ADMIT-1

Automatic Differentiation and MATLAB Interface Toolbox – The MATLAB AD tool version

Thomas F. Coleman and Arun Verma
Department of Computer Science
Cornell University
Ithaca, NY 14853

coleman@cs.cornell.edu **verma@cs.cornell.edu**

Contents

1 Introduction

ADMIT-1 enables you to compute *sparse* Jacobian and Hessian matrices, using automatic differentiation technology, from a MATLAB environment. You need only supply a M-function to be differentiated and ADMIT-1 will exploit sparsity if present to yield sparse derivative matrices (in sparse MATLAB form). ADMIT-1 also allows for the calculation of gradients and has several other related functions.

For information about downloading ADMIT please see the URL :
http://www.cs.cornell.edu/home/verma/AD/research.html.

2 An example

Here is simple example illustrating how to use ADMIT-1 to calculate the Jacobian of the function $y = F(x)$, $F : \Re^n \rightarrow \Re^n$ where

$$y(1) = 2x(1)^2 + \sum_1^n x(i)^2,$$
$$y(i) = x(i)^2 + x(1)^2, \quad i = 2 : n.$$

The Jacobian of function F has an arrowhead sparsity structure, as shown in Figure 1 for $n = 50$.

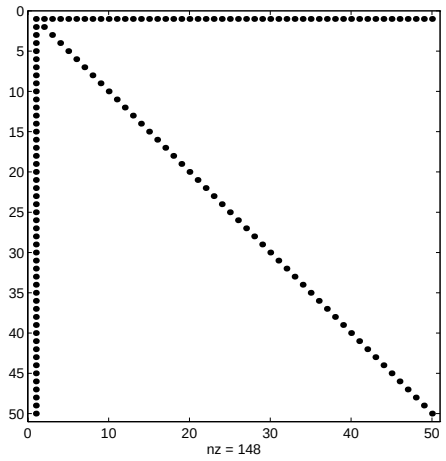


Figure 1: The sparsity structure of Jacobian J

```
function f= examplefun(x,m,Extra)
   f=x.*x;
   f(1)=f(1)+x'*x;
   f=f+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); n = 5;
>> JPI = getJPI('myfun',n);
>> [f,J]=evalJ('myfun',x,[],[],JPI);
>> f
f =

    7
    2
    2
    2
    2
>> spy(J) <- Sparsity structure is displayed.
```

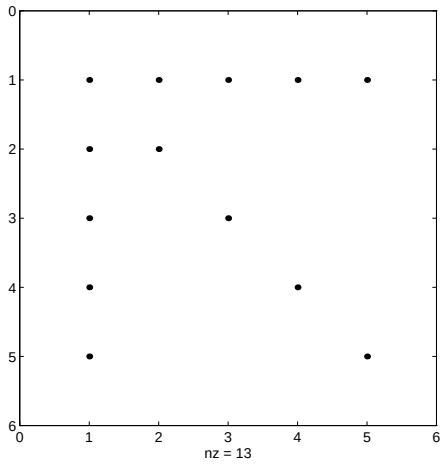


Figure 2: The sparsity structure of example computation

The second statement, involving "getJPI" extracts sparsity/coloring information. As illustrated below in the Newton iteration example, only one execution of "getJPI" is required for a given target function.

2.1 Newton Process in ADMIT

Typically, derivative matrices are used in an iterative procedure, requiring repeated evaluation at different arguments. The most important thing to remember in this situation is that sparsity information ("getJPI") need be obtained only once; therefore, these statements reside outside of the main loop. For example, below is a simple Newton iteration. The target function is contained in file broyd1a.c.

```
x = zeros(10,1); tol = 1e-5;
n = length(x);
JPI=getJPI('broyd1a',n); <- Compute sparsity pattern/coloring once, for all iterations
```

```
[f,J]=evalJ('broyd1a',x,[],[],JPI);
it=0;

% The Newton Iteration
while (norm(f) > tol) & (it < 100)

        delta= -J\f;
        x=x+delta;
        [f,J]=evalJ('broyd1a',x,[],[],JPI);
        it=it+1;
end
```

3 The ADMIT functions

evalJ

Purpose

Compute the value of a differentiable vector mapping f and its' Jacobian J. Function evalJ is designed for the case where J is a sparse matrix.

Synopsis

f=evalJ(fun,x)

f=evalJ(fun,x,Extra)

f=evalJ(fun,x,Extra,m)

[f,J]=evalJ(fun,x,Extra,m,JPI)

[f,J]=evalJ(fun,x,Extra,m,JPI,verb)

Description

f=evalJ(fun,x) Evaluate the function at the input argument x. The function is assumed to be a square mapping with dimension defined by the length of x.

f=evalJ(fun,x,Extra) You can provide a *full* matrix, Extra, to be used by your target function. Extra cannot be a MATLAB *sparse* matrix.

f=evalJ(fun,x,Extra,m) Scalar m is the row dimension of the vector mapping, i.e., $f : \Re^n \rightarrow \Re^m$.

[f,J]=evalJ(fun,x,Extra,m,JPI) Evaluate the sparse Jacobian J at the point x. JPI encodes the "coloring" information about the sparse matrix J. (See getJPI.) Different sparsity-exploiting methods are possible; the default sparse method is direct determination using bi-coloring [8].

[f,J]=evalJ(fun,x,Extra,m,JPI,verb) Indicates the display level.

verb ≤ 0 No display.

verb ≥ 1 The number of groups used are displayed.

verb ≥ 2 Information is displayed in graph form.

evalH

Purpose

Compute the value of a scalar-valued function, the gradient, and possibly the Hessian matrix. When the Hessian matrix is computed, sparsity is exploited (using graph-coloring techniques, etc. [1, 4])

Synopsis

v=evalH(fun,x)

v=evalH(fun,x,Extra)

[v,grad]=evalH(fun,x,Extra)

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

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

Description

v=evalH(fun,x) Determine the (scalar) value of fun at the input argument x.

v=evalH(fun,x,Extra) You can provide a *full* matrix, Extra, to be used by your target function. Extra cannot be a MATLAB *sparse* matrix.

[v,grad]=evalH(fun,x,Extra) The gradient (a dense vector) is evaluated at x.

[v,grad,H]=evalH(fun,x,Extra,HPI) Evaluate the sparse Hessian matrix H at x. HPI encodes the "coloring" information about H required to compute a compact representation of H. (See getHPI.) Different sparsity-exploiting methods are possible; the default sparse method used is direct determination (ignoring the symmetry).

[v,grad,H]=evalH(fun,x,Extra,HPI,verb) Indicates the display level.

verb ≤ 0 No display.

verb ≥ 1 The number of groups used are displayed.

verb ≥ 2 Information is displayed in graph upon termination.

getJPI

Purpose

Compute sparsity and coloring information to allow for the efficient determination of a (sparse) Jacobian matrix.

Synopsis

JPI= getJPI(fun, m)

JPI= getJPI(fun, m, n)

JPI= getJPI(fun, m, n,Extra)

JPI= getJPI(fun, m, n, Extra, method)

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

[JPI,SPJ]= getJPI(fun, m, ..)

Description

JPI= getJPI(fun, m) encapsulates (in a MATLAB sparse matrix) the sparsity pattern and graph coloring information necessary to efficiently compute the sparse Jacobian matrix, assumed to be m $\times$ m; the coloring determined corresponds to the default – direct bi-coloring.

JPI= getJPI(fun, m, n) The Jacobian matrix is assumed to be m $\times$ n. JPI= getJPI(fun, m, n, Extra) You can provide a *full* matrix, Extra, to be used by your target function fun .

JPI= getJPI(fun, m, n,Extra, method)

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.

JPI= getJPI([], m, n,Extra, method, SPJ) You can supply SPJ, a sparse MATLAB matrix representing the sparsity structure of the Jacobian matrix. This is helpful since the procedure to compute the sparsity via AD tool is quite expensive. The sparse matrix structure SPJ is required on input when method = 'f'.

[JPI,SPJ]= getJPI(fun, m, ..) The sparsity structure of the Jacobian is returned in SPJ.

getHPI

Purpose

Compute the sparsity structure and graph coloring information for the sparse Hessian matrix H.

Synopsis

HPI= getHPI(fun, n)

HPI= getHPI(fun, n, Extra)

HPI= getHPI(fun, n, Extra, method)

HPI= getHPI([], n, Extra, method, SPH)

[HPI,SPH]= getHPI(fun, n, ...)

Description

HPI= getHPI(fun, n) The sparsity structure and relevant coloring information (to allow for efficient calculation of the sparse Hessian H) is encapsulated in HPI, a sparse matrix. The default coloring corresponds to direct determination.

HPI= getHPI(fun, n, Extra) You can provide a *full* matrix, Extra, to be used by your target function (if required).

HPI= getHPI(fun, n,Extra,method)

method = 'i-a': The default, ignore the symmetry. Compute exactly using AD.

method = 'd-a': direct method [4], using AD.

method = 's-a': substitution method [4] using AD.

method = 'i-f': ignore the symmetry and use finite differences(FD)

method = 'd-f': direct method [4] with FD.

method = 's-f': substitution method [4] with FD.

HPI= getHPI([], n,Extra,method,SPH) You can supply SPH, a sparse MATLAB matrix representing the sparsity structure of the Hessian matrix. This can be helpful since the computation to compute SPH is quite expensive. The sparse structure SPH is required as input when method = 's-f'.

[HPI,SPH]= getHPI(fun, n,..) Returns the sparsity pattern of the Hessian matrix.

4 Design of User function : "fun"

Design your target M-function as follows.

```
function f= examplefun(x,m,Extra)

%crunch

end
```

The input argument x is a vector of dimension n; y is the output vector of dimension m. Extra is additional parameter for use in the function.

4.1 An example function

Here is a simple example, which leads to a arrow-head sparsity structure.

```
function f= examplefun(x,m,Extra)
  f=x.*x;
  f(1)=f(1)+x'*x;
  f=f+x(1)*x(1);
```

5 Display Reports

For both evalJ and evalH two different levels of display output are possible. You can choose the level of display using the input parameter verb.

- evalJ.
  - if verb ≤ 0 then there is no information displayed.
  - if verb ≥ 1 then the number of groups used for "coloring" are displayed. For example, the output produced by the direct bi-coloring Method looks like :

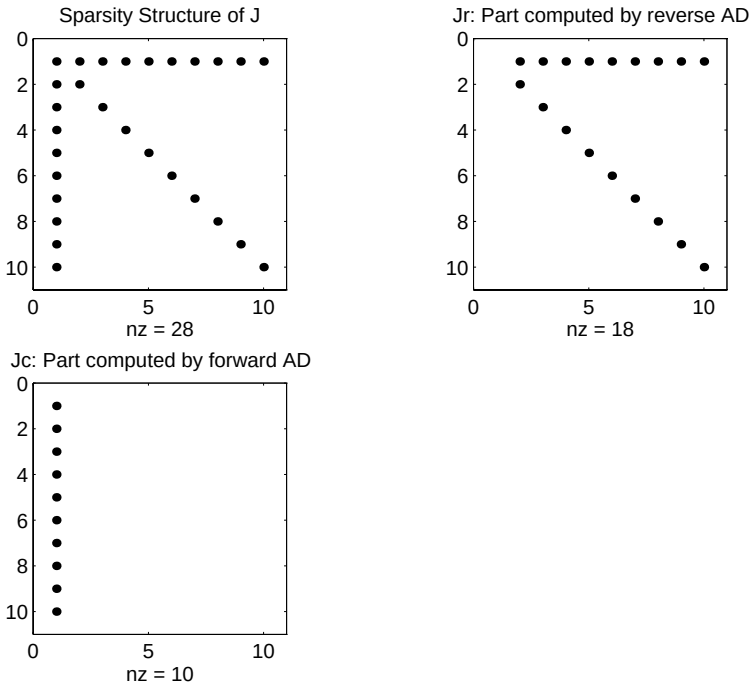
    Size of the Jacobian = 10x10

    Number of Row groups = 1

    Number of column groups = 2

    Total Number of groups = 3

  - if verb ≥ 2 then additional sparsity patterns are shown. For example, for the bi-coloring methods an array of 3 subplots is shown.



subplot(2,1,1) illustrates the sparsity structure of the Jacobian matrix.

subplot(2,1,2) graphs the sparsity structure of the the portion of the Jacobian computed by reverse mode AD.

subplot(2,1,3) illustrates the sparsity structure of the portion of the Jacobian computed by forward mode AD.

- evalH.
  - if verb ≤ 0 then there is no information displayed.
  - if verb ≥ 1 then the number of groups used for "coloring" are displayed. looks like :

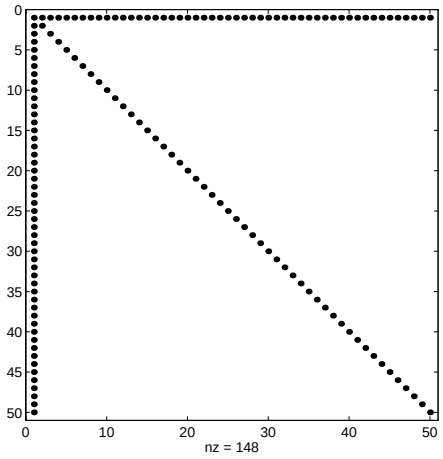
    Number of groups = 2

  - if verb ≥ 2 then the sparsity structure of the Hessian matrix is displayed.

6 Examples

Here is simple example illustrating the use of evalJ. The Jacobian is computed by three different methods: bi-coloring (direct), bi-coloring (substitution), and sparse finite differences. The target function, with an arrowhead Jacobian structure, is (examplefun.m):

```
function f= examplefun(x,m,Extra)
f=x.*x;
f(1)=f(1)+x'*x;
f=f+x(1)*x(1);
```



To execute the demo:

```
>> example('fun',verb);          <- Run demo ; use different verb values
```

6.1 Trying different coloring methods

ADMIT-1 allows for usage of different coloring method options, via the two functions, getJPI and getHPI. In the following illustration, we demonstrate how to use and switch between different methods.

```
>> m=100; n=100;
>> JPId = getJPI('fun',m,n);  <-  JPI for direct bi-coloring (default) method
>> JPIs = getJPI('fun',m,n,[],'s');  <-  JPI for substitution bi-coloring method
>> JPIc = getJPI('fun',m,n,[],'c');  <-  JPI for column coloring  method
```

In the above illustration the sparsity pattern of the Jacobian is computed three times. This is costly and it can be avoided:

```
>> ....
>> [JPId,SPJ] = getJPI('fun',m,n);  <-  JPI for direct bi-coloring (default) method
>> JPIs = getJPI([],m,n,[],'s',SPJ);  <-  JPI for substitution bi-coloring method
>> JPIc = getJPI([],m,n,[],'c',SPJ);  <-  JPI for column coloring  method
```

NOTE : please don't use 'jacexamp' for sizes =m=n < 50. It is made for larger dimensions.

Similarily for Hessians :

```
>> n=100;
>> [HPIi,SPH] = getHPI('examplesfun',n);  <-  HPI for ignore symmetry (default) method
>> HPId = getHPI([],n,[],'d-a',SPH);  <-  HPI for direct symmetry exploiting  method
>> HPIs = getHPI([],n,[],'s-a',SPH);  <-  HPI for substitution symmetry exploiting  method
```

7 The MATLAB AD tool

The motivation is to be able to build an AD tool for a high level language like MATLAB. Thinking about AD in terms of high-level matrix vector operations, helps, specially in terms of storage requirements in the reverse mode, where you have to just save the high level vectors. There are other insights gained by this high-level view, e.g. information about parallelization of derivative code.

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 MATLAB5's OOP (Object Oriented Programming) feature for implementation.

Usage of this tool is very simple – We include some easy examples along with the description of the tool in this writeup.

7.1 The @deriv class (Forward Mode of AD)

7.1.1 Example

- 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).

7.1.2 Description of the @deriv 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, value and derivative. 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.

Now 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.

Chain rule propagation rules :

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

An Example overload method 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)
global globp;
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.

7.2 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.

7.2.1 example

- 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.

7.2.2 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.

Rules for propagation of adjoints :

- $a = x^T y$

  $x_d^* += z_d^* * y.$

  $y_d^* += z_d^* * x.$

- $z = x + y$

  $x_d^* += z_d^*.$

  $y_d^* += z_d^*.$

- $z = x. * y$

  $x_d^* += z_d^* * diag(y).$

  $y_d^* += z_d^* * diag(x).$

- $y = Ax$

  $x_d^* += A^T * z_d.$

  $A_d(:, j, :)^* += x(j). * z_d^*.$

- $y = A \backslash x$

  $x_d^* += A^T \backslash y_d^T.$

  $A_d(:, :, i)^* += -(A^T \backslash y_d^{*T})^T y_d^*(:, i).$(outer product)

- $C = A + B$

  $A_d^* += C_d^*.$

  $B_d^* += C_d^*.$

- $C = A * B$

  $A_d(:, :, i)^* == B^T * C_d^*(:, :, i).$

  $B_d(:, :, i)^* == A^T * C_d^*(:, :, i).$

- $C = A. * B$

  $A_d(:, :, i)^* += C_d(:, :, i)^*. * B.$

  $B_d(:, :, i)^* += C_d(:, :, i)^*. * A.$

- $C = A./B$

  $A_d(:, :, i)^* += C_d(:, :, i)^*./B.$

  $B_d(:, :, i)^* += -C_d(:, :, i)^*. * A./B^2.$

7.3 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.

7.3.1 Example

- 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.

7.4 Computing Jacobian sparsity pattern

7.4.1 Example

- 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).

7.4.2 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.

Below, the $S_1 + S_2$ operation on sparsity pattern denotes, taking the superset of sparsity pattern of S_1 and S_2, equivalent to matlab command on sparse matrices : $S = spones(S_1 + S_2)$. Similarly for multiplication operation like $S = spones(S_1 * S_2)$.

The rules for propgation Jacobian sparsity pattern are similar to those of taylor series coefficients and are mentioned here.

Rules for prapagating Jacobian sparsity pattern

- $a = x^T y$

  $S_z = \sum S_{x(i)} + S_{y(i)}$.

- $z = x + y$
  $S_z = S_x + S_y$.
- $z = x.*y$
  $S_z = S_x + S_y$.
- $y = Ax$
  $S_y(i) = \sum S_{A(i,:))} * x + A(i,:) * S_x$.
- $y = A\backslash x$
  $S_y = A\backslash S_x - S_A * x$.
- $C = A + B$
  $S_C = S_A + S_B$.
- $C = A * B$
  $S_C = S_A + S_B$.
- $C = A.*B$ $S_C = S_A + S_B$.

7.5 Computing Hessian sparsity pattern

7.5.1 example

7.5.2 Example

- 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).

7.5.3 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.

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 .

Rules for prapagating Hessian sparsity pattern

- $a = x^T y$
  $\dot{z} = \dot{x}^T y + \dot{y}^T x$.
- $z = x + y$
  $H_z = H_x + H_y$.

- $z = x.*y$

  $H_z = H_x + H_y + J_x J_y^T + J_y J_x^T$.

- $C = A + B$

  $H_z = H_x + H_y$.

- $C = A.*B$

  $H_z = H_x + H_y + J_x J_y^T + J_y J_x^T$.

7.6 Problems

This AD tool can AD only the MATLAB code included in the M-file, and can't differentiate external code, e.g. MEX files.

- **MEX compiled files in MATLAB toolbox** . 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.

8 The AD Drivers

Additional functions for driving AD tool are described in this section. These drivers are all in form of MEX files.

forwprod

Purpose:

Computes the Jacobian matrix product, $J \times V$, where J is the Jacobian of a nonlinear vector mapping and V is a matrix. The product is computed directly via automatic differentiation – the cost is proportional to the number of columns in V. Note: forwprod is particularly efficient when the number of columns of V is small. Otherwise, when J is sparse it may be more efficient to compute J first (using evalJ and exploiting sparsity) and then perform the multiplication.

Synopsis

[f,JV]=forwprod(fun,x,V)

[f,JV]=forwprod(fun,x,V,m)

[f,JV]=forwprod(fun,x,V,m,Extra)

Description

[f,JV]=forwprod(fun,x,V) returns the function value and the product JV = $J*$V, evaluated at x. The row dimension of the Jacobian matrix is assumed to be equal to the column dimension, i.e., length(x).

[f,JV]=forwprod(fun,x,V,m) The row dimension of the Jacobian matrix is m.

[f,JV]=forwprod(fun,x,V,m,Extra) You can provide a full matrix, Extra, for use in your target function "fun" (if required).

revprod

Purpose:

Compute $W^T \times J$ where $J = J(x)$ is the Jacobian matrix of a nonlinear vector mapping and W is an arbitrary (consistent) matrix. The product is computed directly via automatic differentiation with the computational cost proportional to the number of columns in W. Note: revprod is particularly efficient when the number of columns of W is small. Otherwise, when J is sparse it may be more efficient to compute J first (using evalJ and exploiting sparsity) and then perform the multiplication.

Synopsis

[f,WJ]=revprod(fun,x,W);

[f,WJ]=revprod(fun,x,W,Extra);

Description

[f,WJ]=revprod(fun,x,W) returns the function value and the product WJ $= (W^T * J)^T = J^T W$.

[f,WJ]=revprod(fun,x,W,Extra) You can provide a full matrix, Extra, to be used by your target function "fun" (if required).

HtimesV

Purpose:

Compute $H \times V$ where $H = H(x)$ is a Hessian matrix of a scalar-valued function and V is a compatible matrix. Note: Function HtimesV is particularly efficient when the number of columns of V is small. Otherwise, when H is sparse it may be more efficient to compute H first (using evalH and exploiting sparsity) and then perform the multiplication.

Synopsis

HV=HtimesV(fun,x,V)

HV=HtimesV(fun,x,V,Extra)

Description

HV=HtimesV(fun,x,V) returns the product HV $= H * V$, where the Hessian matrix H is evaluated at the given point x.

HV=HtimesV(fun,x,V,Extra) You can provide a full matrix, Extra, to be used (if required) by your target function "fun".

funcval

Purpose:

Computes the value and the gradient of a scalar function.

Synopsis

val=funcval(fun,x)

val=funcval(fun,x,Extra)

[val,grad]=funcval(fun,x)

[val,grad]=funcval(fun,x,Extra)

Description

val=funcval(fun,x) Returns the scalar value of the function at the given point x.

val=funcval(fun,x,Extra) You can provide a full matrix, Extra, to be used by your target function (if required).

[val,grad]=funcval(fun,x, ...) Returns the gradient vector grad.

funcvalJ

Purpose:

Compute the value of a vector valued function.

Synopsis

val=funcvalJ(fun,x)

val=funcvalJ(fun,x,m)

val=funcvalJ(fun,x,m,Extra)

Description

val=funcvalJ(fun,x) Returns the value of the function at the given point x. The function is assumed to be a square mapping, $\Re^n \rightarrow \Re^n$, where $n = length(\mathsf{x})$.

val=funcvalJ(fun,x,m) The target function maps n-vectors to m-vectors.

val=funcvalJ(fun,x,m,Extra) You can provide a full matrix, Extra, to be used by your target function (if required).

hesssp

Purpose:

Computes the sparsity pattern of the Hessian matrix.

Synopsis

SPH=hesssp(fun,n)

SPH=hesssp(fun,n,Extra)

Description

SPH=hesssp(fun,n) Returns the n $\times$ n sparsity structure of the Hessian matrix. SPH is a MATLAB sparse matrix. Note the current point is not required: a superstructure of the sparsity structure for all points x is returned. The structure can be displayed by spy(SPH).

SPH=hesssp(fun,n,Extra) You can provide a full matrix, Extra, to be used by your target function "fun ", if required.

jacsp

Purpose:

Compute the sparsity pattern of the Jacobian matrix.

Synopsis

SPJ=jacsp(fun,m)

SPJ=jacsp(fun,m,n)

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

Description

SPJ=jacsp(m) Returns the m $\times$ m sparsity structure of the Jacobian matrix. SPJ is a MATLAB sparse matrix. The structure can be displayed by spy(SPJ).

SPJ=jacsp(fun,m,n) Returns the m $\times$ n sparsity structure of the Jacobian matrix.

SPJ=jacsp(fun,m,n,Extra) You can provide a full matrix, Extra, to be used by your target function "fun".

tenssp

Purpose:

Compute the sparsity pattern of the Hessian matrix of the function $w^T F$ for an arbitrary vector w and a nonlinear vector mapping $F = F(x)$.

Synopsis

SP=tenssp(fun,n)

SP=tenssp(fun,n,m)

SP=tenssp(fun,n,m,Extra)

Description

SP=tenssp(fun,n) Returns the n $\times$ n sparsity structure of the the Hessian matrix of the function $w^T F(x)$ for a general multiplier vector w and a nonlinear vector mapping F. The mapping is assumed to be from n-vectors to m-vectors. SP is a MATLAB sparse matrix.

The first input argument, fun, is integer handle identifying the function.

SP=tenssp(fun,n,m) F maps n-vectors to m-vectors.

SP=TesSP(fun,n,m,Extra) You can provide a full matrix, Extra, to be used by your target function (if required).

tensprod

Purpose:

Computes $H \times V$ where H is the Hessian of a function of the form $w^T F(x)$; F is a vector-valued function; w is a compatible vector; V is a compatible matrix.

Synopsis

TV=tensprod(fun,x,V,w)

TV=tensprod(fun,x,V,w,Extra)

Description

TV=tensprod(fun,x,V,w) returns the product $H \times$ V where H is the hessian of the function the function $w^T F$, evaluated at x.

TV=tensprod(fun,x,V,w,Extra) You can provide a full matrix, Extra, to be used by your (target) function "fun ".

9 References

Here are some useful related references:

- Automatic differentiation [10, 9]
- Single-sided determination of sparse Jacobians [5, 2]
- Determination of sparse Hessian matrices [1, 4, 3]
- Bi-coloring [8].
- Structured Jacobians, Hessians [7, 6]
- ADOL-C [11]

References

[1] T. F. Coleman and J.-Y. Cai, *The cyclic coloring problem and estimation of sparse Hessian matrices*, SIAM J. Alg. Disc. Meth., 7 (1986), pp. 221–235.

[2] T. F. Coleman, B. S. Garbow, and J. J. Moré, *Software for estimating sparse Jacobian matrices*, ACM Trans. Math. Software, 10 (1984), pp. 329–345.

[3] ——, *Software for estimating sparse Hessian matrices*, ACM Trans. Math. Software, 11 (1985), pp. 363–377.

[4] T. F. Coleman and J. J. Moré, *Estimation of sparse Hessian matrices and graph coloring problems*, Math. Programming, 28 (1984), pp. 243–270.

[5] ——, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. on Numerical Analysis, 20 (1984), pp. 187–209.

[6] T. F. Coleman and A. Verma, *Structure and efficient Hessian calculation*, Tech. Report TR96–258, Cornell Theory Center, Cornell University, September 1996.

[7] ——, *Structure and efficient Jacobian calculation*, in Computational Differentiation: Techniques, Applications, and Tools, M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds., SIAM, Philadelphia, Penn., 1996, pp. 149–159.

[8] ——, *The efficient computation of sparse Jacobian matrices using automatic differentiation*, SISC, (1997 (to appear)).

[9] A. Griewank, *Direct calculation of Newton steps without accumulating Jacobians*, in Large-Scale Numerical Optimization, T. F. Coleman and Y. Li, eds., SIAM, Philadelphia, Penn., 1990, pp. 115–137. Also appeared as Preprint MCS–P132–0290, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., February 1990.

[10] ——, *Some Bounds on the Complexity of Gradients, Jacobians, and Hessians*, in Complexity in Nonlinear Optimization, P. Pardalos, ed., World Scientific Publishers, 1993, pp. 128–161.

[11] 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.

Appendix

A Example/Demo of Drivers

To execute a demo illustrating the use of different ADOL-C drivers, try:

```
>> [f1,f2]=configureexADC;
>> exampleAD(f1,f2);
```

B An example Newton process

B.1 Newton Process for nonlinear equations F(x)=0

We illustrate the use of ADMIT in a Newton process. The example target function is broyd1a.m. Here is the shell (MATLAB) program:

```
>> fun = 'broyd1a';
>> itbnd=100;
>> tol= 1e-6;
>> xstart=[zeros(50,1);0.2*ones(50,1)];
>>
>> get the Coloring Info Once and for all
>> JPI= getJPI(fun,100);
>>
>> [x,it,norm] = newton(fun,xstart,tol,itbnd,JPI);
>> cleanup
>> exit
```

Below we list our M-file containing the Newton procedure.

```
function [x,it,nf]= newton(fun, xstart, tol, itbnd, JPI)

% Initializations
n=length(xstart);
if (nargin < 3) tol=1e-5; end
if (nargin < 4) itbnd=60; end
if (nargin < 5)
% Get the Coloring Info Once and for all
        JPI= getJPI(fun,n);
end
n=length(xstart);
x=xstart;

% First Evaluation
[f,J]=evalJ(fun,x,[],[],[],JPI);
it=0;
```

```
% The Newton Iteration
while ((norm(f) > tol) & (it < itbnd))

        delta= -J\f;
        x=x+delta;
        [f,J]=evalJ(fun,x,[],[],[],JPI);
        it=it+1;
end

nf=norm(f);
```

B.2 An example Newton process for optimization

The function newtonH codes the newton process solves the minimization problem, usinf iterative solution $Hs = -g$. The example target function is brown1.m. Here is the shell (MATLAB) program:

```
>> fun = 'brown1';
>> itbnd=100;
>> tol= 1e-6;
>> xstart=[zeros(50,1);0.2*ones(50,1)];
>>
>> get the Coloring Info Once and for all
>> HPI= getHPI(fun,100);
>>
>> [x,it,norm] = newtonH(fun,xstart,tol,itbnd,HPI);
>> cleanup
>> exit
```

C Newton step via conjugate gradient for nonlinear least squares

It is easy to write a conjugate gradient solver for NLS using ADMIT. In this section, we describe the two conjugate gradient solvers in nonlinear least squares setting

$$\min \quad \|F(x)\|_2^2$$

Typically a Newton iteration is applied to solve above problem, and two different kinds of Newton steps are popular :

- **Gauss Newton step**
$$J(x)^T J(x)s = -J(x)^T F(x) \tag{1}$$
- **Complete Newton step**
$$H(x)s = -\nabla f(x) \tag{2}$$
where $f(x) := \|F(x)\|_2^2$.

IN ADMIT-1, we have conjugate gradient solvers, congrd and congrdH respectively for the Gauss-Newton step and the complete Newton step.

congrd

Purpose:

Computes the Gauss Newton step $s = -(J(x)^T J(x))^{-1} J(x)^T F(x)$ via conjugate gradient method.

Synopsis

s=congrd(fun,x)

s=congrd(fun,x,m,tol)

s=congrd(fun,x,m,tol)

s=congrd(fun,x,m,tol,P)

s=congrd(fun,x,m,tol,P,J)

s=congrd(fun,x,m,tol,P,J,JPI)

[s,it]=congrd(fun,x,...)

[s,it,normres]=congrd(fun,x,...)

Description

s=congrd(fun,x) returns the step $s = -(J^T J)^{-1} J^T F$. The function F, denoted by the identifier fun, is assumed to be a square mapping. Default tolerance and preconditioner ($\equiv I$) are used.

s=congrd(fun,x,m) fun is allowed to be a nonsquare mapping, i.e from $\Re^n \rightarrow \Re^m$. Default tolerance and preconditioner ($\equiv I$) are used.

s=congrd(fun,x,m,tol The user supplied tolerance on the residual norm is used.

s=congrd(fun,x,m,tol,P) The user supplied preconditioner is used.

s=congrd(fun,x,m,tol,P,J) User can supply th Jacoboian J(x).

s=congrd(fun,x,m,tol,P,J,JPI) User can supply precomputed JPI, the partition and coloring information structure for efficiency.

[s,it] =congrd(fun,x,...) Number of iterations required are also returned.

[s,it,normres] =congrd(fun,x,...) normres which contains the norm of the residual is also returned.

congrdH

Purpose:

Computes the complete Newton step $s = -H(x)^{-1} \nabla f(x)$ via conjugate gradients.

Synopsis

s=congrdH(fun,x)

s=congrdH(fun,x,tol)

s=congrdH(fun,x,tol,P)

s=congrdH(fun,x,tol,P,H)

s=congrdH(fun,x,tol,P,H,HPI)

[s,it]=congrdH(fun,x,...)

[s,it,normres]=congrdH(fun,x,...)

Description

s=congrdH(fun,x) returns the step $s = -H^{-1}\nabla f$. The scalar function f is denoted by the identifier fun. Default tolerance and preconditioner ($\equiv I$) are used.

s=congrdH(fun,x,tol The user supplied tolerance on the residual norm is used.

s=congrdH(fun,x,tol,P) The user supplied preconditioner is used.

s=congrdH(fun,x,tol,P,H) User can supply precomputed H(x).

s=congrdH(fun,x,tol,P,H,HPI) User can supply precomputed HPI, the partition and coloring information structure for efficiency.

[s,it] =congrdH(fun,x,...) Number of iterations required are also returned.

[s,it,normres] =congrdH(fun,x,...) norm which contains the norm of the residual is also returned.

C.1 Using conjugate gradient solvers

In ADMIT distribution, we have two example functions cgfun.c which codes the function $F(x)$, and cgsumsqr.c which codes the function $f(x) := \|F(x)\|_2^2$.

C.1.1 Using congrd

```
>> fun = 'cgfun';
>> tol= 1e-6;
>> x=ones(100,1);
>>
>> get the Coloring Info Once and for all
>> JPI= getJPI(fun,100);
>>
>> P is the preconditioner
>> P= ...
>>
>> [s,it,normres] = congrd(fun,x,100,tol,P,JPI);
```

Below we list our M-file which codes congrd.

```
function [s,it,normres]=congrd(fun,x,m,tol,P,JPI)

% Initializations
n=length(x);
if nargin < 3 m =[]; end
if nargin < 4 tol =[]; end
if nargin < 5 P =[]; end
if nargin < 6 JPI =[]; end

if m==[] m =n; end
if tol==[] tol =1e-6; end
if P==[]  P =speye(n); end
if JPI==[] JPI =getJPI(fun,m,n); end

[f,J]=evalJ(fun,x,[],m,JPI);
k=0;
s=zeros(n,1);
r=-J'*f;
normb=norm(r);
rho=normb;

% Main loop
while (rho > tol*normb)
        z=P\r;
        k=k+1;
        if k==1
                p=z;
        else
                beta=r'*z/(rprev'*zprev);
                p=z+beta.*p;
        end
        temp=J*p;
        alpha=r'*z/(temp'*temp);
        s=s+alpha.*p;
        zprev=z;
        rprev=r;
        r=r-alpha.*J'*temp;
        rho=norm(r);
end
it=k;
normres=rho;
```

C.1.2 Using congrdH

```
 >> fun = 'cgsumsqr';
 >> tol= 1e-6;
 >> x=ones(100,1);
 >>
 >> get the Coloring Info Once and for all
 >> HPI= getHPI(fun,100);
 >>
 >> P2 is the preconditioner
 >> P2= ...
 >>
 >> [s,it,normres] = congrdH(fun,x,tol,P2,HPI);
```

Below we list our M-file which codes congrdH.

```
function [s,it,normres]=congrd(fun,x,tol,P,HPI)

n=length(x);
if nargin < 3 tol =[]; end
if nargin < 4 P =[]; end
if nargin < 5 HPI =[]; end

if tol==[] tol =1e-6; end
```

```
if P==[]  P =speye(n); end
if HPI==[] HPI =getHPI(fun,n); end

[f,g,H]=evalH(fun,x,[],HPI);
k=0;
s=zeros(n,1);
r=-g;
normb=norm(r);
rho=normb;

while (rho > tol*normb)
        z=P\r;
        k=k+1;
        if k==1
                p=z;
        else
                beta=r'*z/(rprev'*zprev);
                p=z+beta.*p;
        end
        temp=H*p;
        alpha=r'*z/(p'*temp);
        s=s+alpha.*p;
        zprev=z;
        rprev=r;
        r=r-alpha.*temp;
        rho=norm(r);
end
it=k;
normres=rho;
```

C.2 Other CG solvers

The ADMIT package contains alternative solvers for both the Gauss Newton step and the complete Newton step, congrd2 and congrdH2. Their usage is exactly the same as congrd and congrdH respectively, except they employ the AD-drivers (see the appendix) for directly computing the products $J^T Jv$ and Hv with out first computing J and H respectively. Functions congrd2 and congrdH2 do not require JPI and HPI as arguments, respectively.

D The feval interface

Examples of overloaded 'feval' interface

This section provides two illustrated examples of the usage of the overload feval interface – once each of a vector mapping and a scalar mapping. The functions used are examplefun.m and examplesfun.m – to view the example functions and demo, go to /home/verma/ADMAT, run matlab and execute the commands given below. ("type examplefun" and "type examplesfun" will let you see what the functions look like).

- **Vector Mapping :**

```
>> myfun=fun('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)
```

- **Vector Mapping :**

```
>> mysfun=fun('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).

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

Description

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

@fun

Purpose

Synopsis

derivfun=fun(funstr)

derivfun=fun(funstr,scalar)

Description

derivfun=fun(funstr)
e.g myfun=fun('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.

Description

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.