ADMAT : An Automatic Differentiation Toolbox for MATLAB

Thomas F. Coleman[†] Arun Verma[‡]

May 14, 1998

Abstract

ADMAT enables you to differentiate target functions defined via M-files. ADMAT is implemented using the operator overloading technology in MATLAB (version 5.0 and above) [6] and can compute derivatives of upto second order. ADMAT can be used as a plug-in tool for ADMIT-1 [3, 1] and ADMIT-2 [2] toolboxes, enabling the computation of sparse Jacobian and Hessian matrices and derivatives of structured computations.

1 Introduction

Numerical solutions of large scale nonlinear problems involve computing the derivative information in form of gradients, Jacobian and Hessian matrices, often repeatedly making the computation of derivatives a central part of the solution process. Also it turns out that efficiency of solution of nonlinear optimization and nonlinear equations problems depends directly on accurate and efficient derivative computation making it one of the most computationally challenging part of the solution process.

ADMAT enables you to differentiate MATLAB functions, and allows to compute gradients, Jacobian matrices and Hessian matrices of nonlinear maps defined via M-files.

This is the first ever AD tool written for differentiating M-files. This tool belongs to the "operator overloading" class of AD tools and uses the Object Oriented Programming feature present in MATLAB 5 for implementation. A direct consequence of this being that ADMAT can be used only with MATLAB 5.

ADMAT can be used as a plug-in tool for ADMIT-1 and ADMIT-2 toolboxes, enabling the computation sparse Jacobian and Hessian matrices and derivatives of structured computations written as MATLAB functions. Here is a simple ADMIT-1 example which uses ADMAT for computing the sparse Jacobian of a simple test problem which has an arrowhead Jacobian sparsity pattern :

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

        y=x.*x;
        y(1)=y(1)+x'*x;
        y=y+x(1)*x(1);
```

Assume this program is saved in file myfun.m. To evaluate the function F and the Jacobian J at $x' = (1, 1, ..., 1)$ for $n = 5$, and then display the structure of J:

```
>> x=ones(5,1);
```

[†]Computer Science Department and Center for Applied Mathematics, Cornell University, Ithaca NY 14850.

[‡]Computer Science Department, Cornell University, Ithaca NY 14850.

```
>> [f,J] =evalJ('myfun',x);
```

As illustrated by this example, all the details of the plug-in AD tool (ADMAT here) are completely hidden from the the users of ADMIT-1; making it a very user friendly tool.

This document is organized as follows. In §2, we present the basics of an AD tool based on overloaded methods. In §3 we present the high-level software design of ADMAT, followed by the basics of doing AD of a matlab-like language in §5. In §6, we present all the implementation details and usage of ADMAT. In §8, we present some insights on parallelization of AD of matrix vector operations.

For more information about ADMAT, please refer to our website : http://www.cs.cornell.edu/home/verma/AD/research.html.

2 Basics of a OOPS based AD tool

There are mainly two ways to implement an automatic differentiation tool. One kind is source to source transformation tool, and the other are object oriented AD tools. For more on this subject, refer to survey article on AD tools [**?**].

In this section, we mainly describe how an AD tool based on object oriented technology works. Another such tool is ADOLC [5].

A typical OOPS based AD tool defines all the variables in the program to be **active**. These active variables carry the value of the variable as well as the derivative information. The actual computational statements of the user provided code need not be altered for the purposes of automatic differentiation. All arithmetic operations, as well as the comparison and assignment operators, are overloaded, so any or all of their operands can be an active variable. E.g., ADOL-C overloads all the mathematical functions contained in the ANSI C standard for the math library are overloaded for active arguments. Similarly, ADMAT overloads all the basic MATLAB functions such as "+", "-", "sqrt", "mpower ($\wedge$)" for general matrix and vector active arguments.

Functioning of such an AD tool can be illustrated best using a flowchart. The flowchart shown in figure 1 corresponds to the following simple MATLAB program, we have given simple scalar values to the variables for the purpose of illustration.

```
function y = getfun(x)

        z = x*x;
        z=x+z;
        y=z*z;
```

x is the independent variable.

The values of the derivatives is propagated along with the values of the variables as shown in the flowchart. The value variables being represented by x, z, w, y and the derivatives by $dx/dx, dz/dx, dw/dx.dy/dx$.

3 Software design of ADMAT

The design of the ADMAT toolbox is as shown in figure 2.

ADMAT is designed as a three-layer toolbox, with the layer on top inheriting from and adding to the functionality to the bottom layer. The core of the ADMAT toolbox is the class deriv, which is the basic forward mode computing engine of ADMAT. The

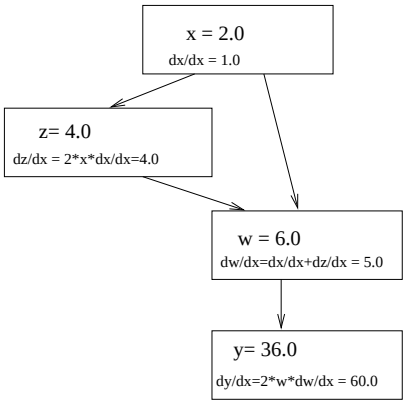

FIG. 1. *Flowchart corresponding to a simple program*

layer above deriv contains two classes, namely, derivtape which is the basic reverse mode computing engine of ADMAT and derivspj which is the Jacobian sparsity computing engine of ADMAT. The topmost layer which concerns the computation of second order derivatives also consists of two classes, class derivtapeH for computing Hessian vector products and class derivsph for computing sparsity pattern of Hessians.

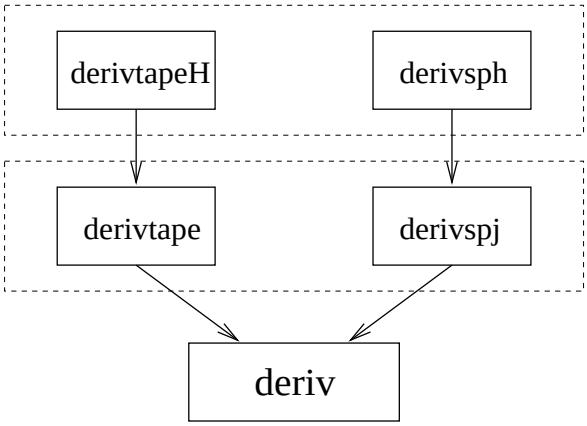

FIG. 2. *Design of ADMAT toolbox*

4 Functionality of ADMAT

ADMAT has both reverse and forward mode capability and can compute derivatives of upto second order. In summary, ADMAT provides the following five functionality features :

1. **Vector valued functions :**
   (a) **Jacobian-Matrix (forward) product**. $(F, x, V) \rightarrow J(x)V$.
   (b) **Matrix-Jacobian (reverse) product**. $(F, x, W) \rightarrow J(x)^T W$.
   (c) **Jacobian Sparsity Pattern**. $F \rightarrow SPJ$.
2. **Scalar valued functions :**
   (a) **Hessian-Matrix product**. $(f, x, V) \rightarrow H(x)V$.
   (b) **Hessian Sparsity Pattern**. $f \rightarrow SPH$.

ADMAT can be used as a plug-in tool for ADMIT-1, enabling computation of sparse Jacobians and Hessians of target M-files, and with ADMIT-2, allowing the computation of derivatives of structured computations.

For computation of derivatives of target M-files, ADMAT is recommended to be used in conjunction with ADMIT-1 which provides the user a very high-level interface for computing sparse Jacobian and Hessian matrices and as well as the lower-level derivative functionality, e.g. the tangent and adjoint products, JV, J^TW. It can also be used stand-alone providing access to the five functionalities mentioned above; the stand-alone usage of ADMAT is illustrated in §6.

5 Differentiating Matrix Vector Operations

In §2 we reviewed the basics of an Object Oriented AD tool. ADMAT is an AD tool for MATLAB and hence it overloads all the basic matrix-vector operations present in MATLAB.

Thinking about AD in terms of high-level matrix vector operations as opposed to the scalar level operations has a lot of advantages, e.g. it saves the storage in the reverse mode, where you have to just save the high level vectors instead of all the elementary intermediate variables. To illustrate, consider the example of dot product of two vectors: At the elementary level it is coded using a for loop :

```
z=0;
for i = 1: n
    z = z + x(i)*y(i);
end
```

The above code generates $n + 1$ extra intermediate variables, which don't exist if we work at the matrix-vector level.

Even more glaring example is that of matrix matrix product between two matrices of size $n \times n$. The elementary code generates n^3 intermediate variables, so the straightforward reverse mode will generate extra space complexity of n^3 variables. But the matrix-level reverse mode won't generate any extra space requirements, and the amount of space required will be $O(n^2)$.

There are other insights gained by this high-level view, e.g. information about parallelization of derivative code. We present a basic treatment of parallelizing AD of matrix vector operations in §8.

5.1 Forward mode

No we present the basic ideas involved in Automatic differentiation of a high-level language like MATLAB. We overload all the elementary functions(MATLAB builtin functions in this case, e.g. exp, sum, +, - etc), which not only compute the "value" of the output, but also update "derivative" of output consistently using chain rule to propagate taylor coefficients. In table 1 we present a listing of how we handle some of the matrix vector operations. $\dot{z}$ corresponds to the forward product $\frac{\partial z}{\partial I} \cdot V$, I denotes the independent variables and V denote the initial tangent direction, $\dot{I} = V$.

For a vector $x \in \Re^{n\times 1}$, the forward product $\dot{x}$ is of size $n \times p$, where p is the number of columns in V. $\dot{x}(:, i)$ denotes the derivative in the ith tangential direction. For a matrix $A \in \Re^{m\times n}$, the forward product $\dot{A}$ is a tensor of size $m \times n \times p$. $\dot{A}(:, :, i)$ denotes the derivative in the ith tangential direction.

Operation	Tangent Rule
$z = x^T y$	$\dot{z} = \dot{x}^T y + \dot{y}^T x$
$z = x + y$	$\dot{z} = \dot{x} + \dot{y}$
$z = x. * y$	$\dot{z}(:,i) = \dot{x}(:,i). * y + \dot{y}(:,i). * x$
$y = Ax$	$\dot{y}(:,i) = \dot{A}(:,:,i)x + A\dot{x}(:,i)$
$y = A\backslash x$	$\dot{y}(:,i) = A\backslash(\dot{x}(:,i) - \dot{A}(:,:,i)y)$
$C = A + B$	$\dot{C} = \dot{A} + \dot{B}$
$C = A * B$	$\dot{C}(:,:,i) = \dot{A}(:,:,i)B + A * \dot{B}(:,:,i)$
$C = A. * B$	$\dot{C}(:,:,i) = \dot{A}(:,:,i). * B + A. * \dot{B}(:,:,i)$
$C = A./B$	$\dot{C}(:,:,i) = \dot{A}(:,:,i)./B - A. * (\dot{B}(:,:,i)./B^2)$

TABLE 1

Tangent propagation rules

5.2 Reverse mode

Now we present the rules for propagation of adjoints. Again all the elementary functions are overloaded for this purpose. In table 2 we present a listing of how we handle the computation of adjoints for some of the matrix vector operations. z^* corresponds to the adjoint product $\frac{\partial O}{\partial z}^T \cdot W$, O denotes the output variables and W denote the initial adjoint direction, $\dot{O} = W$.

For a vector $x \in \Re^{n\times 1}$, the adjoint x^* is of size $n \times p$, where p is the number of columns in W. $x^*(:,i)$ denotes the derivative in the ith adjoint direction. For a matrix $A \in \Re^{m\times n}$, the adjoint A^* is a tensor of size $m \times n \times p$. $A^*(:,:,i)$ denotes the derivative in the ith adjoint direction.

Operation	Adjoint Rule
$z = x^T y$	$x^* = z^* * y, \quad y^* = z^* * x$
$z = x + y$	$x^* = z^*, \quad y^* = z^*$
$z = x. * y$	$x^* = z^* * diag(y), \quad y^* = z^* * diag(x)$
$y = Ax$	$x^* = A^T * z^*, \quad A^*(:,j,:) = x(j). * z^*$
$y = A\backslash x$	$x^* = A^T\backslash y^*, \quad A^*(:,:,i) = -(A^T\backslash y^*)(:,i)y^T$
$C = A + B$	$A^* = C^*, \quad B^* = C^*$
$C = A * B$	$A^*(:,:,i) = B^T * C^*(:,:,i), \quad B^*(:,:,i) = A^T * C^*(:,:,i)$
$C = A. * B$	$A^*(:,:,i) = C^*(:,:,i). * B, \quad B^*(:,:,i) = C^*(:,:,i). * A$
$C = A./B$	$A^*(:,:,i) = C^*(:,:,i)./B, \quad B^*(:,:,i) = -C^*(:,:,i). * A./(B. * B)$

TABLE 2

Adjoint propagation rules

Reverse mode at this high-level saves considerable amount of space complexity, in operations $x^T y$, Ax, $A\backslash x$, $A * B$. Particularly In the operations involving $n \times n$ matrices, the reduction can be an order of magnitude.

5.3 Sparsity Pattern computation

One of the major functions of ADMAT tool is that it can compute the sparsity patterns of Jacobian and Hessian matrices automatically. The sparsity pattern of Jacobian matrix can be propagated exactly the forwprod products. In table 3 a listing of the

rules similar to the propagation of forward products.

Assume that the size of the independent vector I is $n \times 1$. For an intermediate vector $v \in \Re^{n_v \times 1}$, the Jacobian sparsity pattern J_v is of size $n_v \times n$. For an intermediate matrix $A \in \Re^{m_A \times n_A}$, the Jacobian sparsity pattern J_A is a tensor of size $m_A \times n_A \times n$.

Operation	Sparsity pattern Rule
$z = x^T y$	$J_z = \sum J_{x(i)} + J_{y(i)}$
$z = x + y$	$J_z = J_x + J_y$
$z = x. * y$	$J_z = J_x + J_y$
$y = Ax$	$J_y(i) = \sum J_{A(i,:))} * x + A(i,:) * J_x$
$y = A \backslash x$	$J_y = A \backslash J_x - J_A * x$
$C = A + B$	$J_C = J_A + J_B$
$C = A * B$	$J_C = J_A * B + A * J_B$
$C = A. * B$	$J_C = J_A + J_B$
$C = A./B$	$J_C = J_A + J_B$

TABLE 3

Jacobian sparsity pattern propagation rules

5.3.1 Computing Hessian sparsity pattern The sparsity pattern of Hessian matrix is slightly more complex. Here we need to propagate the sparsity patterns of Jacobians (Ist order derivatives) together with the Hessian sparsity patterns (2nd order derivatives). Hence we need chain rule propagation from 2nd order taylor series.

In table 4 a listing of the rules for the propagation of Hessian sparsity patterns. Here H_z denotes the sparsity pattern of Hessian of z, and J_z denotes the sparsity pattern of the Jacobian(gradient) of z. Propagation of J_z is governed as specified in the previous section .

Assume that the size of the independent vector I is $n \times 1$. For an intermediate vector $v \in \Re^{n_v \times 1}$, the Jacobian sparsity pattern J_v is of size $n_v \times n$. For an intermediate matrix $A \in \Re^{m_A \times n_A}$, the Jacobian sparsity pattern J_A is a tensor of size $m_A \times n_A \times n$.

Operation	Sparsity pattern Rule
$z = x + y$	$H_z = H_x + H_y$
$z = x. * y$	$H_z = H_x + H_y + J_x J_y^T + J_y J_x^T$
$C = A + B$	$H_C = H_A + H_B$
$C = A. * B$	$H_C = H_A + H_B + J_A J_B^T + J_B J_A^T$
$C = A./B$	$H_C = H_A + H_B + J_A J_B^T + J_B J_A^T$

TABLE 4

Hessian sparsity pattern propagation rules

Once we have defined all these rules to propagate the forward product, reverse product, Jacobian and Hessian sparsity pattern for general matrix vector operations, its possible to compute the derivative matrix products, JV, $J^T W$, HV and sparsity patterns for general M-functions, making ADMAT eligible as a plug-in tool for ADMIT-1.

6 Implementation of ADMAT

In this section we provide implementation details as well as the usage of ADMAT by including some easy examples.

6.1 Forward Mode

6.1.1 Description of the deriv class Class deriv is an extension of double (regular MATLAB variables belong to class double). All the variables in user's computation belong to this class(deriv) in the "AD-mode" (instead of the usual double variables in the regular computation.)

deriv class has two fields, val and deriv which stand for the value of the variable and the derivative(generally speaking) respectively. For a deriv variable x, $x.value$ is the value of x, and $x.derivative$ is used to represent the derivative of this value $\dot{x}$ w.r.t a chosen set of independent variables.

All the elementary functions(MATLAB builtin functions in this case, e.g. exp, sum, +, - etc) are overloaded for deriv class. which not only compute the "value" of the output, but also update "derivative" of output consistently using chain rule to propagate taylor coefficients.

6.1.2 Methods

deriv

Purpose

This is the constructor function for the deriv class.

Synopsis

y=deriv(x)

y=deriv(x,V)

Description

y=deriv(x) If x is a double variable, y is a deriv variable with the value of x, and the derivative field set to zero. If x is a deriv variable, it is returned without change.

y=deriv(x,V) Derivative field of y is set to V.

getval

Purpose

Returns the value field of the deriv variable.

Synopsis

y=getval(x)

Description

y=getval(x)
y is set to the value field of the deriv variable x.

getydot

Purpose

Returns the derivative field of the deriv variable.

Synopsis

y=getydot(x)

Description

y=getydot(x)
y is set to the derivative field of the deriv variable x.

6.1.3 Example The following steps illustrate the way to use the deriv class in ADMAT to compute the tangential derivative. The example function used is the broyden's nonlinear function.

- Define input point – x=ones(N,1).
- Make x belong to deriv class, and Initialize the seed matrix – xdot=eye(N); x=deriv(x,xdot).
- Compute the function (as well as the derivatives via overloading) – y=broy1a(x).
- Value of y, val=getval(y), Derivative (or product JV) = J since V(xdot) =eye(N), JV=getydot(y).

Methods of deriv class

- **Constructor :**

```
function s= deriv(a)
%
% Derivative class for AD of M-files
 global globp;
if nargin==0
   s.val=0;
   s.deriv=zeros(1,globp);
   s=class(s,'deriv');
elseif isempty(a)
   s.val=[];
   s.deriv=[];
   s=class(s,'deriv');
elseif isa(a,'deriv')
   s=a;
else
   s.val=a;
   [m,n]=size(a);
   if ((m==1) &(n==1))
      s.deriv=zeros(1,globp);
   elseif (m==1)
      s.deriv=zeros(n,globp);
   elseif (n==1)
      s.deriv=zeros(m,globp);
   else
      s.deriv=zeros(m,n,globp);
   end
   s=class(s,'deriv');
end
```

- **Method for addition :**

```
function sout=plus(s1,s2)
         if ( isa(s1,'deriv')) s1=deriv(s1); end
         if ( isa(s2,'deriv')) s2=deriv(s2); end
         sout.val=s1.val+s2.val;
         sout.deriv=s1.deriv+s2.deriv;
         sout=class(sout,'deriv');
```

For a given function $F(x) : \Re^n \rightarrow \Re^m$, an AD tool in forward mode can compute the Jacobian-Matrix product $JV = \frac{dy}{dx}V$. Computing this product using deriv class is easy, we just need to assign $\dot{x} = V$, and by definition $\dot{y}$ which comes out of this computation will be exactly JV.

6.2 Implementation of reverse(adjoint) mode

To implement the reverse mode, the AD tool implements a tape, which records **all** the intermediate values and operations performed in the function evaluation. Computation of adjoints is done by a reverse pass on the tape, and at the end of the pass the adjoints of independent variables are picked up from the front of the tape.

6.2.1 Description of derivtape class For a given function $F(x) : \Re^n \rightarrow \Re^m$, an AD tool in adjoint mode can compute the Matrix-Jacobian product $WJ = W^T \frac{dy}{dx}$. For this, we need rules for propagation of adjoints.

6.2.2 Methods

derivtape

Purpose

This is the constructor for the derivtape class.

Synopsis

y=derivtape(x)

Description

y=derivtape(x) If x is a double variable, y is a derivtape variable with the value of x. If x is a derivtape variable, it is returned without change.

parsetape

Purpose

This function computes the adjoint product of a functions once the tape is created using derivtape overloaded methods.

Synopsis

WJ=parsetape(W)

Description

WJ=parsetape(W)
WJ is the adjoint product given the matrix W.

6.2.3 Example The following steps illustrate the way to use the derivtape class in ADMAT to compute the adjoint derivative. The example function used is the broyden's nonlinear function.

- Define input point – x=ones(N,1).
- Make x belong to derivtape class – x=derivtape(x).
- Compute the function and create tape (taping every intermediate via overloading), y=broy1a(x).
- Initialize the adjoint seed matrix – W = eye(N).
- Parse and process the tape backwards to compute $J^T * W$ – parsetape(W).
- Grab the adjoint from the front end of the tape – JtW=tape(1).W.

6.3 Computing Hessian Matrix products

For a scalar function $f(x) : \Re^n \rightarrow \Re$, we need to compute HV where H is the Hessian matrix of $f(x)$. Computing HV combines the forward and reverse modes.

$\nabla^2 fV = (\frac{d((\nabla f)^T V)}{dx})^T$.

Compute $w = (\nabla f)^T V$ by forward mode and then $(\frac{dw}{dx})^T$ by full reverse mode since w has less number of variables than x.

6.3.1 Methods

derivtapeH

Purpose

This is the constructor for the derivtapeH class.

Synopsis

y=derivtapeH(x)

Description

y=derivtapeH(x)

If x is a double variable, y is a derivtapeH variable with the value of x. If x is a derivtapeH variable, it is returned without change.

6.3.2 Example The following steps illustrate the way to use the derivtapeH class in ADMAT to compute the Hessian matrix times vector product. The example function used is the brown nonlinear function.

- Define input point – x=ones(N,1).
- Make x belong to derivtapeH class – x=derivtapeH(x).
- Forward mode : Compute the function+first derivatives and create tape . y=brown1(x) – computes $grad(f)^T * V$ and creates tape for subsequent reverse mode.
- Reverse Mode : Parse and process the tape backwards to compute $H * V$, ' do parsetape(eye(size(V,2))) – since the output of forward mode is same size as number of columns in V.
- Grab the adjoint from the front end of the tape – HV=tape(1).W.

6.4 Computing Jacobian sparsity pattern

6.4.1 The derivspj class description For computing the sparsity pattern of the Jacobian, the AD tool uses a different class called derivspj. This time the sparsity pattern of the gradient of each intermediate value is propagated using the methods in derivspj.

6.4.2 Methods

derivspj

Purpose

This is the constructor

Synopsis

y=derivspj(x)

y=derivspj(x,S)

Description

y=derivspj(x) If x is a double variable, y is a derivspj variable with the value of x, and the sparsity pattern field set to a empty matrix. If x is a derivspj variable, it is returned without change.

y=derivspj(x,S)
Sparsity pattern field of y set to a S.

6.4.3 Example The following steps illustrate the way to use the derivspj class in ADMAT to compute the Jacobian sparsity pattern. The example function used is the brown nonlinear function.

- Define dimension N=10.
- Define a dummy input point – x=rand(N,1).
- Make x belong to derivspj class, and Initialize the seed matrix – xdot=speye(N); x=derivspj(x,xdot).
- Compute the function (as well as the sparsity pattern via overloading) – y=broy1a(x).
- Grab the sparsity pattern off y : SPJ=getydot(y).

6.5 Computing Hessian sparsity pattern

6.5.1 Description of derivsph class For computing the sparsity pattern of the Hessian, the AD tool uses a different class called derivsph which builds on derivspj. This time the sparsity pattern of the gradient as well as the Hessian of each intermediate value is propagated using the methods in derivsph.

6.5.2 Methods

derivsph

Purpose

This is the constructor for the derivsph class.

Synopsis

y=derivsph(x)

y=derivsph(x,S)

Description

y=derivsph(x)
If x is a double variable, y is a derivsph variable with the value of x, and the sparsity pattern field set to a empty matrix. If x is a derivsph variable, it is returned without change.

y=derivsph(x,S)
Sparsity pattern field of y set to a S.

6.5.3 Example The following steps illustrate the way to use the derivsph class in ADMAT to compute the Hessian sparsity pattern. The example function used is the brown nonlinear function.

- Define dimension N=10.
- Define a dummy input point – x=rand(N,1).

- Make x belong to derivtapeH class – x=derivtapeH(x). (Sets xdot = I, xdoubledot = 0
- Compute the function (as well as the sparsity pattern via overloading) – y=brown1(x).
- Grab the sparsity pattern off y : SPH=getydot(y).

6.6 Complete list of Matlab's elementary functions

```
abs.m        cos.m        floor.m      log10.m      power.m      spy.m
acos.m       cosh.m       ge.m         log2.m       prod.m       sqrt.m
acosh.m      cot.m        getval.m     lt.m         qr.m         subsasgn.m
acot.m       coth.m       getydot.m    lu.m         rank.m       subsindex.m
acoth.m      csc.m        gt.m         max.m        rdivide.m    subsref.m
acsc.m       csch.m       horzcat.m    min.m        real.m       sum.m
acsch.m      ctranspose.m imag.m       minus.m      rem.m        tan.m
and.m                     inv.m        mldivide.m   reshape.m    tanh.m
asec.m       diag.m       isfinite.m   mpower.m     round.m      times.m
asech.m         isinf.m      mrdivide.m   sec.m        transpose.m
asin.m           isnan.m      mtimes.m     sech.m       tril.m
asinh.m      eig.m        isnumeric.m  ndims.m      sign.m       triu.m
atan.m       eq.m         isreal.m     ne.m         sin.m        uminus.m
atanh.m      exp.m        issparse.m   norm.m       sinh.m       uplus.m
             eye.m        ldivide.m    not.m        size.m       vertcat.m
ceil.m       find.m       le.m         ones.m       sort.m       zeros.m
chol.m       fix.m        length.m     or.m         sparse.m
colon.m      fliplr.m     linspace.m   plot.m       spdiags.m
cond.m       flipud.m     log.m        plus.m       speye.m
```

7 The ADMAT "tape" or the computational graph

For the reverse propagation of derivatives, the whole execution trace of the original evaluation program must be recorded, unless it is recalculated as illustrated in [4]. In ADMAT, this potentially huge data set is written into a MATLAB structure which is referred to as the **tape**. The user may create several tapes (presumably corresponding to different functions user is dealing with) in several named arrays. During subsequent derivative evaluations, tapes are always accessed sequentially avoiding the overhead associated with overloaded methods of derivative object classes thus making the derivative evaluation process more efficient.

In ADMAT, the tape is designed as follows. The tape contains the complete execution trace of the computation. The tape is a computation graph, with a node corresponding to every intermediate variable. A node has the following fields :

1. **Op** : Stands for the arithmetic operation which generated this intermediate variable.
2. **Val** : Value of this intermediate variable.
3. **Arglist** : Pointers to nodes(other intermediate or input) variables involved in computation of this variable.
4. **deriv** : This field contains the associated derivative information of this intermediate variable. In forward mode computation (when using the tape), this will be the intermediate jacobian-matrix product JV, and in the reverse mode this field will contain the adjoint.

Figure 3 shows an example tape for the sample function described in §2(reproduced here for convenience) :

```
function y = getfun(x)

        z = x*x;
        z=x+z;
        y=z*z;
```

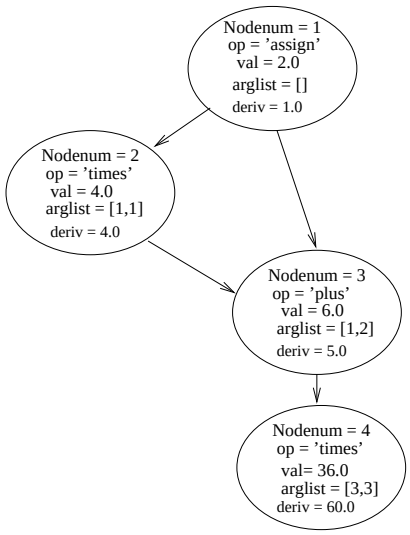

FIG. 3. *tape corresponding to a simple program*

ADMAT has methods to do reverse "sweeps" on the tape to compute the adjoint product. You don't need the tape for doing a forward sweep, but if you have already constructed the tape, then a forward sweep on the tape is more efficient than computing the forward product using operator overloading for each operation.

8 Making your own "fun"

The input argument x is a vector of dimension n; y is the output vector of dimension m. Extra is a 1-dimensional array corresponding to a 2-dimensional (full) matrix stacked column-by-column. The matrix represented by Extra is of size numrows-by-numcols.

The design of the target MATLAB function is as follows.

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

% Crunch
```

x is the input argument of dimension n, y is the output vector of dimension m, Extra is the extra user argument.

9 Parallelism in Matrix Vector Operations

We present some parallel implementation ideas here, which can be used to parallelize the AD of MATLAB like Matrix vector operations as presented in §4.

We have developed these ideas keeping in mind the potential implementation in MultiMATLAB.

We consider the setting of propagating the forward and reverse products where $W \in \Re^{m \times p}$ and $V \in \Re^{n \times p}$.

We look at some basic matrix vector operations and comment on the parallelization aspect. The cost equations can be defined by using three variables N the size of the vector (or $[M, N]$ the size of the matrix), NUMPROC the number of processors, and p the the column dimension of the product.

For the purpose of this section we make a basic assumption about parallel distribution of the value and the derivative data. A vector valued variable, has its value distributed among the processors, and its derivative $n \times p$ has each column distributed among the processors.

Here are the basic set of parallel matrix vector operations, the cost equations are based on :

- **Add(n) :** Stands for addition or subtraction operation between two n vectors.
- **Dot(n) :** Dot product of two n vectors.
- **Mul(n) :** Multiplication (element by element) of two n vectors.
- **Scale(n) :** Multiplication of an n vector by a scalar. For all purposes this operation is equivalent to Mul, so we will use Mul to represent this operation.

All matrix vector operations can be broken down in the terms of this basis of operations.

9.1 Parallelization of basic forward and reverse modes

- $z = x^T y$
  $\dot{z} = \dot{x}^T y + \dot{y}^T x.$
  $x_d^* += z_d^* * y.$
  $y_d^* += z_d^* * x.$
  Comments : implementation of the forward mode is similar to implementation of the computation it self. But the adjoint computation is more parallel. Consider the case where x and y are vectors, and z is a scalar – the adjoint computation is just a scaling operation.
  Function evaluation = 1 dot(n) Forward Mode = $2 \times p$ dot(n) + p add(n) = 2p * function evals. Reverse mode = $2 * p$ mul(n)
- $z = x + y$
  $\dot{z} = \dot{x} + \dot{y}.$ $x_d^* += z_d^*.$
  $y_d^* += z_d^*.$
  Both operations are embarrasingly parallel.
  Function evaluation = 1 add Forward Mode = p * adds function evals. Reverse mode = 2*p* adds
- $z = x. * y$
  $\dot{z}(:, i) = \dot{x}(:, i). * y + \dot{y}(:, i). * x.$
  $x_d^* += z_d^* * diag(y).$
  $y_d^* += z_d^* * diag(x).$
  Function evaluation = 1 mul Forward Mode = 2*p * mul function evals. Reverse mode = 2*p*scalings
- $y = Ax$
  $\dot{y}(:, i) = \dot{A}(:, :, i)x + A\dot{x}(:, i).$
  $x_d^* += A^T * z_d.$
  $A_d(:, j, :)^* += x(j). * z_d^*.$
  Depending on the shape of A, the parallel implementation of reverse and forward mode can be compared. E.g. if A is a single row, then this operation is equivalent to the dot product which we have discussed eariler. If A is square, both are equivalent to a square matrix times a vector,

if A has a single column, x is a scalar i.e., then forward mode is embarassingly parallel but reverse mode is a dot product.

function eval = 1 mul forward 1 outer prod + p mul reverse p dot +

- $y = A\backslash x$
  $\dot{y}(:,i) = A\backslash(\dot{x}(:,i) - \dot{A}(:,:,i)x)$.
  $x_d^* += A^T\backslash y_d^T$.
  $A_d(:,:,i)^* += -(A^T\backslash y_d^{*T})^T y_d^*(:,i)$.(outer product)
  complexity = solve

  It becomes more complicated for operations like solve. So we will simpify it by assuming that solve is a basic operation.
- $C = A + B$
  $\dot{C} = \dot{A} + \dot{B}$.
  $A_d^* += C_d^*$.
  $B_d^* += C_d^*$.
  Both are embarasssingly parallel.
- $C = A * B$
  $\dot{C}(:,:,i) = \dot{A}(:,:,i)B + A * \dot{B}(:,:,i)$.
  $A_d(:,:,i)^* += = B^T * C_d^*(:,:,i)$.
  $B_d(:,:,i)^* += = A^T * C_d^*(:,:,i)$.
  Again depending on dimensions, forward mode or reverse mode's paralleism can be different.
- $C = A. * B$
  $\dot{C}(:,:,i) = \dot{A}(:,:,i). * B + A. * \dot{B}(:,:,i)$.
  $A_d(:,:,i)^* += C_d(:,:,i)^*. * B$.
  $B_d(:,:,i)^* += C_d(:,:,i)^*. * A$.
- $C = A./B$
  $\dot{C}(:,:,i) = \dot{A}(:,:,i)./B - A. * (\dot{B}(:,:,i)./B^2)$.
  $A_d(:,:,i)^* += C_d(:,:,i)^*./B$.
  $B_d(:,:,i)^* += -C_d(:,:,i)^*. * A./B^2$.

Computation of gradient and hessian matrix product can be treated as special cases of forward and adjoint products. Hessian matrix product can be seen as forward product on the gradient. For the gradient, we can see paralleism of the reverse mode with $p = 1$.

9.2 Rules for prapagating Jacobian and Hessian sparsity pattern

This will involve sparse parallel linear algebra. The basic operations will be addition of two sparse matrices, reduction operations like summation of all rows.

10 ADMAT Caveats

x(1:end) doesn't work!!

- **MEX compiled files in MATLAB toolbox** . This AD tool can AD only the MATLAB code included in the M-file, and can't differentiate external code, e.g. MEX files. There are two options to handle MEX files, one is to employ Finite differencing – and the other is to do AD of Mex (C) source using external AD tools and integrate it with this tool.

  The MATLAB operations which are implemented in ADMAT using finite differencing include :

1. LU factorization
2. QR factorization
3. etc..

- **Higher dimensional matrices**
  ADMAT can handle only 1 and 2 dimensional matrices in Matlab 5.
- zeros, ones..

11 Conclusions

The capability of doing automatic differentiation of MATLAB opens up a wide range of applications which can use the facilities of AD in a much easier and quicker manner. A reason for this being that working in MATLAB domain makes it easier to write complicated applications mainly due to high-level nature of te language, but also due to the application specific toolboxes which are a part of MATLAB. With ADMAT, it is now possible to differentiate through MATLAB toolboxes, thus enabling push-the-button AD of complicated MATLAB applications.

We believe that an AD tool like ADMAT, should be an integral part of a MATLAB optimization and nonlinear solver.

References

[1] T. F. Coleman and A. Verma, *ADMIT-1: Automatic differentiation and matlab interface toolbox*, tech. rep., In preparation(for toms).

[2] ——, *ADMIT-2: Automatic differentiation and matlab interface toolbox for structured computation, User Guide*, tech. rep., in preparation.

[3] ——, *ADMIT-1: Automatic differentiation and matlab interface toolbox, User Guide*, Tech. Rep. CTC97TR271, Theory Center, Cornell University, 1997.

[4] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and Software, 1 (1992), pp. 35–54.

[5] A. Griewank, D. Juedes, and J. Utke, *ADOL–C, a package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. On Math. Software, 22 (1996), pp. 131–167.

[6] MATLAB 5.0 for UNIX, The Mathworks, Inc., 24 Prime Park Way, Natick, Massachusetts 01760.

A The Overloaded 'feval' interface

In this section we describe a new idea, which allows usage of AD technology in a very transparent manner. We have implemented this idea in the setting of LSOT using ADMIT for computation of derivatives (gradients, Jacobians and Hessians) with ADMAT as the plug-in AD tool.

The most important feature of this idea is that the optimization software interface remains the same irrespective of whether derivatives are computed via AD or provided by the user. Typically, the modern optimization software use a lot of plug-in calls to AD, which are simply replaced by universal feval calls, the call sequence remains the same whether using AD or user provided hand-coded derivatives.

Now we provide two illustrated examples of the usage of the overload feval interface – once each of a vector mapping and a scalar mapping.

- **Vector Mapping :**

```
>> myfun=ADfun('examplefun');
>>
>> x = ones(10,1);
>> y=feval(myfun,x);
>> [f,J]=feval(myfun,x);
>>
>> options=setopt('forwprod',ones(10,1));
>> [f,JV]=feval(myfun,x,[],options);
>>
>> options=setopt('revprod',ones(10,1));
>> [f,WJ]=feval(myfun,x,[],options);
>>
>> options=setopt('jacsp');
>> SPJ=feval(myfun,x,[],options);
>> spy(SPJ)
```

- **Scalar Mapping :**

```
>> mysfun=ADfun('examplesfun',1);  <-- a scalar problem
>>
>> x = ones(10,1);
>> v=feval(mysfun,x);
>> [v,grad]=feval(mysfun,x);
>> [v,grad,H]=feval(mysfun,x);
>>
>> options=setopt('htimesv',eye(10,2));
>> HV=feval(mysfun,x,[],options);
>>
>> options=setopt('hesssp');
>> SPH=feval(mysfun,x,[],options);
>> spy(SPH)
```

feval

Purpose

feval interface to all AD functionality.

fun represents a function written in Matlab as "y = function(x,Extra)"

Synopsis

f=feval(fun,x)

f=feval(fun,x,Extra)

[f,J]=feval(fun,x,Extra)

[f,grad]=feval(fun,x,Extra)

[f,grad,H]=feval(fun,x,Extra);

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

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

SPJ=feval(fun,x,Extra,options)

HV=feval(fun,x,Extra,options)

SPH=feval(fun,x,Extra,options)

Description

f=feval(fun,x)
Evaluates the function. Default Extra = []; See the class @fun for information to set up fun(Vector and scalar mappings).

f=feval(fun,x,Extra)
Takes the user provide Extra parameter.

[f,J]=feval(fun,x,Extra)
Also computes the sparse Jacobian (using ADMIT-1).

[f,grad]=feval(fun,x,Extra)
For scalar functions, computes the gradient.

[f,grad,H]=feval(fun,x,Extra)
For scalar functions, computes the gradient as well as the sparse Hessian.

[f,JV]=feval(fun,x,Extra,options)
Computes $J * V$. See setopt .

[f,WJ]=feval(fun,x,Extra,options)
Computes $J^T W$. See setopt .

SPJ=feval(fun,x,Extra,options)
Computes the sparsity pattern of Jacobian . See setopt .

HV=feval(fun,x,Extra,options) For scalar mappings. Computes $H * V$. See setopt .

SPH=feval(fun,x,Extra,options)
For scalar mappings. Computes sparsity pattern of Hessian. See setopt .

ADfun

Purpose

Prepares a matlab function for automatic derivative computation using ADMAT.

Synopsis

derivfun=ADfun(funstr)

derivfun=ADfun(funstr,scalar)

Description

derivfun=ADfun(funstr)

e.g myfun=ADfun('examplefun'), and then use myfun in all overloaded feval calls (AD calls). Default scalar = 0, i.e. the function is treated as a vector mapping. Also see feval.

derivfun=fun(funstr,scalar)

scalar =0 function funstr is a Vector Mapping.

scalar =1 function funstr is a scalar Mapping.

setopt

Purpose

Sets the options parameter for the overloaded feval interface.

Synopsis

options=fun(ADfunc, val)

Description

options=fun(ADfunc, val)

ADfunc='forwprod' Returns the options to setup the next feval call to compute the Jacobian Vector product. In this case, val is the matrix (or vector) you want to multiply J with.

ADfunc='revprod' Returns the options to setup the next feval call to compute the Matrix - Jacobian product(reverse mode). In this case, val is the matrix (or vector) you want to multiply J with.

ADfunc='htimesv' Returns the options to setup the next feval call to compute the Hessian Matrix product(reverse mode). In this case, val is the matrix (or vector) you want to multiply H with.

ADfunc='jacsp' Returns the options to setup the next feval call to compute the sparsity pattern of the Jacobian.

ADfunc='hesssp' Returns the options to setup the next feval call to compute the sparsity pattern of the Hessian.