB optimization solver 'fminunc' and with ADMAT as a derivative computation method. (See §7.2 for details).

   ```
   >> options = optimset('fminunc');
   >> myfun = ADfun('brownv', 1);
   >> options = optimset(options, 'GradObj', 'on');
   >> options = optimset(options, 'Hessian', 'on');
   >> [ x1, FVAL1 ] = fminunc(myfun, x0, options, V);
   ```

6. Set the parameter $V$ to 'deriv' class.

   >> V = deriv(V,1);

7. Compute the function value of $\text{brownv}(x, V)$ at the optimal point $x1$ with 'deriv' class objective $V$.

   >> f = brownv(x1, V)

```
val =
    0.0795
deriv =
    -0.0911
```

8. Get the sensitivity of $V$ at optimal point $x_{opt}$.

```
>> sen = getydot(f)
sen =
    -0.0911
```

In summary, analyzing the sensitivity $f(x, \mu)$ with respect to $\mu$ at $x = x_{opt}$ requires two steps.

- Solve the minimization problem of $f(x, \mu)$ with respect to $x$ at a fixed value $\mu$.
- Differentiate the function $f(x, \mu)$ with respect to $\mu$ at the optimal point $x_{opt}$ to get the sensitivity of $f(x_{opt}, \mu)$ with respect to $\mu$.

# Bibliography


[1] autodiff.org, *www.autodiff.org*, 2008.

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

[3] C. H. Bischof, B. Lang and A. Vehreschild, *Automatic differentiation for MATLAB programs*, Proceedings in Applied Mathematics and Mechanics, 2003, 50–53

[4] C. H. Bischof, A. Carle, G. F. Corliss, A. Griewank and P. D. Hovland, *ADIFOR: generating derivative codes from Fortran programs*, Scientific Programming, Vol. 1, 1992, 11–29.

[5] C. H. Bischof, A. Carle, P. Khademi and A. Mauer, *ADIFOR 2.0: automatic differentiation of Fortran 77 programs*, IEEE Computational Science and Engineering, Vol. 3, 1996, 18–32.

[6] C. H. Bischof L. Roh and A. Mauer, *ADIC — an extensible automatic differentiation tool for ANSI-C*, Software–Practice and Experience, Vol. 27, 1997, 1427–1456.

[7] C.G. Broyden, *A class of methods for solving nonlinear simultaneous equations*, Mathematics of Computaions, Vol. 19, No.92, 1965, 577-593.

[8] T.F. Coleman and G.F. Jonsson, *The efficient computation of structured gradients using automatic differenciation*, SIAM J. Sci. Comput. Vol. 20, 1999, 1430–1437.

[9] T.F. Coleman and J.J. Moré, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. Numer. Anal., Vol. 20, No.1, 1983, 187-209.

[10] T.F. Coleman and J.J. Moré, *Estimation of sparse Hessian matrices and graph coloring problems* , Math Programming, Vol. 28, 1984, 243-270.

[11] T. F. Coleman and A. Verma, *Structure and efficient Hessian calculation*, Advances in Nonlinear Programming, Yaxiang Yuan(ed.), 1998, 57-72.

[12] T. F. Coleman and A. Verma, *ADMAT: An automatic differentiation toolbox for MATLAB*, Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-Operable Scientific and Engineering Computing, SIAM, Philadelphia, PA, 1998.

[13] T. F. Coleman and A. Verma, *The eifficient computation of sparse Jacobian matrices using automatic differentiation*, SIAM, J. Sci. Comput., Vol. 19, 1998, 1210-1233.

[14] T. F. Coleman and A. Verma, *ADMIT-1: Automatic differentiation and MATLAB interface toolbox*, ACM Transactions on Mathematical Software, Vol. 26, 2000, 150-175.

[15] T. F. Coleman, F. Santosa and A. Verma, *Semi-Automatic differentiation*, Proceedings of Optimal Design and Control Workshop, VPI, 1997.

[16] T. F. Coleman and W. Xu, *Fast Newton computations*, SIAM J. Sci. Comput., Vol.31, 2008, 1175-1191.

[17] S. A. Forth, *An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB*, ACM Transactions on Mathematical Software, vol. 32, 2006, 195–222.

[18] A. Griewank, *Some bounds on the complexity gradients*, Compleity in Nonlinear Optimization, P. Pardalos, Ed. World Scientific Publishing Co., Inc., River Edge, NJ, 1993, 128-161.

[19] A. Griewank and G.F. Corliss, Eds, *Automatic Differentiation of Algorithms: Theory, Implementation and Applications*, SIAM, Philadelphia, PA, 1991.

[20] A. Griewank, D. Juedes and J. Utke, *Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++*, ACM Transactions on Mathematical Software, vol 22, 1996, 131–167.

[21] Mathematics and Computer Science Division, Argonne National Laboratory, Center for High Performance Software Research, Rice University and Computational Engineering Research Group, RWTH Aachen, Germany, *http://www-unix.mcs.anl.gov/OpenAD*, 2008.

[22] The MathWorks Inc., 3 Apple Hill Drive, Natick MA 01760-2098, *C and Fortran API Reference* , July 2008, http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/apiref.pdf.

[23] C. W. Straka, *ADF95: Tool for automatic differentiation of a FORTRAN code designed for large numbers of independent variables*, Computer Physics Communications, Vol. 168, 2005, 123–139.

[24] S. Stamatiadis, R. Prosmiti and S. C. Farantos, *auto_deriv: Tool for automatic differentiation of a fortran code*, Comput. Phys. Commun., Vol. 127, 2000, 343-355.