The New User Interface of ADiMat and How to Use it with DAE Solvers in Matlab and Octave

Johannes Willkomm\textsuperscript{\times}  Christian H. Bischof\textsuperscript{\times}
H. Martin Bütter\textsuperscript{*}

\textsuperscript{\times}Fachgebiet Scientific Computing
TU Darmstadt

\textsuperscript{*}Institute for Scientific Computing
RWTH Aachen University

Twelfth European Workshop on Automatic Differentiation
Humboldt-Universität zu Berlin
December 9, 2011
Overview

ADiMat

Usage

How ADiMat works

Performance

DAEs

Conclusion

J. Willkomm, C.H. Bischof, H.M. B"ucker

Solving DAEs with Matlab and Octave Using ADiMat 2
ADiMat

**ADiMat: Automatic Differentiation for Matlab**

- ... and for Octave
- [http://www.sc.rwth-aachen.de/adimat](http://www.sc.rwth-aachen.de/adimat)
- Developed by André Vehreschild first, now by me

**New in ADiMat**

- Reverse mode
- Second, alternative forward mode implementation
- Server for source transformation
- **High-level user interface**
Derivatives of Matlab functions

Consider Matlab function \([y \ z] = f(a, b)\)

- Jacobian matrix of derivatives:

\[
J = \frac{\partial (y, z)}{\partial (a, b)} = \begin{pmatrix}
\frac{\partial y}{\partial a} & \frac{\partial y}{\partial b} \\
\frac{\partial z}{\partial a} & \frac{\partial z}{\partial b}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
\frac{\partial y_1}{\partial a_1} & \cdots & \frac{\partial y_1}{\partial a_{n_a}} & \frac{\partial y_1}{\partial b_1} & \cdots & \frac{\partial y_1}{\partial b_{n_b}} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\frac{\partial y_{n_y}}{\partial a_1} & \cdots & \frac{\partial y_{n_y}}{\partial a_{n_a}} & \frac{\partial y_{n_y}}{\partial b_1} & \cdots & \frac{\partial y_{n_y}}{\partial b_{n_b}} \\
\frac{\partial z_1}{\partial a_1} & \cdots & \frac{\partial z_1}{\partial a_{n_a}} & \frac{\partial z_1}{\partial b_1} & \cdots & \frac{\partial z_1}{\partial b_{n_b}} \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
\frac{\partial z_{n_y}}{\partial a_1} & \cdots & \frac{\partial z_{n_y}}{\partial a_{n_a}} & \frac{\partial z_{n_y}}{\partial b_1} & \cdots & \frac{\partial z_{n_y}}{\partial b_{n_b}}
\end{pmatrix}
\in \mathbb{C}^{(n_y+n_z)\times(n_a+n_b)}
\]

- \(a_i\) is \(a(i)\), \(i\)-th component of multidimensional array \(a\)
- \(n_a\) is number of components in \(a\)
Forward mode in ADiMat

There are two alternative FM implementations in ADiMat

- **admDiffFor**: first order and second order (experimental)
- **admDiffVFor**: first order
- Some more on the differences later...

Example (FM with ADiMat)

\[
\begin{bmatrix} J & y & z \end{bmatrix} = \text{admDiffFor}(\@f, S, a, b)
\]

\[
\begin{bmatrix} J & y & z \end{bmatrix} = \text{admDiffVFor}(\@f, S, a, b)
\]

- **S**: Seed matrix \( S_{FM} \)
- **J**: Jacobian matrix product \( J \cdot S_{FM} \)
- **S = 1**: Shortcut for \( S_{FM} = I_{n_a+n_b} \), conforming identity matrix
  - Compute full Jacobian \( J \)
- **y, z**: Function results returned by AD process
Reverse mode in ADiMat

Example (RM with ADiMat)

\[ [J \ y \ z] = \text{admDiffRev}(@f, S, a, b) \]

- \textbf{S}: Seed matrix \( S_{RM} \)
- \textbf{J}: Jacobian matrix product \( S_{RM} \cdot J \)

- Basic support for recomputation (on the function call level)
  - Directive \(<\text{recompute}>g</\text{recompute}>\)
  - Recompute following call to function \( g \)

- Several stack implementations
  - Keep data in memory
  - Write data to file (asynchronously)
Options

Example (Passing options to ADiMat)

```matlab
opts = admOptions(name1, value1, name2, value2, ...)
```

- Create options structure

```matlab
J = admDiffFor(@f, S, a, b, opts)
```

- Pass options structure as last argument
  - Works with any number of function arguments

Example (Specifying independent and dependent variables)

```matlab
J = admDiffFor(@f, S, a, b, admOptions('i', 1, 'd', [1, 2]))
```

- J: left “half” of Jacobian, i.e. $\frac{\partial(y,z)}{\partial a} \cdot S_{FM}$
- Independent variables: first parameter $a$
- Dependent variables: both output parameters $y$ and $z$
- 'i', 'd' are shortcuts for 'independents', 'dependents'
Compressed Jacobian computation

Often Jacobian $J$ is sparse

- \textit{Sparsity exploitation} can reduce the number of derivative directions $n_{dd}$
- Non-zero (NZ) pattern $P$ of $J$ must be known

Example (Compressed Jacobian computation with ADiMat)

```
opts = admOptions('JPattern', P)
J = admDiffFor(@f, @cpr, a, b, opts)
```

- $P$: Non-zero pattern $P$
- $J$: Full Jacobian $J$ returned as sparse matrix
- $cpr$: Curtis-Powell-Reed heuristic
Alternatives to AD in ADiMat

ADiMat provides two non-AD methods to compute derivatives

- Can use the same options shown before, including sparsity exploitation

Example (Finite difference (FD) method)

\[
[JFD \ y \ z] = \text{admDiffFD}(@f, S, a, b)
\]

- Central, forward, and backward, up to fourth order derivatives, also higher accuracy order stencils

Example (Complex variable method [LYNESS AND MOLER 1967])

\[
[JCV \ y \ z] = \text{admDiffComplex}(@f, S, a, b)
\]

Example (Nested application of ADiMat)

\[
\text{Hessian} = \text{admDiffComplex}(@\text{admDiffFVor}, 1, @f, 1, x, y, ... \text{admOptions}(’i’, 3:4))
\]

- AD over AD will in general not work
How ADiMat works

\texttt{admDiffFor(@f, S, a, b, opts)} works in two steps

- Source transformation (ST)
  - Differentiate the source code, produce function \( \texttt{d\_f} \)
  - Only done if necessary
- Derivative evaluation (DE)
  - Actually compute the numbers, calling \( \texttt{d\_f} \)

\[
[J \ y \ z] = \texttt{admDiffFor(@f, S, a, b, opts)}
\]
Source Transformation

Source code is differentiated by transformation server

```matlab
function [y z] = f(a, b)
    y = a + sqrt(b);
    z = sqrt(a) .* b;
end
```

Example (Differentiated function `d_f`)

```matlab
function [d_y y d_z z] = d_f(d_a, a, d_b, b)
    [d_tmpca1 tmpca1] = diff_sqrt(d_b, b);
    d_y = opdiff_sum(d_a, d_tmpca1);
    y = a + tmpca1;
    [d_tmpca1 tmpca1] = diff_sqrt(d_a, a);
    d_z = opdiff_emult(d_tmpca1, tmpca1, d_b, b);
    z = tmpca1 .* b;
end
```

- Redone, when source code or options are modified
Derivative Evaluation

Run differentiated code \texttt{d\_f} to evaluate derivatives

- Derivative arguments \texttt{d\_a}, \texttt{d\_b} are created from \texttt{a}, \texttt{b} and \texttt{S}
- Extract derivatives from outputs \texttt{d\_y}, \texttt{d\_z} and create \texttt{J}

Scalar mode

- \texttt{d\_f} is called \( n_{dd} \) times
- Derivative variable \texttt{d\_a} is double array with same shape as \texttt{a}
  - Fast execution, but redundant computations when \( n_{dd} > 1 \)

Vector mode

- Single call of \texttt{d\_f}
- Derivative variable \texttt{d\_a} is \textit{derivative class} object
  - Object internally holds \( n_{dd} \cdot n_a \) derivative values
  - Dispatching of overloaded operators at runtime, hence slow
- \texttt{admDiffVFor}: \texttt{d\_a} is double array with \( n_{dd} \cdot n_a \) items
  - Overloaded operators replaced by function calls
  - Only possibility for vector mode in Octave
Performance

Factor $T_\partial / T_f$, For: admDiffFor, /D: double deriv., /O: deriv. objects

Non-vectorized code: Multiphase flow in porous media

[Büsing et al. 2011]

<table>
<thead>
<tr>
<th>$n_{dd}$</th>
<th>For/D</th>
<th>For/O</th>
<th>VFor</th>
<th>Rev/D</th>
<th>Rev/O</th>
<th>FD</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.15</td>
<td>17.2</td>
<td>3.15</td>
<td>18.6</td>
<td>58.3</td>
<td>2.03</td>
<td>1.02</td>
</tr>
<tr>
<td>10</td>
<td>21.2</td>
<td>26.4</td>
<td>3.19</td>
<td>185</td>
<td>67.9</td>
<td>20.4</td>
<td>10.2</td>
</tr>
<tr>
<td>100</td>
<td>211</td>
<td>119</td>
<td>3.28</td>
<td>1853</td>
<td>213</td>
<td>203</td>
<td>102</td>
</tr>
</tbody>
</table>

$T_f = 13.3s$, $f$: 761 LOC, 372 statements, 16 functions, 75 function calls

Vectorized code: 1D Burgers PDE Solver

Research code by Micheal Herty

<table>
<thead>
<tr>
<th>$n_{dd}$</th>
<th>For/D</th>
<th>For/O</th>
<th>VFor</th>
<th>Rev/D</th>
<th>Rev/O</th>
<th>FD</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.43</td>
<td>42.4</td>
<td>9.87</td>
<td>32.7</td>
<td>129</td>
<td>2.03</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>32.4</td>
<td>97.3</td>
<td>32.5</td>
<td>332</td>
<td>235</td>
<td>20.1</td>
<td>–</td>
</tr>
<tr>
<td>100</td>
<td>324</td>
<td>674</td>
<td>304</td>
<td>3350</td>
<td>1348</td>
<td>201</td>
<td>–</td>
</tr>
</tbody>
</table>

$T_f = 1.64s$, $f$: 169 LOC, 57 statements, 6 functions, 27 function calls, admDiffComplex is not applicable
Matlab: ode15s

Matlab has ODE solvers, e.g. ode15s, which can also solve index 1 DAEs [Shampine, Reichelt, and Kierzenka 1999]

- Solves $M(t, y)\dot{y} = f(t, y)$
- This is a DAE when mass matrix $M(t, y)$ is singular.
- Inputs required: function $f$, initial $y_0$, start and end times $t_0$ and $t_f$, mass matrix or function
- Consistent initial $\dot{y}_0$ automatically computed (or via option)
- Use AD: User can provide a handle to function that computes the Jacobian of $f$ w.r.t. $y$
Octave: DASSL, DASPK and ODE package

Octave provides interfaces to the DAE solvers DASSL [Petzold 1982] and DASPK [Brown, Hindmarsh, and Petzold 1999]

- Solve equation $0 = f(y, \dot{y}, t)$
- Inputs required: $f$, $y_0$, $\dot{y}_0$, vector of time points $t$
- Option to compute consistent initial conditions
- Use AD: User can provide a handle to function that computes the Jacobian $J_c = \frac{\partial f}{\partial y} + c \frac{\partial f}{\partial \dot{y}}$ for a given $c$

There is also an ODE package which provides drop-in replacements for Matlab’s ODE solvers
Example: Binary Distillation Column

Model of 42 differential and 83 algebraic equations [DIEHL 2001]

- Adapted version found in the Nonlinear Model Library from hedengred.net (Distillation 4)
- Coded as function \( xdot = \text{distill}(t,y,\text{mode}) \)
Distillation example with Matlab and AD

The call to `ode15s` looks like this:

```matlab
opts = odeset('Mass', distill(t, y_0, 'mass'), ...
               'MassSingular', 'yes', ...
               'MStateDependence', 'none', ...)
               'JPattern', distill(t, y_0, 'jpat')));
[t, y] = ode15s(@distill, [0 tf], y_0, opts);
```

Use AD: set `opts.Jacobian` to a function that computes \( \frac{\partial f}{\partial y} \)

```matlab
adopts = admOptions('i', 2);
adopts.JPattern = distill(t, y_0, 'jpat');
opts.Jacobian = @(t, y) ...
                admDiffVFor(@distill, @cpr, t, y, '', adopts);
```

- Pass the non-zero pattern of \( J \) to ADiMat instead of `ode15s`
Distillation example with Octave and DASSL

First, a wrapper function to adapt \texttt{distill} to \texttt{dassl} interface

\begin{verbatim}
function res = dassl_distill(y,ydot,t)
    global M
    res = M*ydot - distill(t,y,"");
\end{verbatim}

Use AD: give DASSL a second function handle, for \(J_c\):

\begin{verbatim}
global JPat
adopts = admOptions('i',1:2);
adopts.JPattern = JPat;
adopts.coloringFunction = 'cpr';

seed = [speye(numel(y))
c.*speye(numel(y))];
jac = admDiffVFor(@dassl_distill, seed, y, ...
ydot, t, adopts);
\end{verbatim}

\begin{equation}
    S = \begin{pmatrix}
        1 \\
        \vdots \\
        c \\
        \vdots \\
        c
    \end{pmatrix}
\end{equation}

This way \(J_c = J \cdot S\) will be compressed
Binary Distillation Column results

- $J$ compressed to 68, $J_c$ to 86 columns
- $n_F$: calls to `distill`, $n_\partial$: calls to derivative function
- $t$: solve time

<table>
<thead>
<tr>
<th>Derivative</th>
<th>$n_F$</th>
<th>$n_\partial$</th>
<th>$t$/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matlab</td>
<td>–</td>
<td>109</td>
<td>–</td>
</tr>
<tr>
<td>FD</td>
<td>41</td>
<td>1</td>
<td>0.60</td>
</tr>
<tr>
<td>VFor</td>
<td>41</td>
<td>1</td>
<td>0.88</td>
</tr>
<tr>
<td>Octave</td>
<td>–</td>
<td>1791</td>
<td>–</td>
</tr>
<tr>
<td>FD</td>
<td>41</td>
<td>14</td>
<td>59.7</td>
</tr>
<tr>
<td>VFor</td>
<td>41</td>
<td>14</td>
<td>28.6</td>
</tr>
</tbody>
</table>

`distill`: 442 LOC, 148 statements, 1 function, 17 function calls

Lenovo T420s: Core i5-2520M @2.5 GHz, Linux 3.0.0, Matlab R2011a, Octave 3.2.4, and ADiMat 0.5.6-3150
Example: Food web with DASPK and Octave

From DASPK paper [Brown, Hindmarsh, and Petzold 1999]
- One prey, one predator species on \((L + 2) \times (L + 2)\) grid
- No preconditioning
- \(F\): Loops over grid, matrix operations for species interaction
- Jacobian \(J_c\) can be compressed to 11 columns
Example: Food web with DASPK and Octave

- $t_F$: time in $F$, $t_\partial$: time in derivative function

<table>
<thead>
<tr>
<th>Derivative</th>
<th>$L$</th>
<th>$n_F$</th>
<th>$n_\partial$</th>
<th>$t$/s</th>
<th>$t_F$/s</th>
<th>$t_\partial$/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD</td>
<td>20</td>
<td>741</td>
<td>53</td>
<td>126</td>
<td>38.1</td>
<td>75.8</td>
</tr>
<tr>
<td>VFor</td>
<td>20</td>
<td>921</td>
<td>49</td>
<td>133</td>
<td>47.3</td>
<td>73.7</td>
</tr>
<tr>
<td>FD</td>
<td>40</td>
<td>1111</td>
<td>57</td>
<td>850</td>
<td>207</td>
<td>338</td>
</tr>
<tr>
<td>VFor</td>
<td>40</td>
<td>995</td>
<td>56</td>
<td>848</td>
<td>189</td>
<td>340</td>
</tr>
<tr>
<td>FD</td>
<td>60</td>
<td>917</td>
<td>55</td>
<td>3556</td>
<td>375</td>
<td>828</td>
</tr>
<tr>
<td>VFor</td>
<td>60</td>
<td>925</td>
<td>51</td>
<td>3321</td>
<td>371</td>
<td>760</td>
</tr>
</tbody>
</table>

$F$: 58 LOC, 35 statements, 2 functions, 16 function calls

Lenovo T420s

- For $L = 20$ and $L = 60$ FD results in less calls to $F$
- In all cases less derivative evaluations with AD
Conclusion

New user interface for ADiMat

- Makes sophisticated AD features easily accessible
  - Two alternative implementations of the forward mode of AD
  - Reverse mode of AD
  - Support for compressed Jacobian computation

- Try it out
  - http://www.sc.rwth-aachen.de/adimat

- Please do report bugs
  - ADiMat is far from complete
  - We need feedback to enhance ADiMat to suit your needs
  - Tell us which builtin functions you miss

Solving DAEs with AD

- AD runtime is comparable to numerical methods
Lawrence F. Shampine, Mark W. Reichelt, and Jacek A. Kierzenka
Solving Index 1 DAEs in MATLAB and Simulink
SIAM Review, Vol 41, 1999

Peter N. Brown, Alan C. Hindmarsh, and Linda R. Petzold,
Using Krylov Methods in the Solution of Large-Scale Differential Algebraic Systems