A Matlab Implementation of the Minpack-2 Test Problem Collection

Dr Shaun Forth
Centre for Simulation and Analytics
S.A.Forth@cranfield.ac.uk

EuroAD Workshop, Paderborn Germany, 30 November 2015
www.cranfield.ac.uk
Introduction

1. AD Packages in Matlab
2. The Minpack-2 Test Problem Collection
Introduction

- **Grad** [Rich and Hill, 1992] - forward mode AD on a Matlab string via Turbo-C

- **ADMAT** [Verma, 1999] - forward and reverse (via tape) mode AD using OO features of Matlab (+ second order derivatives and sparsity detection)

- **ADiMat** [Bischof et al., 2003] - source transformed forward/reverse mode

- **MAD** [Forth, 2006] - optimised derivatives storage class `derivvec` to give improved overloaded forward mode performance over ADMAT

- **MSAD** [Kharche and Forth, 2006] - source-transformation by specialising and inlining MAD’s derivative objects.

- **ADiGator** [Patterson et al., 2013, Weinstein and Rao, 2015] - source-transformed, sparsity-exploiting forward mode AD (vertex elimination with forward ordering [Griewank and Reese, 1991]).

Need for testcases.
The Minpack-2 Test Problem Collection [Averick et al., 1991]

- Describes 24 optimisation problems, including Fortran 77 source code, of three problem types:
  - unconstrained minimisation - objective function, gradient, Hessian
  - least squares minimisation - residual, Jacobian
  - systems of nonlinear equations - residual, Jacobian
- For large-scale problems source code for Hessian/Jacobian sparsity pattern and Hessian/Jacobian-vector products also provided.
- Appears 82 times in Scopus
- Widely used for AD tool validation, eg, Bischof et al. [1996], Walther and Griewank [2004], Naumann and Utke [2005], Giering and Kaminski [2006], Shin and Hovland [2007], Forth et al. [2004].
Matlab Implementation of Minpack-2 Problems

Converting Minpack-2 to Matlab
Re-coding Minpack-2 in Matlab
Lenton [2005] hand-converted all the Minpack problems to Matlab
  ▶ Changes of syntax, array constructors
  ▶ Arrays must have lower index 1 in Matlab (also affects loop indices)

Validated by
  ▶ Fortran program creates random vectors \( \mathbf{x} \), calls Fortran Minpack, writes \( f(\mathbf{x}), \nabla f, \ldots \) to formatted text file.
  ▶ Text file read into Matlab and results compared to those from Matlab version of Minpack calls
  ▶ Results agree to within i/o and floating point round-off
Re-coding Minpack-2 in Matlab

- Lenton’s conversion satisfactory for small-scale, fixed $n$ problems.
- For large scale problems Lenton’s Fortran-based coding uses loops and subscripting operations.
- Large number of overloaded function calls in overloaded AD.
- Re-coded Minpack-2 problems using array operations to give a vectorised version.
- Uniform interface to all functions with problem specific parameters supplied in a structure.
Separate functions for:
  - Setting problem parameters - constants, standard start vector, etc
  - Function evaluation only + vectorised version where appropriate.
  - Function + Gradient/Jacobian
  - Gradient/Jacobian only
  - Jacobian/Hessian-vector product
  - Sparsity pattern.

Use Matlab’s **Code Analyzer** to eliminate: dead code, unused variables.

Hand-coded adjoint for gradients.
Examples

- Human Heart Dipole Problem (HHD)
- Minimal Surface Area (MSA) Problem
Human Heart Dipole Problem (HHD)

Calculating dipole moment of human heart, find \( \mathbf{x} \in IR^8 \) such that 
\[
\mathbf{F}(\mathbf{x}) = 0
\]
with,
\[
\mathbf{F}(\mathbf{x}) = \begin{bmatrix}
    x_1 + x_2 - \sigma_{mx} \\
    x_3 + x_4 - \sigma_{my} \\
    x_5 x_1 + x_6 x_2 - x_7 x_3 - x_8 x_4 - \sigma_A \\
    x_7 x_1 + x_8 x_2 + x_5 x_3 + x_6 x_4 - \sigma_B \\
    x_1 (x_5^2 - x_7^2) - 2x_3 x_5 x_7 + x_2 (x_6^2 - x_8^2) - 2x_4 u_6 u_8 - \sigma_C \\
    x_3 (x_5^2 - x_7^2) + 2x_1 x_5 x_7 + x_4 (x_6^2 - x_8^2) + 2x_2 u_6 u_8 - \sigma_D \\
    x_1 (x_5 x_6^2 - 3v_5^2) + x_3 x_7 (x_7^2 - 3v_5^2) \ldots \\
    \ldots + x_5 x_6 (x_6^2 - 3x_8^2) + x_4 x_8 (x_8^2 - 3x_6^2) - \sigma_E \\
    x_3 (x_5 x_6^2 - 3v_5^2) - x_1 x_7 (x_7^2 - 3v_5^2) + x_4 x_6 (x_6^2 - 3x_8^2) \ldots \\
    \ldots - x_2 x_8 (x_8^2 - 3x_6^2) - \sigma_F
\end{bmatrix}
\]
for given constant \( \sigma_{mx}, \sigma_{my}, \sigma_A, \ldots, \sigma_F \).
subroutine dhhdfj(n,x,fvec,fjac,ldfjac,task,prob)

  c set constants
  if (prob .eq. 'DHHD1') then
    summx = 0.485d0
  :
  else if (prob .eq. 'DHHD2') then

  c standard start point
  if (task .eq. 'XS') then
    if (prob .eq. 'DHHD1') then
      x(1) = 0.299d0
  c intermediate variables
  a = x(1)

  c function
  if (task .eq. 'F' .or. task .eq. 'FJ') then
    fvec(1) = a + b - summx

  c Jacobian
  if (task .eq. 'J' .or. task .eq. 'FJ') then
    fjac(1,1) = one
function varargout=dhhdfj(x,task,prob)
if (prob == 'DHHD1')
    summx = 0.485d0; ...
elseif (prob == 'DHHD2')
    % Compute the standard starting point if task = 'XS'.
    switch task
    case 'XS'
        if (prob == 'DHHD1')
            x = [0.299d0; 0.186d0; -0.0273d0; 0.0254d0; ...
            varargout{1}=x;
        % function and/or Jacobian value
    case {'F','J','FJ'}
        if (task == 'F' | task == 'FJ')
            fvec=[a + b - summx;... % array constructor
                varargout{1}=fvec; % task dependant returned values
        if (task == 'J' | task == 'FJ')
            fjac= [ 1 1 0 0 0 0 0 0; ... % array constructor
                varargout{nout}=fjac; % task dependant returned values
Structure `Prob` used to store all problem data

```matlab
function Prob=MinpackHHD_Prob(vers)
Prob.user.vers = vers;
% set problem version dependent constants and x0
switch vers
  case 1
    Prob.user.summx = 0.485d0;
    Prob.user.summy = -0.0019d0;
    ...
  case 2
    ...
end
```
For AD this is the code we would expect the user to supply.

```matlab
function F=MinpackHHD_F(x,Prob)
    % unpack constants
    summx=Prob.user.summx ;
    summy=Prob.user.summy ;
    :
    % intermediate variables
    a = x(1);
    b = x(2);
    :
    % Evaluate the function
    F=[a + b - summx;
       c + d - summy;
    ];
    :
```

This will be the source code for our AD tests.
For many optimisation packages this is the ideal function definition. Jacobian only computed if required.

function [F,J]=MinpackHHD_FJ(x,Prob)
% unpack constants
summx=Prob.user.summx ;
summy=Prob.user.summy ;
:
% intermediate variables
a = x(1);
b = x(2);
F=[a + b - summx;
   c + d - summy;]
:
% Jacobian if required
if nargout==2
   if nargin==2
      J= [ 1 1 0 0 0 0 0 0 ;
          0 0 1 1 0 0 0 0 ;
          0 0 1 1 0 0 0 0 ;
          0 0 1 1 0 0 0 0 ;
          0 0 1 1 0 0 0 0 ;
          0 0 1 1 0 0 0 0 ;
          0 0 1 1 0 0 0 0 ;
          0 0 1 1 0 0 0 0 ;
          0 0 1 1 0 0 0 0 ];
   
   for i=1:4
      J(i,:)=[1 0 0 0 0 0 0 0 ;
              0 1 0 0 0 0 0 0 ;
              0 0 1 0 0 0 0 0 ;
              0 0 0 1 0 0 0 0 ];
   end

end
Some packages require Jacobian computed separately from function

Remove unused variables

function J=MinpackHHD_J(x,Prob)
% constants - not required
% intermediate variables
a = x(1);
b = x(2);
:
% Jacobian
J= [ 1 1 0 0 0 0 0 0 ;
    0 0 1 1 0 0 0 0 ;
    : ];
Minimal Surface Area (MSA) Problem

- Supply height on $x$ and $y$ boundaries of $[-\frac{1}{2}, \frac{1}{2}] \times [-\frac{1}{2}, \frac{1}{2}]$
Supply height on $x$ and $y$ boundaries of $\left[-\frac{1}{2}, \frac{1}{2}\right] \times \left[-\frac{1}{2}, \frac{1}{2}\right]$

Determine surface $u(x, y)$ of minimal surface area with given boundaries.
Minimal Surface Area (MSA) Problem

- Supply height on $x$ and $y$ boundaries of $\left[-\frac{1}{2}, \frac{1}{2}\right] \times \left[-\frac{1}{2}, \frac{1}{2}\right]$
- Determine surface $u(x, y)$ of minimal surface area with given boundaries.
- Known analytic solution due to Enneper.
Known boundary values

Unknown values

\[ T_{L,i,j}, T_{U,i,j} \]

Piecewise Linear function on

\[ \min f(u) = \sum_{i=1}^{n_i+1} \sum_{j=1}^{n_j+1} (f_{L,i,j} + f_{U,i,j}) \]
Known boundary values

\( \sum_{i=1}^{n_i+1} \sum_{j=1}^{n_j+1} (f_{L,i,j} + f_{U,i,j}) \)
Known boundary values

Unknown values
Known boundary values
Unknown values
Piecewise Linear function on $T_{i,j}$
$$\begin{align*}
\text{Min } f(u) &= \\
&= \sum_{i=1}^{n_i+1} \sum_{j=1}^{n_j+1} \left( t_{i,j}^L + t_{i,j}^U \right)
\end{align*}$$
Minimise,

\[ f(u) = \sum_{i=1}^{n_i+1} \sum_{j=1}^{n_j+1} \left( f_{i,j}^L + f_{i,j}^U \right), \]

with \( f_{i,j}^L, f_{i,j}^U \) the surface area on the lower/upper triangles:

\[
\begin{align*}
f_{i,j}^L &= \frac{h_x h_y}{2} \left\{ 1 + \left( \frac{u_{i+1,j} - u_{i,j}}{h_x} \right)^2 + \left( \frac{u_{i,j+1} - u_{i,j}}{h_y} \right)^2 \right\}^{\frac{1}{2}} \\
f_{i,j}^U &= \frac{h_x h_y}{2} \left\{ 1 + \left( \frac{u_{i+1,j+1} - u_{i,j}}{h_x} \right)^2 + \left( \frac{u_{i+1,j+1} - u_{i+1,j}}{h_y} \right)^2 \right\}^{\frac{1}{2}}
\end{align*}
\]

and we only consider \( u_{i,j} \) with \( i = 2, \ldots, n_i + 1, j = 2, \ldots, n_j + 1 \).
subroutine dmsafg(nx, ny, x, f, fgrad, task, bottom, top, left, right)

:  
     c  function and gradient over the lower triangular elements.
     do 50 j = 0, ny
     do 40 i = 0, nx
         k = nx*(j-1) + i ! 1-D indexing
         if (i .ge. 1 .and. j .ge. 1) then
             v = x(k) ! first vertex in triangle
         else
             if (j .eq. 0) v = bottom(i+1)
         :  
         :  
         :  
         if (i .lt. nx .and. j .gt. 0) then
             vr = x(k+1) ! right vertex
         :  
         :  
         dvdx = (vr-v)/hx
         dvdy = (vt-v)/hy
         fl = sqrt(one+dvdx**2+dvdy**2)
         if (feval) f = f + fl
         if (geval) then
function varargout=dmsafg(nx,ny,x,task,bottom,top,left,right)
:
switch task
  case {'F','G','FG'}
    for j = 0:ny
      for i = 0:nx
        k = nx*(j-1) + i; % 1-D indexing
        if i >= 1 && j >= 1
          v = x(k); % first vertex in triangle
        else if j == 0
          v = bottom(i+1);
        end if j == 0
        else if j >= 0
          v = bottom(i+1);
        end
        :
        if i < nx && j > 0
          vr = x(k+1); % right vertex
        end
        dvdx = (vr-v)/hx;
        dvdy = (vt-v)/hy;
        fl = sqrt(1+dvdx^2+dvdy^2);
        if geval
          22/ 44
MSA - Re-Coding the Problem Definition

Structure Prob used to store all problem data

```matlab
function [Prob,nuse]=MinpackMSA_Prob(varargin)
% [Prob,nuse]=MinpackMSA_Prob(nx,ny,bottom,top,left,right)
% check/set boundary conditions
if isempty(bottom)
    bottom=Enneper('bottom',nx,ny);
elseif ~(isvector(bottom) && length(bottom)==nx+2)
    error(['MinPackMSA_Prob: bottom must be a length nx+2 = ',...
end :
% Compute the standard starting point
x_0 = reshape((top(2:nx+1)*alpha + bottom(2:nx+1)*(1-alpha) +... 
Prob.x_0=x_0;
Prob.user.nx=nx;
:
nuse=nx*ny;

function bcvec=Enneper(bc,nx,ny)
:
```

Regularise the interface to use `Prob` structure

```matlab
function f = MinpackMSA_F(x,Prob)
    bottom = Prob.user.bottom;
    top = Prob.user.top;
    left = Prob.user.left;
    right = Prob.user.right;

otherwise similar to Lenton coding with loops and branching.
```
function f = MinpackMSA_Fvec(x,Prob)
:
bottom=Prob.user.bottom;
:
% transfer interior values x to entire grid v
v = zeros(nx+2,ny+2,'like',x); % 'like' ensures v has class of x
v(2:nx+1,2:ny+1) = reshape(x,nx,ny);
% apply boundary conditions
v(:,1) = bottom;
:
% computer dvdx and dvdy on each edge of the grid
dvdx = (v(2:nx+2,:)-v(1:nx+1,:))/hx;
dvdy = (v(:,2:ny+2)-v(:,1:ny+1))/hy;
% quadratic term over lower and upper elements
fL=sqrt(1+dvdx(1:nx+1,1:ny+1).^2+dvdy(1:nx+1,1:ny+1).^2);
fU=sqrt(1+dvdx(1:nx+1,2:ny+2).^2+dvdy(2:nx+2,1:ny+1).^2);
f = area*(sum(sum(fL+fU)));

No loops or branches!
From vectorised function - easy (!) to write vectorised gradient

```matlab
function fgrad = MinpackMSA_Gvec(x,Prob)
% coding as for MinpackMSA_Fvec:

%f quadractic term over lower and upper elements
fL=sqrt(1+dvdx(1:nx+1,1:ny+1).^2+dvdy(1:nx+1,1:ny+1).^2);
fU=sqrt(1+dvdx(1:nx+1,2:ny+2).^2+dvdy(2:nx+2,1:ny+1).^2);
%f gradient just use interior points
i=2:nx+1;
j=2:ny+1;
fgrad=area*(...
    (1/hx)*(1./fL(i-1,j)+1./fU(i-1,j-1)).*dvdx(i-1,j)...
    -(1/hx)*(1./fL(i,j)+1./fU(i,j-1)).*dvdx(i,j)...
    +(1/hy)*(1./fL(i,j-1)+1./fU(i-1,j-1)).*dvdy(i,j-1)...
    -(1/hy)*(1./fL(i,j)+1./fU(i-1,j)).*dvdy(i,j));
fgrad=fgrad(:);
```
Testing

Verification versus the Fortran Implementation
Unit Testing
Verification versus the Fortran Implementation

- Wish to validate Matlab coding against original Fortran coding.
- Only Intel Visual Fortran can be used to compile Fortran Mex files compatible with Matlab [Mathworks, 2015a].
- Mex-file interface difficult to write.
- Lenton [2005] used
  - Formatted text file used to transfer data (eg, $\mathbf{x}$) from Matlab to a Fortran program
  - Fortran program evaluated $\mathbf{F}(\mathbf{x})$, $\mathbf{JF}(\mathbf{x})$, $f(\mathbf{x})$, $\nabla f(\mathbf{x})$ etc.
  - Formatted text file used to transfer data back to Matlab.
  - Technique subject to truncation errors associated with I/O conversion.
- Present approach uses (unformatted) binary files to eliminate I/O conversion errors.
Matlab’s unit test framework [Mathworks, 2015b] used to run tests on randomised inputs.

Almost complete for Nonlinear Equations.
## Unit Testing For Nonlinear Equations

### Common Features

<table>
<thead>
<tr>
<th>Problem</th>
<th>version</th>
<th>$n$</th>
<th>$x_s$</th>
<th>$x_l/x_u$</th>
<th>$F$</th>
<th>$JF$</th>
<th>$F + JF$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPF</td>
<td>-</td>
<td>11</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CPR</td>
<td>-</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FDC</td>
<td>-</td>
<td>16,100</td>
<td>✓</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FIC</td>
<td>-</td>
<td>16,96</td>
<td>✓</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>HHD</td>
<td>-</td>
<td>8</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IER</td>
<td>-</td>
<td>18,93</td>
<td>✓</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SFD</td>
<td>-</td>
<td>14,98</td>
<td>✓</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SFI</td>
<td>-</td>
<td>16,100</td>
<td>✓</td>
<td>n/a</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Unit Testing For Nonlinear Equations

### Largescale Features

<table>
<thead>
<tr>
<th>Problem</th>
<th>n</th>
<th>S(JF)</th>
<th>JF · v</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDC</td>
<td>16,100</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FIC</td>
<td>16,96</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IER</td>
<td>18,93</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SFD</td>
<td>14,98</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SFI</td>
<td>16,100</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Unit Testing For Nonlinear Equations
Vectorised Features

<table>
<thead>
<tr>
<th>Problem</th>
<th>( n )</th>
<th>( \mathbf{F}_v )</th>
<th>( \mathbf{JF}_v )</th>
<th>( \mathbf{F}_v + \mathbf{JF}_v )</th>
<th>( S_v(\mathbf{JF}) )</th>
<th>( \mathbf{JF} \cdot \mathbf{v}_v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDC</td>
<td>16,100</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIC</td>
<td>16,96</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IER</td>
<td>18,93</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SFD</td>
<td>14,98</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SFI</td>
<td>16,100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results

5

Results

- HHD Problem
- MSA Problem
HHD Problem - Run time Ratios

<table>
<thead>
<tr>
<th>what</th>
<th>how</th>
<th>Problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Minpack</td>
<td>8</td>
</tr>
<tr>
<td>F+G</td>
<td>Minpack</td>
<td>1</td>
</tr>
<tr>
<td>F+G</td>
<td>fmad-sparse</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
</tr>
</tbody>
</table>

Run time ratio = \( \frac{\text{CPU(What by How)}}{\text{CPU(F)}} \)
# MSA Problem - Run time Ratios

<table>
<thead>
<tr>
<th>what</th>
<th>how</th>
<th>Problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>F</td>
<td>Minpack</td>
<td>1</td>
</tr>
<tr>
<td>F+G</td>
<td>Minpack</td>
<td>1.0000</td>
</tr>
<tr>
<td>F+G</td>
<td>fmad-sparse</td>
<td>5250</td>
</tr>
<tr>
<td>Fvec</td>
<td>vectorised</td>
<td>0.5000</td>
</tr>
<tr>
<td>Fvec+G</td>
<td>fmad-sparse</td>
<td>190.0000</td>
</tr>
</tbody>
</table>
Conclusions

- We have a Matlab implementation of all problems in the Minpack-2 Test Problem Collection.
- Lenton’s implementation has been validated.
- Moving to re-coded version with uniform interface and an implementation of the function alone to allow testing of AD for:
  - First Derivatives for Jacobians and gradients
  - Sparsity Patterns for Jacobians and Hessians
  - Jacobian-vector and Hessian-vector products
  - Impact of code vectorisation.
- Code vectorisation will have a major impact on efficiency of overloaded and source transformed AD codes - including compile-time costs for source transformation.
Advert - AD2016
7th International Conference on
Algorithmic Differentiation

- Monday 12th - Thursday 15th September 2016
- Christ Church Oxford, UK
- http://www.autodiff.org/ad16/
- Key dates:
  ▶ 1 January 2016 - Second Announcement and opening of abstract submission
  ▶ 30 March 2016 - Deadline for submission of 4 page extended abstracts
  ▶ 31 May 2016 - Notification of Acceptance of presentations and posters
  ▶ 30 December 2016 - submission of papers to Optimization Methods and Software
- Please start thinking about your submission!


References IV


