Accurate Derivatives for Computational Engineering with Automatic Differentiation for Matlab (ADiMat)

Johannes Willkomm× Christian H. Bischof×
H. Martin Bücker*

× Fachbereich Informatik
TU Darmstadt

*Institute for Scientific Computing
RWTH Aachen University

2nd International Conference on Computational Engineering
October 6, 2011
Overview

ADiMat

Usage

Example

How ADiMat works

Performance

Conclusion
Automatic Differentiation for Matlab

- **Automatic Differentiation (AD)**
  - Automatically differentiate numeric programs
  - Available for e.g. C/C++, Fortran, Python, Matlab
  - Accurate derivatives at arbitrary points
  - See [http://www.autodiff.org](http://www.autodiff.org)

- **Matlab**
  - Matlab is popular for prototyping and rapid development
  - Can also solve large problems efficiently
  - Is easy to learn
  - Provides a wealth of high-level built-in functions
  - Open-source alternative: GNU Octave

- **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

- **Other Matlab AD tools**: TOMLAB/MAD and INTLAB
Where derivatives are needed

Whenever dealing with non-linear problems in numerics, derivatives are necessary

Simulation

- PDE-Solvers
  - Finite Element, Finite Volume, or Finite Difference schemes
- Non-linear equation systems
  - Newton method
  - Need *Jacobian* matrix

Optimization

- Inverse problems, Shape optimization, Parameter estimation
- Non-linear optimization
  - Need *Gradient* of cost/objective function
  - Possibly second-order derivative information
Consider Matlab function \([y \ z] = f(a, b)\)

- Jacobian matrix of derivatives:

\[
J = \frac{\partial(y, z)}{\partial(a, b)} = \left( \begin{array}{c|c}
\frac{\partial y}{\partial a} & \frac{\partial y}{\partial b} \\
\hline
\frac{\partial z}{\partial a} & \frac{\partial z}{\partial b}
\end{array} \right)
\]

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

- \(a_i\) is \(a(i)\), \(i\)-th component of multidimensional array \(a\)
- \(n_a\) is number of components in \(a\)
AD in a nutshell

Forward Mode (FM)

- Computes \( J \cdot S_{FM} \), with seed matrix \( S_{FM} \in \mathbb{C}^{(n_a+n_b) \times n_{dd}} \)
  - \( n_{dd} \): Number of derivative directions
  - Time: \( T_\partial / T_f < c \cdot n_{dd} \)
  - Memory: \( M_\partial / M_f < c \cdot n_{dd} \)

Reverse Mode (FM)

- Computes \( S_{RM} \cdot J \), \( S_{RM} \in \mathbb{C}^{n_{dd} \times (n_y+n_z)} \)
  - Time: \( T_\partial / T_f < c \cdot n_{dd} \)
  - Memory: \( M_\partial \approx T_f \), without checkpointing

\[ n_y + n_z \begin{bmatrix} J \end{bmatrix} \times \begin{bmatrix} S_{FM} \end{bmatrix} \text{ “for free”} \]

\[ n_{dd} \triangleq \text{costs} \]

\[ n_{dd} \triangleq \text{costs} \]

\[ n_y + n_z \begin{bmatrix} S_{RM} \end{bmatrix} \times \begin{bmatrix} J \end{bmatrix} \text{ “for free”} \]

\[ n_{dd} \triangleq \text{costs} \]
Forward mode in ADiMat

There are two alternative FM implementations in ADiMat

- Some more on that later...

Example (FM with ADiMat)

\[
[J \ y \ z] = \text{admDiffFor}(\mathbf{f}, S, a, b) \\
[J \ y \ z] = \text{admDiffVFor}(\mathbf{f}, S, a, b)
\]

- \(S\): 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)

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

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

- RM evaluates derivatives in reverse direction of program flow
  - Chain rule of differentiation is associative
  - Have to store every single variable value, plus program flow
  - Can use recomputation to do better
- RM can be used to efficiently compute gradients
  - Costs do not depend on number of input parameters
  - RM computes discrete adjoints ("first discretize, then differentiate")
- Data storage options in ADiMat:
  - Keep data in memory
  - Write data to file (asynchronously)
Options

Example (Passing options to ADiMat)

opts = admOptions(name1, value1, name2, value2, ...)
- Create options structure

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)

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 fields independents, dependents
- b is treated as a constant parameter
Compressed Jacobian computation

Often Jacobian $J$ is sparse

- *Sparsity exploitation* can reduce the required $n_{dd}$
- Non-zero (NZ) pattern $P$ of $J$ (a bit matrix) must be known

Example (Compressed Jacobian computation with ADiMat)

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

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

ADiMat provides two non-AD methods to compute derivatives

Example (Finite difference (FD) method)

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

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

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

- Can use the same options shown before, including sparsity exploitation
- Will skip details in the talk
Finite difference method

Example (Finite difference (FD) method)

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

- **JFD**: product \( J \cdot S_{FM} \), as in FM of AD
- Uses central FD by default

\[
\frac{d f}{d x} \approx \frac{f \left( x + \frac{h}{2} \right) - f \left( x - \frac{h}{2} \right)}{h}
\]

- Forward and backward FD via option
- Accuracy half the machine precision, with right \( h \) (option)
- Not useful near discontinuities
Complex variable method

Example (Complex variable method [Lyness and Moler 1967])

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

\[ \ni \quad J_{CV}: \text{product } J \cdot S_{FM}, \text{as in FM of AD} \]

\[ \ni \quad \text{Uses} \]

\[
\frac{df}{dx} \approx \frac{\Im[f(x + i\epsilon)]}{\epsilon}
\]

\[ \ni \quad \text{Accuracy as with AD, full machine precision, any tiny } \epsilon \text{ will do} \]

\[ \ni \quad \text{Function } f \text{ must be } \textit{real analytic} \]

\[ \ni \quad \text{Time: } T_{\partial}/T_f < c \cdot n_{dd} \]

\[ \ni \quad \text{Potential problems when program flow changes} \]
Usage example

Example (Function \( f \))

```
function \([y \ z] = f(a, \ b)\)
    \(y = a + \sqrt{b}\);
    \(z = \sqrt{a} \times b\);
```

Example (Run \( f \))

```
>> a = [9 16];
>> b = [25 36];
>> [y \ z] = f(a, \ b)
```

```
y = 14 22
z = 75 144
```

Example (AD of \( f \))

```
>> J = admDiffFor(@f, 1, a, b)
```

```
J = 1.0000 0 0.1000 0
    0 1.0000 0 0.0833
    4.1667 0 3.0000 0
    0 4.5000 0 4.0000
```
How ADiMat works

`admDiffFor(@f, S, a, b, opts)` works in two steps

- **Source transformation (ST)**
  - Differentiate the source code, produce function `d_f`
  - Only done if necessary

- **Derivative evaluation (DE)**
  - Actually compute the numbers, calling `d_f`

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

Source code is differentiated by transformation server

\[ f.m \]

Server \texttt{adimat.sc.rwth-aachen.de}

Example (Differentiated function \texttt{d_f})

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

Aside: This is actually the code produced by \texttt{admDiffVFor}

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
Why source transformation

The source transformation step in ADiMat is difficult, but:

- ST can analyze the code
  - *Activity analysis*
    - Do not differentiate code that does not influence result
    - Derivative code becomes shorter
    - Avoid unnecessary derivative computations
  - *Optimizations*
    - Can apply optimizations to code before running it
    - Before and after differentiation
    - May consider more than a single expression/statement
    - We lack type information, though

- ST can create code that suits the needs of AD
  - E.g. replace calls of overloaded operators by function calls
  - Shift work from runtime to transformation and “compile” (to intermediate code) time

- We have a flexible ST framework for Matlab that might be useful for other transformation tasks (ideas?)
### Performance

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

**Non-vectorized code: Two-phase flow in porous media**

Henrik Büsing, c.f. ICCE talk “Using exact Jacobians in an implicit ...”

<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

**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, admDiffComplex is not applicable
Conclusion

ADiMat, AD tool for Matlab and Octave

- Is easy to use
- Provides sophisticated features
  - 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