A Brief Guide for Using ADMAT™ 2.0 Professional

This guide provides an introduction to the use of ADMAT 2.0 Professional (henceforth referred to as ADMAT). Please consult our User's Guide for a detailed discussion on the full spectrum of functionalities and usage of ADMAT.

ADMAT belongs to an "operator overloading" class of automatic differentiation tools and uses object-oriented programming features in MATLAB. Thus, ADMAT requires MATLAB 6.5 or above.

1. Installation of ADMAT

1.1 Installation Instructions for Windows Users

ADMAT is provided in a zip file **ADMAT-2.0.zip**. Place the zip file in a proper directory and unzip it. There are two methods for setting the MATLAB search paths for ADMAT.

Method 1:

1. Click "File" in MATLAB window.
2. Choose the "Set Path" option.
3. Click the "Add with Subfolders" button.
4. Find the directory for ADMAT in the "Browse for Folders" window and click "OK".
5. Click the "Save" button to save the paths for ADMAT and click the "Close" button.
6. Type "startup" at the MATLAB prompt or exit MATLAB and re-start MATLAB.

Method 2:

1. Access the ADMAT directory.
2. Edit the **startup.m** file to add all subdirectories of ADMAT to the MATLAB search path manually.
3. Type "startup" at MATLAB prompt to set up the paths for ADMAT.

If there is a "success" message, ADMAT is correctly installed.

1.2 Installation Instructions for Unix or Linux Users

Unzip **ADMAT-2.0.zip** using "unzip ADMAT-2.0.zip" at the Unix or Linux prompt. Then follow Method 2 described above.

2. Computation of Gradients, Jacobians, and Hessians

2.1 Computing Gradients

Compute the gradient of the Brown function. Please refer to Section 3.1 of the User's Guide for a definition of the Brown function.

1. Set the problem size n, where n is the number of independent variables.

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

2. Set the Brown function as the function to be differentiated.

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

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

3. Initialize the independent variable *x*.

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

4. Call **feval** to get the function value and the gradient of the Brown function, allowing ADMAT to decide whether the forward mode or the reverse mode is used. By default, ADMAT uses reverse mode to compute the gradients. For more on "modes", please consult our User's Guide.

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

f =
   8
grad =
    4  8  8  8  4
```

Note that the gradient *"grad"* is a row vector.

2.2 Computing Jacobians

Compute the Jacobian of the Broyden function. Please refer to Section 3.2 of the User's Guide for a definition of the Broyden function.

1. Set the problem size n, where n is the number of independent variables.

```
>> n = 5
```

2. Set the Broyden function 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 number of independent variables (i.e. the column dimension in the Jacobian).

3. Initialize the independent variable *x*.

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

4. Call **feval** to compute the function value and the Jacobian matrix at *x*.

```
>> [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
```

In the above example, we have shown how to compute a square Jacobian matrix. In the following we will illustrate how to compute a rectangular Jacobian matrix. We choose the Variably Dimensioned Function (VDF) from the More-Garbow-Hillstrom collection as an example. The VDF function mapping R^n to R^{n+2} is defined in an M-file "VDF.m" as follows.

```
function y = VDF(x, Extra)

n = length(x);
y = zeros(n+2,1);
y(1:n) = x-1;

tmp = 0;
for i=1:n
   tmp = tmp+i*(x(i)-1);
end

y(n+1) = tmp;
y(n+2) = tmp^2;
```

1. Set the problem size n, where n is the number of independent variables.

```
>> n = 5
```

2. Set VDF as the function to be differentiated.

```
>> myfun = ADfun('VDF', n+2);
```

3. Initialize the independent variable *x*.

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

4. Call **feval** to compute the function value and the Jacobian matrix at *x*.

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

F =
   0
```

```
     0
     0
     0
     0

J =
     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
     1     2     3     4     5
     0     0     0     0     0
```

2.3 Computing Gradients and Hessians

Compute the gradients and Hessians of the Brown function.

1. Set the problem size n, where n is the number of independent variables.

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

2. Set the Brown function as the function to be differentiated.

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

3. Initialize the independent variable *x*.

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

4. Call **feval** to get the function value, gradient and Hessian of the Brown function at point *x*.

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

f =
     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
```

The function to be differentiated is specified in an M-file. The interface of the function must contain just one output argument and exactly two input arguments. For example,

```
y = FOO(x, Extra),
```

where y is the only output, x is the independent variable and Extra is a MATLAB cell structure which stores all remaining parameters required by the function FOO. If a function to be differentiated requires just one input parameter, Extra can be omitted or Extra can be assigned the empty array, '[ ]'. An example for illustrating the use of Extra can be found in Section 3.1 of the User's Guide.

3. Sparse Jacobian and Hessian Computation

If Jacobians (or Hessians) are sparse and they are evaluated many times at different points, users can use the graph coloring technique implemented in ADMAT to accelerate the Jacobian or Hessian computation.

3.1 Sparse Jacobian Computation

Compute the Jacobian matrix of an arrowhead function. Please see Example 4.1.1 in Section 4.1 of the User's Guide for the definition of arrowhead function. The procedure to evaluate the Jacobian *J* of the arrowhead function at point $x = [1\ 1\ 1\ ...\ 1\ 1]'$ is as follows.

1. Set the problem size n, where n is the number of independent variables.

```
>> n = 500;
```

2. Initialize the independent variable *x*.

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

3. Compute the sparsity information of the Jacobian matrix of the arrowhead function and store it in JPI.

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

4. Compute the function value and the Jacobian matrix (in MATLAB sparse format) of the arrowhead function at point *x* based on the computed sparsity information stored in JPI. The input argument 'Extra' is set to '[ ]' since it is empty.

```
>>  [F, J] = evalj('arrowfun', x, [], n, JPI)
F =
     502
     2
     2
     2
     2
     :
     :
J =
   (1,1)       6
   (2,1)       2
   (3,1)       2
   (4,1)       2
   (5,1)       2
   (6,1)       2
     :         :
```

: :

3.2 Sparse Hessian Computation

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

1. Set the problem size n, where n is the number of independent variables.

```
>> n = 500;
```

2. Set the independent variable *x*.

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

3. Compute the sparsity information of the Hessian of the Brown function and store it in HPI.

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

4. Compute the function value, gradient and Hessian of the Brown function at point *x* based on the computed sparsity information stored in HPI. The input argument 'Extra' is set to '[ ]' since it is empty.

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

v =

998

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) 24
: :
: :

4. Advanced Features of ADMAT

ADMAT provides users with additional functionalities -- forward mode AD and reverse mode AD so that derivative can be computed with even greater flexibility and ease.

4.1 Forward Mode AD

Using the forward mode AD provided in ADMAT, a user can easily compute the first derivatives of functions that are defined using arithmetic operations and intrinsic functions of MATLAB. Furthermore, the forward mode AD provides users with greater flexibility than just using **feval**. Users can define their own MATLAB functions as usual. There is no restriction on the number of input arguments a user-defined function may have. If there is more than one input argument, the derivative with regard to any input argument can be computed by the forward mode AD without any change to the function definition.

In this section, we will give an example on the use of the forward mode AD for computing the Jacobian matrix of Broyden function at point $x = [1\ 1\ 1]$.

1. Create a **deriv** class object.

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

   val =
        1   1   1
  deriv =
        1    0    0
        0    1    0
        0    0    1
```

2. Evaluate the Broyden function at point *x*.

```
>>  y = broyden(x)

val =
     0   -1    1
deriv =
    -1   -2    0
    -1   -1   -2
     0   -1   -1
```

3. Get the value of the Broyden function at point *x*.

```
>>  yval = getval(y)

   yval =
        0   -1    1
```

4. Get the Jacobian matrix of the Broyden function at point *x*.

```
>> J = getydot(y)

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

4.2 Reverse mode AD

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

The following example shows how to compute the transpose of the Jacobian matrix J^T of the Broyden function at point $x = [1, 1, 1]$ by using the reverse mode AD.

1. Create a **derivtape** class object.

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

val =
     1  1  1
varcount =
     1
```

2. Evaluate the Broyden function at point *x*.

```
>>  y = broyden(x)

val =
     0
    -1
     1
varcount =
    40
```

3. Get the value of the Broyden function at point *x*.

```
>>  yval = getval(y)

yval =
     0
    -1
     1
```

4. Get the transpose of the Jacobian of the Broyden function at point *x*.

```
>>  JT = parsetape(eye(3))

JT =
   -1   -1    0
   -2   -1   -1
    0   -2   -1
```

5. Troubleshooting

Below we list several 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 statement and make sure that they are of the same data type.

2. Error using ==> XXX

Function XXX has not been defined for variables of class **deriv**. A number of MATLAB functions have not been overloaded in ADMAT yet. Please contact Cayuga Research for extending ADMAT to the MATLAB functions of your interest.

3. Undefined function or variable deriv

ADMAT has not been installed yet. Please make sure that ADMAT is properly installed.

4. Error using ==> deriv/deriv
ADMAT detects a possible license error. Please restart ADMAT.

5. The ADMAT 2.0 license has expired. Please contact Cayuga Research for a license extension.

6. Do not use MATLAB command **clear all** to clear your workspace while using ADMAT. This would remove all ADMAT global variables from memory: unpredictable errors may then occur. Instead, use **clear** selectively as needed.

7. ADMAT 2.0 only performs 1-D and 2-D matrix operations. In other words, it cannot perform 3-D or higher-dimension operations.

8. The computed derivatives are incorrect. Please check the following issues.

- The command **clear all** should not be called while using ADMAT.
- The data type of the dependent variable must be consistent with that of the input independent variable in a user-defined function (See Section 3.4 in the User's Guide for details).

Finally, if there is still a problem persists, please contact Cayuga Research for assistance.