Introduction to Automatic Differentiation

Johannes Willkomm

PLEIAD Seminar, Universidad de Chile
27 Nov. 2009
Santiago de Chile
Outline of the talk

• Automatic Differentiation (AD)
  – Definition by example
  – Forward and reverse mode
  – Scalar and vector mode

• AD implementation
  – Source transformation and operator overloading
  – Reverse mode example
  – Tools

• Alternatives to AD
  – Divided differences, Complex-Variable method
  – Symbolic and manual differentiation

• Summary
• **Automatic or Algorithmic Differentiation (AD)**
  - Given a numeric program, that implements function $F$
  - AD creates a new program that computes $F'$, the first order derivative of $F$
  - And sometimes also the higher order derivatives $F''$, $F'''$, $F^{IV}$, etc.
• Consider the beam of a lighthouse rotating with angular velocity $\omega$ as it runs along a quay with slope $\gamma$ at distance $v$, as a function of time $t$. 

\[
y = \gamma x
\]
Lighthouse example

- The coordinates of the point where the light hits the quay are given by
  \[ x = \frac{v \tan(\omega t)}{\gamma - \tan(\omega t)} \]
  \[ y = \frac{\gamma v \tan(\omega t)}{\gamma - \tan(\omega t)} \]

- A program implementing this function
  \[ v_1 = \omega \times t; \]
  \[ v_2 = \tan(v_1); \]
  \[ v_3 = \gamma - v_2; \]
  \[ v_4 = v \times v_2; \]
  \[ x = v_4 / v_3; \]
  \[ y = \gamma \times x; \]
• Program code can be mechanically differentiated
  
  - Differentiate each statement and insert it before the original statement

  \[
  \begin{align*}
  v_1 &= \omega \times t; \quad \Rightarrow \quad \delta v_1 &= \delta \omega \times t + \omega \times \delta t; \\
  v_2 &= \tan(v_1); \quad \Rightarrow \quad \delta v_2 &= \delta v_1 / \cos^2(v_1); \\
  v_3 &= \gamma - v_2; \quad \Rightarrow \quad \delta v_3 &= \delta \gamma - \delta v_2; \\
  v_4 &= \nu \times v_2; \quad \Rightarrow \quad \delta v_4 &= \delta \nu \times v_2 + \nu \times \delta v_2; \\
  x &= v_4 / v_3; \quad \Rightarrow \quad \delta x &= \delta v_4 / v_3 + v_4 \delta v_3 / v_3^2; \\
  y &= \gamma \times x; \quad \Rightarrow \quad \delta y &= \delta \gamma \times x + \gamma \times \delta x;
  \end{align*}
\]
• The AD code has new input and output variables
  \( \delta t, \delta \gamma, \delta \nu, \text{ and } \delta \omega \) are new inputs
  \( \delta x, \delta y \) are new results

• The user must set the input derivatives
  – \( \delta t = \frac{dt}{dp}, \delta \gamma = \frac{d\gamma}{dp}, \delta \nu = \frac{d\nu}{dp}, \text{ and } \delta \omega = \frac{d\omega}{dp} \),
    where \( p \) is the parameter to differentiate to

• Examples:
  – Setting \( \delta t = 1, \delta \gamma = 0, \delta \nu = 0, \text{ and } \delta \omega = 0 \), the AD code computes \( \frac{dx}{dt} \) and \( \frac{dy}{dt} \)
  – Setting \( \delta t = 0, \delta \gamma = 1, \delta \nu = 0, \text{ and } \delta \omega = 0 \), the AD code computes \( \frac{dx}{d\gamma} \) and \( \frac{dy}{d\gamma} \), etc.

• To get all eight derivatives, the code must be run four times: this is the scalar forward mode
We can also transform the derivative variables into vectors

- Using 4-vectors we can compute all derivatives at once

Example

- Set $\delta t = [1,0,0,0]$, $\delta \gamma = [0,1,0,0]$, $\delta \nu = [0,0,1,0]$, and $\delta \omega = [0,0,0,1]$
- As the result we obtain the full Jacobian matrix $J = DF$

$$
\delta x = \begin{bmatrix}
\frac{dx}{dt} & \frac{dx}{d\gamma} & \frac{dx}{d\nu} & \frac{dx}{d\omega}
\end{bmatrix}
$$

$$
\delta y = \begin{bmatrix}
\frac{dy}{dt} & \frac{dy}{d\gamma} & \frac{dy}{d\nu} & \frac{dy}{d\omega}
\end{bmatrix}
$$
Formalize forward AD

- To differentiate a program
  - Create new variable $\delta v$ for each program variable $v$
  - Differentiate each statement and insert it before the original statement
  - Each $\delta v$ holds the derivative $dv/dp$ of $v$ w.r.t. the input parameter $p$

$$z = f(u, v, w); \quad \delta z = \frac{\partial f}{\partial u} \delta u + \frac{\partial f}{\partial v} \delta v + \frac{\partial f}{\partial w} \delta w;$$
Reverse mode AD

- AD is also possible by running the program backwards
- For each statement we propagate the derivative of the LHS to the derivatives of the variables on the RHS
  - Create the so-called adjoint statements

\[
\delta u = \delta u + \frac{\partial f}{\partial u} \delta z; \\
\delta v = \delta u + \frac{\partial f}{\partial v} \delta z; \\
\delta w = \delta u + \frac{\partial f}{\partial w} \delta z;
\]
Reverse mode AD

- **Forward sweep**
  - The program is executed, saving all variable values

- **Initialize adjoints**
  - Initialize all derivative variables $\delta v$ to zero

- **Return sweep**
  - Execute the adjoint statements in reverse order
  - Now, at any one time, $\delta v$ contains the adjoint $df/dv$ of $v$
Lighthouse in reverse

- Run code
- Zero adjoints
- Run adjoint code

\[
\begin{align*}
\delta x & += \gamma \ast \delta y; \\
\delta \gamma & += x \ast \delta y; \\
\delta t & = 0; \\
\delta v_4 & += \delta x / v_3; \\
\delta v_3 & += -v_4 / v_3^2 \ast \delta x; \\
\delta v_2 & += v_2 \delta v_4; \\
\delta v_1 & = 0; \\
\delta v_2 & += \nu \delta v_4; \\
\delta v_3 & += 0; \\
\delta v_2 & += -\delta v_3; \\
\delta v_1 & += \delta v_2 / \cos^2(v_1); \\
\delta \omega & += t \ast \delta v_1; \\
\delta t & += \omega \ast \delta v_1; \\
\end{align*}
\]

\[
\begin{align*}
v_1 & = \omega \ast t; \\
v_2 & = \tan(v_1); \\
v_3 & = \gamma - v_2; \\
v_4 & = \nu \ast v_2; \\
x & = v_4 / v_3; \\
y & = \gamma \ast x;
\end{align*}
\]
Running reverse mode AD code

- The adjoint code has new in- and outputs
  - $\delta x, \delta y$ are new inputs
  - $\delta t, \delta \gamma, \delta \nu, \text{ and } \delta \omega$ are new results
- Values for $\delta x$ and $\delta y$ are supplied by the user
  - $\delta x = \frac{dx}{dr}$ and $\delta y = \frac{dy}{dr}$ where $r$ is the result to differentiate
- Example
  - Setting $\delta x = 1$ and $\delta y = 0$, the code computes $\frac{dx}{dt}$, $\frac{dx}{dy}$, $\frac{dx}{dv}$, and $\frac{dx}{d\omega}$
  - Setting $\delta x = 0$ and $\delta y = 1$, the code computes $\frac{dy}{dt}$, $\frac{dy}{dy}$, $\frac{dy}{dv}$, and $\frac{dy}{d\omega}$
- To get all eight derivatives, the code must be run twice, or with 2-vectors as input adjoints
First order AD in general

- Given a function
  \[ y = F(x), \quad F : \mathbb{R}^n \rightarrow \mathbb{R}^m \]
  - First order AD computes the Jacobian
    \[ J = DF \in \mathbb{R}^{m \times n} \]
  - Or products thereof

- AD in forward mode
  \[ J \cdot S, \quad S \in \mathbb{R}^{n \times p} \]
  - Computes Jacobian times vector or Jacobian time matrix products

- AD in reverse mode
  \[ S \cdot J, \quad S \in \mathbb{R}^{p \times m} \]
  - Computes vector times Jacobian or matrix times Jacobian products
AD complexity

- The time complexity depends on the number of rows or columns in $S$ and the runtime $T_F$ of $F$
  - Computing $J$ has $T_F O(m)$ in RM and $T_F O(n)$ in FM
  - The $c$ in $O$ is $3 < c < 50$, depending on tool & strategy

\[
S \in \mathbb{R}^{p \times m} \quad J \in \mathbb{R}^{m \times n} \quad J \in \mathbb{R}^{m \times n} \quad S \in \mathbb{R}^{n \times p}
\]

- Space complexity is $O(T_F)$ in RM!
• Source transformation
  – New program text is generated
  – Higher order derivatives often not directly supported, but by repeatedly applying the tool

• Operator Overloading
  – Numeric data type (\texttt{double}) is replaced by new type
  – Tapeless: Derivatives are stored inside the active variables and updated on the fly
    • Forward mode only
  – With Taping: Computations are first recorded on a so-called Tape, which is then read (forwards or backwards) to compute the derivatives
  – Higher order derivatives are not much more difficult to implement than first order
• Compute polynomial of order $n$

$$F(x, c) = \prod_{i=0}^{n} c_i x^i$$

• A C-style implementation in MATLAB
  - If $x$, $c_i$ are all scalars that could also be a one-liner

```matlab
function r = polynom(x, c)
    r = 0;
    powX = 1;
    for i = 1:length(c)
        r = r + c(i) .* powX;
        powX = powX .* x;
    end
end
```
function \([a\_x\ a\_c\ nr\_r] = a\_polynom(x, c, a\_r)\)
\[
tmpc1 = 0; \\
r = 0; \\
powX = 1; \\
tmpfl = length(c); \\
for i=1 : tmpfl \\
push(tmpc1); \\
tmpc1 = c(i) .* powX; \\
push(r); \\
r = r + tmpc1; \\
push(powX); \\
powX = powX .* x; \\
end \\
push(tmpfl); \\
r\_r = r;
\]

- **Forward sweep**
  - Run (canonicalized) code
  - Save all values overwritten
  - Save control flow

\[
[a\_powX\ a\_tmpc1] = a\_zeros(powX, tmpc1); \\
[a\_x\ a\_c] = a\_zeros(x, c); \\
if nargin < 3 \\
[a\_r] = a\_zeros(r); \\
end \\
[tmpfl] = pop; \\
for i=flip1r(1 : tmpfl) \\
  [powX] = pop; \\
  a\_x = a\_x + adjred(x, powX .* a\_powX); \\
  a\_powX = adjred(powX, a\_powX .* x); \\
  [r] = pop; \\
  a\_tmpc1 = a\_tmpc1 + adjred(tmpc1, a\_r); \\
  a\_r = adjred(r, a\_r); \\
  [tmpc1] = pop; \\
  a\_c(i) = a\_c(i) + adjred(c(i), a\_tmpc1 .* powX); \\
  a\_powX = a\_powX + adjred(powX, c(i) .* a\_tmpc1); \\
  [a\_tmpc1] = a\_zeros(tmpc1); \\
end
\]

- **Return sweep**
  - Zero adjoints
  - Run backwards
  - Compute adjoints
<table>
<thead>
<tr>
<th>Tool</th>
<th>Language</th>
<th>FM</th>
<th>RM</th>
<th>ST</th>
<th>OO</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADOL-C</td>
<td>C/C++</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>CppAD</td>
<td>C/C++</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ADiFor</td>
<td>Fortran 77</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Tapenade</td>
<td>Fortran 77,</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fortran 90/95</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADiMat</td>
<td>Matlab</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MAD</td>
<td>Matlab</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ADiCape</td>
<td>CapeML</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Alternative ways to compute derivatives

- **Divided differences**
  - Very inaccurate
  - Difficult to find the right value for $h$
  - Only function $F$ is required
  - Only $Jv$ with complexity $O(n)$

$$\frac{df}{dx_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h}$$

- **Complex variable method**
  - Program needs to be changed similar to AD with OO
  - Derivatives are exact, if $h$ is just small enough
  - Need to provide new operations $>$, $<$, $\text{abs}$
  - Only $Jv$ with complexity $O(n)$

$$\frac{df}{dx_k} = \Im \{f(x + hie_k)\}$$
• The CV-Method is more precise
  – Usually up to machine precision
• And it is safer to use
  – Just set $h$ to a very small value, e.g. $h = 10^{-60}$
Alternative ways to compute derivatives

• Symbolic differentiation
  – May be difficult to write a whole program as one expression
  – Large derivative expressions with lots of repeated subexpressions
  – Often very large runtimes
    • Especially for higher order derivatives
    • Differentiation has to be done only once however

• Manual differentiation
  – Usually efficient derivative code
  – Often tedious and error-prone, especially when $F$ is changed
  – Discretization of $F$ and $F'$ has to be taken into account
• Let $F$ be defined by a PDE
  – Usually implemented by discretization
  – e.g. using the Finite Element Method

• Derivative $F'$ often by discretizing the adjoint PDE
  – The discretization introduces errors in both $F$ and $F''$
  – AD of the discretized $F$ differentiates through the discretization errors of $F$
• Solving Inverse Heat Conduction Problem with Conjugate-Gradient optimization using both AD gradient and gradient obtained from adjoint PDE
  – The objective function $J$ drops faster with AD
  – “faster” means fewer number of iterations here

![Graph showing the objective function $J$ dropping with iterations $n$.]
• **AD advantages**
  - AD can provide derivatives of that are efficient, precise, and reliable
  - AD is often easy to apply

• **AD disadvantages**
  - AD tools can be difficult to use and may lack support for language elements and/or higher order derivatives
  - Applying the reverse mode of AD needs special measures to cope with the memory requirements
    - Possible, but not discussed here

• When you need derivatives you should use AD
• You should consult with an AD expert
"Evaluating Derivatives", 2nd edition
Andreas Griewank & Andrea Walther
SIAM, Philadelphia 2008