Automatic Differentiation of programs and its applications to Scientific Computing

Laurent Hascoët

INRIA Sophia-Antipolis, France
http://www-sop.inria.fr/tropics

Trieste, april 18th, 2005
Outline

1. Introduction
2. Formalization
3. Multi-directional
4. Reverse AD
5. Alternative formalizations
6. Reverse AD for Optimization
7. Performance issues
8. Static Analyses in AD tools
9. Some AD Tools
10. Validation
11. Expert-level AD
12. Conclusion
Given a program $P$ computing a function $F$

$$F : \mathbb{R}^m \rightarrow \mathbb{R}^n$$

$X \leftrightarrow Y$

we want to build a program that computes the derivatives of $F$.

Specifically, we want the derivatives of the dependent, i.e. some variables in $Y$, with respect to the independent, i.e. some variables in $X$. 
Which derivatives do you want?

Derivatives come in various shapes and flavors:

- **Jacobian Matrices:** \( J = \left( \frac{\partial y_j}{\partial x_i} \right) \)

- **Directional or tangent derivatives, differentials:**
  \[ dY = \dot{Y} = J \times dX = J \times \dot{X} \]

- **Gradients:**
  - When \( n = 1 \) output: gradient = \( J = \left( \frac{\partial y}{\partial x_i} \right) \)
  - When \( n > 1 \) outputs: gradient = \( \bar{Y}^t \times J \)

- **Higher-order derivative tensors**

- **Taylor coefficients**

- **Intervals?**
Divided Differences

Given $\dot{X}$, run P twice, and compute $\dot{Y}$

$$\dot{Y} = \frac{P(X + \varepsilon \dot{X}) - P(X)}{\varepsilon}$$

- Pros: immediate; no thinking required!
- Cons: approximation; what $\varepsilon$?
  $\Rightarrow$ Not so cheap after all!

Optimization wants inexpensive and accurate derivatives.
$\Rightarrow$ Let’s go for exact, analytic derivatives!
AD Example: analytic tangent differentiation by Program transformation

```
SUBROUTINE FOO(v1, v2, v4, p1)

REAL v1, v2, v3, v4, p1

v3 = 2.0*v1 + 5.0

v4 = v3 + p1*v2/v3

END
```

Just inserts "differentiated instructions" into FOO
AD Example: analytic tangent differentiation by Program transformation

SUBROUTINE FOO(v1, v2, v4, p1)

REAL v1,v2,v3,v4,p1

v3d = 2.0*v1d
v3 = 2.0*v1 + 5.0
v4d = v3d + p1*(v2d*v3-v2*v3d)/(v3*v3)
v4 = v3 + p1*v2/v3

END
SUBROUTINE FOO(v1, v1d, v2, v2d, v4, v4d, p1)
    REAL v1d, v2d, v3d, v4d
    REAL v1, v2, v3, v4, p1

    v3d = 2.0*v1d
    v3 = 2.0*v1 + 5.0
    v4d = v3d + p1*(v2d*v3-v2*v3d)/(v3*v3)
    v4 = v3 + p1*v2/v3
END

Just inserts “differentiated instructions” into FOO.
1 Introduction
2 Formalization
3 .......... Multi-directional
4 Reverse AD
5 .......... Alternative formalizations
6 Reverse AD for Optimization
7 Performance issues
8 .......... Static Analyses in AD tools
9 Some AD Tools
10 Validation
11 .......... Expert-level AD
12 Conclusion
Take control away!

We differentiate programs. But control $\Rightarrow$ non-differentiability!

Freeze the current control:
$\Rightarrow$ the program becomes a simple sequence of instructions
$\Rightarrow$ AD differentiates these sequences:

Caution: the program is only piecewise differentiable!
Computer Programs as Functions

- Identify sequences of instructions
  \[\{ l_1; l_2; \ldots l_{p-1}; l_p; \}\]
  with composition of functions.
- Each simple instruction
  \[l_k : v_4 = v_3 + v_2/v_3\]
  is a function \(f_k : \mathbb{R}^q \to \mathbb{R}^q\) where
  - The output \(v_4\) is built from the input \(v_2\) and \(v_3\)
  - All other variable are passed unchanged
- Thus we see \(P : \{ l_1; l_2; \ldots l_{p-1}; l_p; \}\) as
  \[f = f_p \circ f_{p-1} \circ \cdots \circ f_1\]
Using the Chain Rule

\[ f = f_p \circ f_{p-1} \circ \cdots \circ f_1 \]

We define for short:

\[ W_0 = X \quad \text{and} \quad W_k = f_k(W_{k-1}) \]

The chain rule yields:

\[ f'(X) = f'_p(W_{p-1}) \cdot f'_{p-1}(W_{p-2}) \cdots \cdot f'_1(W_0) \]
Tangent mode and Reverse mode

Full J is expensive and often useless. We’d better compute useful projections of J.

\[
\begin{align*}
\dot{Y} &= f'(X) \cdot \dot{X} = f'_p(W_{p-1}) \cdot f'_{p-1}(W_{p-2}) \cdots f'_1(W_0) \cdot \dot{X} \\
\text{reverse AD :} & \quad \bar{X} = f'^t(X) \cdot \bar{Y} = f'^t_1(W_0) \cdots f'^t_{p-1}(W_{p-2}) \cdot f'^t_p(W_{p-1}) \cdot \bar{Y}
\end{align*}
\]

Evaluate both from right to left: \(\Rightarrow\) always matrix \(\times\) vector

Theoretical cost is about 4 times the cost of P
Costs of Tangent and Reverse AD

\[ F : \mathbb{R}^m \rightarrow \mathbb{R}^n \]

- \( J \) costs \( m \times 4 \times P \) using the tangent mode
  Good if \( m \leq n \)
- \( J \) costs \( n \times 4 \times P \) using the reverse mode
  Good if \( m \gg n \) (e.g. \( n = 1 \) in optimization)
Back to the Tangent Mode example

\[ v3 = 2.0 \times v1 + 5.0 \]
\[ v4 = v3 + p1 \times v2/v3 \]

Elementary Jacobian matrices:

\[ f'(X) = \ldots \begin{pmatrix} 1 & 1 & \frac{p1}{v3} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{p1}{v3} & 0 & 1 & 0 \\ 0 & \frac{1}{v3} & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 2 & 0 \\ 0 & 1 \end{pmatrix} \ldots \]

\[ \dot{v3} = 2 \times \dot{v1} \]
\[ \dot{v4} = \dot{v3} \times \left(1 - p1 \times \frac{v2}{v3^2}\right) + \dot{v2} \times \frac{p1}{v3} \]
Tangent Mode example continued

Tangent AD keeps the structure of $P$:

\[
\begin{align*}
&v_{3d} = 2.0 \cdot v_{1d} \\
v_3 &= 2.0 \cdot v_1 + 5.0 \\
v_{4d} &= v_{3d} \cdot (1 - p_1 \cdot v_2 / (v_3 \cdot v_3)) + v_{2d} \cdot p_1 / v_3 \\
v_4 &= v_3 + p_1 \cdot v_2 / v_3
\end{align*}
\]

Differentiated instructions inserted into $P$’s original control flow.
Outline

1. Introduction
2. Formalization
3. Multi-directional
4. Reverse AD
5. Alternative formalizations
6. Reverse AD for Optimization
7. Performance issues
8. Static Analyses in AD tools
9. Some AD Tools
10. Validation
11. Expert-level AD
12. Conclusion

Laurent Hascoët (INRIA) AD  Trieste, april 18th, 2005 15 / 68
If you want $\dot{Y} = f'(X).\dot{X}$ for the same $X$ and several $\dot{X}$

- either run the tangent differentiated program several times, evaluating $f$ several times.
- or run a “Multi-directional” tangent once, evaluating $f$ once.

Same for $\overline{X} = f''t(X).\overline{Y}$ for several $\overline{Y}$.

In particular, multi-directional tangent or reverse is good to get the full Jacobian.
Sparse Jacobians with seed matrices

When sparse Jacobian, use “seed matrices” to propagate fewer \( \dot{X} \) or \( \dot{Y} \)

- **Multi-directional tangent mode:**

\[
\begin{pmatrix}
a & b \\
c & d \\
e & f & g
\end{pmatrix}
\times
\begin{pmatrix}
1 & 1 \\
1 & 1
\end{pmatrix}
= 
\begin{pmatrix}
a & b \\
c & d \\
e & f & g
\end{pmatrix}
\]

- **Multi-directional reverse mode:**

\[
\begin{pmatrix}
1 & 1 & 1 \\
1 & 1 & 1
\end{pmatrix}
\times
\begin{pmatrix}
a & b \\
c & d \\
e & f & g
\end{pmatrix}
= 
\begin{pmatrix}
a & c & b \\
e & f & d & g
\end{pmatrix}
\]
Outline

1. Introduction
2. Formalization
   ....... Multi-directional
4. Reverse AD
   ....... Alternative formalizations
5. Reverse AD for Optimization
6. Performance issues
8. ....... Static Analyses in AD tools
9. Some AD Tools
10. Validation
11. ....... Expert-level AD
12. Conclusion
Focus on the Reverse mode

\[ \overline{X} = f'^t(X), \overline{Y} = f'^t(W_0) \cdots f'^t(W_{p-1}). \overline{Y} \]

\[
\begin{align*}
I_{p-1} & ; \\
\overline{W} & = \overline{Y} ; \\
\overline{W} & = f'^t(W_{p-1}) \ast \overline{W} ;
\end{align*}
\]
Focus on the Reverse mode

\[ \bar{X} = f'_{\text{R}}(X). \bar{Y} = f'_1(W_0) \ldots f'_p(W_{p-1}). \bar{Y} \]

\[ I_{p-2}; \]
\[ I_{p-1}; \]
\[ \bar{W} = \bar{Y}; \]
\[ \bar{W} = f'_{\text{R}}(W_{p-1}) \ast \bar{W}; \]
\[ \text{Restore } W_{p-2} \text{ before } I_{p-2}; \]
\[ \bar{W} = f'_{p-1}(W_{p-2}) \ast \bar{W}; \]
Focus on the Reverse mode

\[ \bar{X} = f'^t(X). \bar{Y} = f'_1^t(W_0) \ldots f'_p^t(W_{p-1}). \bar{Y} \]

\[ \begin{align*}
I_1 ; \\
\ldots \\
I_{p-2} ; \\
I_{p-1} ; \\
\bar{W} = \bar{Y} ; \\
\bar{W} = f'_p^t(W_{p-1}) * \bar{W} ; \\
\text{Restore } W_{p-2} \text{ before } I_{p-2} ; \\
\bar{W} = f'_p^t(W_{p-2}) * \bar{W} ; \\
\ldots \\
\text{Restore } W_0 \text{ before } I_1 ; \\
\bar{W} = f'_1^t(W_0) * \bar{W} ; \\
\bar{X} = \bar{W} ;
\end{align*} \]

Instructions differentiated in the reverse order!
Reverse mode: graphical interpretation

Bottleneck: memory usage ("Tape").
Back to the example

\[ v_3 = 2.0 \ast v_1 + 5.0 \]
\[ v_4 = v_3 + p_1 \ast v_2 / v_3 \]

Transposed Jacobian matrices:

\[ f'^t(X) = \cdots \begin{pmatrix} 1 & 2 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 - \frac{p_1}{v_3} \frac{v_1 \ast v_2}{v_3^2} \end{pmatrix} \cdots \]

\[ \bar{v}_2 = \bar{v}_2 + \bar{v}_4 \ast \frac{p_1}{v_3} \]
\[ \cdots \]

\[ \bar{v}_1 = \bar{v}_1 + 2 \ast \bar{v}_3 \]
\[ \bar{v}_3 = 0 \]
Reverse AD inverses the structure of $P$:

\[
\begin{align*}
v3 &= 2.0 \times v1 + 5.0 \\
v4 &= v3 + p1 \times v2/v3 \\
\end{align*}
\]

\[
\begin{align*}
\ldots
\end{align*}
\]

\[
\begin{align*}
\ldots
\end{align*}
\]

\[
\begin{align*}
v2b &= v2b + p1 \times v4b/v3 \\
v3b &= v3b + (1-p1 \times v2/(v3 \times v3)) \times v4b \\
v4b &= 0.0 \\
\ldots
\end{align*}
\]

\[
\begin{align*}
\ldots
\end{align*}
\]

\[
\begin{align*}
v1b &= v1b + 2.0 \times v3b \\
v3b &= 0.0 \\
\ldots
\end{align*}
\]

Differentiated instructions inserted into the inverse of $P$'s original control flow.
The control flow of the *forward sweep* is mirrored in the *backward sweep*.

...  
```
if (T(i).lt.0.0) then
    T(i) = S(i)*T(i)
endif
```

...  
```
if (...) then
    Sb(i) = Sb(i) + T(i)*Tb(i)
    Tb(i) = S(i)*Tb(i)
endif
```

...
Reversed loops run in the inverse order

...  
Do i = 1,N  
   T(i) = 2.5*T(i-1) + 3.5  
Enddo

...

Do i = N,1,-1  
   Tb(i-1) = Tb(i-1) + 2.5*Tb(i)  
   Tb(i) = 0.0  
Enddo
Control Flow Inversion: spaghetti

Remember original Control Flow when it merges
Outline

1. Introduction
2. Formalization
   3. "Multi-directional"
4. Reverse AD
5. "Alternative formalizations"
6. Reverse AD for Optimization
7. Performance issues
   8. "Static Analyses in AD tools"
9. Some AD Tools
10. Validation
   11. "Expert-level AD"
12. Conclusion
Yet another formalization using computation graphs

A sequence of instructions corresponds to a computation graph

\[
\begin{align*}
\text{DO } i &= 1, n \\
\text{IF } (B(i).\text{gt.0.0}) \text{ THEN} \\
& \quad r = A(i) \cdot B(i) + y \\
& \quad X(i) = 3 \cdot r - B(i) \cdot X(i-3) \\
& \quad y = \sin(X(i) \cdot r) \\
\text{ENDIF} \\
\text{ENDDO}
\end{align*}
\]

Source program

\[
\text{Computation Graph}
\]

Laurent Hascoët (INRIA)
Trieste, April 18th, 2005
Jacobians by Vertex Elimination

- Forward vertex elimination ⇒ tangent AD.
- Reverse vertex elimination ⇒ reverse AD.
- Other orders ("cross-country") may be optimal.
Yet another formalization: Lagrange multipliers

\[ v3 = 2.0 * v1 + 5.0 \]
\[ v4 = v3 + p1 * v2 / v3 \]

Can be viewed as constrains. We know that the Lagrangian \( \mathcal{L}(v_1, v_2, v_3, v_4, v_3, v_4) = v_4 + v_3.(-v_3 + 2.0* v_1 + 5) + v_4.(-v_4 + v_3 + p_1 * v_2 / v_3) \) is such that:

\[ \bar{v}_1 = \frac{\partial v_4}{\partial v_1} = \frac{\partial \mathcal{L}}{\partial v_1} \quad \text{and} \quad \bar{v}_2 = \frac{\partial v_4}{\partial v_2} = \frac{\partial \mathcal{L}}{\partial v_2} \]

provided

\[ \frac{\partial \mathcal{L}}{\partial v_3} = \frac{\partial \mathcal{L}}{\partial v_4} = \frac{\partial \mathcal{L}}{\partial v_3} = \frac{\partial \mathcal{L}}{\partial v_4} = 0 \]
The $\bar{v}_i$ are the Lagrange multipliers associated to the instruction that sets $v_i$.

For instance, equation $\frac{\partial L}{\partial v_3} = 0$ gives us:

$$\bar{v}_4.(1 - p_1.v_2/(v_3.v_3)) - \bar{v}_3 = 0$$

To be compared with instruction

$$v_{3b} = v_{3b} + (1-p_1*v_2/(v_3*v_3))*v_{4b}$$

(initial $v_{3b}$ is set to 0.0)
Applications to Optimization

From a simulation program \( P : \)

\[ P : (\text{design parameters})\gamma \mapsto (\text{cost function})J(\gamma) \]

it takes a gradient \( J'(\gamma) \) to obtain an optimization program.

Reverse mode AD builds program \( \overline{P} \) that computes \( J'(\gamma) \).

Optimization algorithms (Gradient descent, SQP, . . . ) may also use 2nd derivatives. AD can provide them too.
Special case: steady-state

If $J$ is defined on a state $W$, and $W$ results from an implicit steady state equation

$$\psi(W, \gamma) = 0$$

which is solved iteratively: $W_0, W_1, W_2, \ldots, W_\infty$

then pure reverse AD of $P$ may prove too expensive (memory...)

Solutions exist:

- reverse AD on the final steady state only.
- Andreas Griewank’s “Piggy-backing”
- reverse AD on $\psi$ alone + hand-coding
A color picture *(at last !...)*

AD-computed gradient of a scalar cost (sonic boom) with respect to skin geometry:
... and after a few optimization steps

Improvement of the sonic boom under the plane after 8 optimization cycles:

(Plane geometry provided by Dassault Aviation)
Influence of $T$ at -300 metres on heat flux 20 days later across North section.

- Kelvin wave
- Rossby wave

15° North
Outline

1. Introduction
2. Formalization
3. Multi-directional
4. Reverse AD
5. Alternative formalizations
6. Reverse AD for Optimization
7. Performance issues
8. Static Analyses in AD tools
9. Some AD Tools
10. Validation
11. Expert-level AD
12. Conclusion
From the definition of the gradient $\overline{X}$

$$\overline{X} = f'^t(X). \overline{Y} = f'_1(W_0) \ldots f'_p(W_{p-1}) \overline{Y}$$

we get the general shape of reverse AD program:

$⇒$ How can we restore previous values?
Restoration by recomputation
(RA: Recompute-All)

Restart execution from a stored initial state:

Memory use low, CPU use high $\Rightarrow$ trade-off needed!
Progressively undo the assignments made by the forward sweep

Memory use high, CPU use low $\Rightarrow$ trade-off needed!
Checkpointing (SA strategy)

On selected pieces of the program, possibly nested, don’t store intermediate values and re-execute the piece when values are required.

Memory and CPU grow like $\log(size(P))$
Checkpointing on calls (SA)

A classical choice: checkpoint procedure calls!

Memory and CPU grow like $\log(\text{size}(P))$ when call tree well balanced.

Ill-balanced call trees require not checkpointing some calls.

Careful analysis keeps the snapshots small.
Outline

1 Introduction
2 Formalization
3 .......... Multi-directional
4 Reverse AD
5 .......... Alternative formalizations
6 Reverse AD for Optimization
7 Performance issues
8 .......... Static Analyses in AD tools
9 Some AD Tools
10 Validation
11 .......... Expert-level AD
12 Conclusion

Laurent Hascoët (INRIA)
Activity analysis

Finds out the variables that, at some location
- do not depend on any independent,
- or have no dependent depending on them.
Derivative either null or useless $\Rightarrow$ simplifications

<table>
<thead>
<tr>
<th>orig. prog</th>
<th>tangent mode</th>
<th>w/activity analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c = a*b$</td>
<td>$cd = a<em>bd + ad</em>b$</td>
<td>$cd = a<em>bd + ad</em>b$</td>
</tr>
<tr>
<td>$a = 5.0$</td>
<td>$c = a*b$</td>
<td>$c = a*b$</td>
</tr>
<tr>
<td></td>
<td>$ad = 0.0$</td>
<td>$ad = 0.0$</td>
</tr>
<tr>
<td>$d = a*c$</td>
<td>$a = 5.0$</td>
<td>$a = 5.0$</td>
</tr>
<tr>
<td></td>
<td>$dd = a<em>cd + ad</em>c$</td>
<td>$dd = a*cd$</td>
</tr>
<tr>
<td>$e = a/c$</td>
<td>$d = a*c$</td>
<td>$d = a*c$</td>
</tr>
<tr>
<td></td>
<td>$ed = ad/c - a*cd/c**2$</td>
<td>$ed = 0.0$</td>
</tr>
<tr>
<td>e=floor(e)</td>
<td>$e = a/c$</td>
<td>$e = a/c$</td>
</tr>
<tr>
<td></td>
<td>$ed = 0.0$</td>
<td>$ed = 0.0$</td>
</tr>
<tr>
<td></td>
<td>$e = floor(e)$</td>
<td>$e = floor(e)$</td>
</tr>
</tbody>
</table>
The important result of the reverse mode is in $\overline{X}$. The original result $Y$ is of no interest.

- The last instruction of the program $P$ can be removed from $\overline{P}$.
- Recursively, other instructions of $P$ can be removed too.
<table>
<thead>
<tr>
<th>orig. program</th>
<th>reverse mode</th>
<th>Adjoint Live code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF(a.GT.0.)THEN</td>
<td>IF(a.GT.0.)THEN</td>
<td>IF (a.GT.0.) THEN</td>
</tr>
<tr>
<td></td>
<td>a = LOG(a)</td>
<td>ab = ab/a</td>
</tr>
<tr>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>a = LOG(c)</td>
<td>a = LOG(c)</td>
<td>a = LOG(c)</td>
</tr>
<tr>
<td>CALL SUB(a)</td>
<td>CALL SUB_B(a,ab)</td>
<td>CALL SUB_B(a,ab)</td>
</tr>
<tr>
<td>ENDIF</td>
<td>cb = cb + ab/c</td>
<td>cb = cb + ab/c</td>
</tr>
<tr>
<td>END</td>
<td>ab = 0.0</td>
<td>ab = 0.0</td>
</tr>
<tr>
<td></td>
<td>END IF</td>
<td>END IF</td>
</tr>
</tbody>
</table>

Laurent Hascoêt (INRIA)
“To Be Restored” analysis

In reverse AD, not all values must be restored during the backward sweep.

Variables occurring only in linear expressions do not appear in the differentiated instructions.
⇒ not To Be Restored.
\[
\begin{align*}
x &= x + \text{EXP}(a) \\
y &= x + a^{*2} \\
a &= 3*z
\end{align*}
\]

<table>
<thead>
<tr>
<th>reverse mode: naive backward sweep</th>
<th>reverse mode: backward sweep with TBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL POP(a)</td>
<td>CALL POP(a)</td>
</tr>
<tr>
<td>zb = zb + 3*ab</td>
<td>zb = zb + 3*ab</td>
</tr>
<tr>
<td>ab = 0.0</td>
<td>ab = 0.0</td>
</tr>
<tr>
<td>CALL POP(y)</td>
<td>ab = ab + 2<em>a</em>yb</td>
</tr>
<tr>
<td>ab = ab + 2<em>a</em>yb</td>
<td>xb = xb + yb</td>
</tr>
<tr>
<td>xb = xb + yb</td>
<td>yb = 0.0</td>
</tr>
<tr>
<td>yb = 0.0</td>
<td>ab = ab + \text{EXP}(a)*xb</td>
</tr>
<tr>
<td>CALL POP(x)</td>
<td></td>
</tr>
<tr>
<td>ab = ab + \text{EXP}(a)*xb</td>
<td></td>
</tr>
</tbody>
</table>
In reverse AD, it is important to know whether two variables in an instruction are the same.

<table>
<thead>
<tr>
<th>(a[i] = 3*a[i+1])</th>
<th>(a[i] = 3*a[i])</th>
<th>(a[i] = 3*a[j])</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables certainly different</td>
<td>variables certainly equal</td>
<td>? ⇒</td>
</tr>
</tbody>
</table>
| \(ab[i+1]= ab[i+1]+ 3*ab[i]\) | \(ab[i] = 3* ab[i]\) | \(tmpb = ab[i] + 3*tmpb\)
| \(ab[i] = 0.0\) | \(ab[i] = 0.0\) | \(ab[j] = ab[j]\) |
|                       | \(a[i] = tmp\) | \(a[i] = tmp\) |

\[\text{tmp} = 3*a[j]\]
\[a[i] = \text{tmp}\]
\[ab[j] = ab[j]\]
\[+ 3*tmpb\]
Taking small snapshots saves a lot of memory:

\[ \text{Snapshot}(C) = \text{Use}(\overline{C}) \cap (\text{Write}(C) \cup \text{Write}(\overline{D})) \]
Undecidability

- Analyses are static: operate on source, don’t know run-time data.
- Undecidability: no static analysis can answer yes or no for every possible program: there will always be programs on which the analysis will answer “I can’t tell”
- ⇒ all tools must be ready to take conservative decisions when the analysis is in doubt.
- In practice, tool “laziness” is a far more common cause for undecided analyses and conservative transformations.
Outline

1. Introduction
2. Formalization
   - Multi-directional
3. Reverse AD
4. Reverse AD for Optimization
5. Alternative formalizations
6. Performance issues
7. Static Analyses in AD tools
8. Some AD Tools
9. Validation
   - Expert-level AD
10. Conclusion
Tools for source-transformation AD

AD tools are based on overloading or on source transformation.

Source transformation requires complex tools, but offers more room for optimization.

<table>
<thead>
<tr>
<th>parsing</th>
<th>→analysis</th>
<th>→differentiation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F77</td>
<td>type-checking</td>
<td>tangent</td>
</tr>
<tr>
<td>F9X</td>
<td>use/kill</td>
<td>reverse</td>
</tr>
<tr>
<td>C</td>
<td>dependencies</td>
<td>multi-directional</td>
</tr>
<tr>
<td>MATLAB</td>
<td>activity</td>
<td>. . .</td>
</tr>
<tr>
<td>. . .</td>
<td>. . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>
Some AD tools

- **NAGWARE F95** Compiler: Overloading, tangent, reverse
- **ADOL-C**: Overloading + Tape; tangent, reverse, higher-order
- **ADIFOR/Open-AD**: Regeneration; tangent, reverse?, Store-All + Checkpointing
- **TAPENADE**: Regeneration; tangent, reverse, Store-All + Checkpointing
- **TAF**: Regeneration; tangent, reverse, Recompute-All + Checkpointing
Some Limitations of AD tools

Fundamental problems:
- Piecewise differentiability
- Convergence of derivatives
- Reverse AD of large codes

Technical Difficulties:
- Pointers and memory allocation
- Objects
- Inversion or Duplication of random control (communications, random,...)
Validation methods

From a program $P$ that evaluates

$$F : \mathbb{R}^m \rightarrow \mathbb{R}^n$$

$X \mapsto Y$

tangent AD creates

$$\dot{P} : X, \dot{X} \mapsto Y, \dot{Y}$$

and reverse AD creates

$$\overline{P} : X, \overline{Y} \mapsto \overline{X}$$

Wow can we validate these programs?
- Tangent wrt Divided Differences
- Reverse wrt Tangent
Validation of Tangent \textit{wrt} Divided Differences

For a given $\dot{X}$, set $g(h \in \mathbb{R}) = F(X + h.Xd)$:

$$g'(0) = \lim_{\varepsilon \to 0} \frac{F(X + \varepsilon \times \dot{X}) - F(X)}{\varepsilon}$$

Also, from the chain rule:

$$g'(0) = F'(X) \times \dot{X} = \dot{Y}$$

So we can approximate $\dot{Y}$ by running $P$ twice, at points $X$ and $X + \varepsilon \times \dot{X}$
Validation of Reverse wrt Tangent

For a given $\dot{X}$, tangent code returned $\dot{Y}$

Initialize $\overline{Y} = \dot{Y}$ and run the reverse code, yielding $\overline{X}$. We have:

$$
(\overline{X} \cdot \dot{X}) = (F'^t(X) \times \dot{Y} \cdot \dot{X})
$$

$$
= \dot{Y}^t \times F'(X) \times \dot{X}
$$

$$
= \dot{Y}^t \times \dot{Y}
$$

$$
= (\dot{Y} \cdot \dot{Y})
$$

Often called the “dot-product test”
Black-box routines

If the tool permits, give dependency signature (sparsity pattern) of all external procedures $\Rightarrow$ better activity analysis $\Rightarrow$ better diff program.

After AD, provide required hand-coded derivative (FOO_D or FOO_B)
Make linear or auto-adjoint procedures “black-box”.

Since they are their own tangent or reverse derivatives, provide their original form as hand-coded derivative.

In many cases, this is more efficient than pure AD of these procedures.
Independent loops

If a loop has independent iterations, possibly terminated by a sum-reduction, then

\[
\begin{align*}
\text{Standard:} & \quad \text{Improved:} \\
\text{doi} &= 1,N \\
body(i) & \quad body(i) \\
end & \quad \leftarrow body(i) \\
\text{doi} &= N,1 \\
end & \quad \text{end}
\end{align*}
\]

In the Recompute-All context, this reduces the memory consumption by a factor $N$.
Outline

1. Introduction
2. Formalization
3. ...... Multi-directional
4. Reverse AD
5. ...... Alternative formalizations
6. Reverse AD for Optimization
7. Performance issues
8. ...... Static Analyses in AD tools
9. Some AD Tools
10. Validation
11. ...... Expert-level AD
12. Conclusion

Laurent Hascoët (INRIA) AD Trieste, april 18th, 2005 64 / 68
AD: Context

DERIVATIVES

Div. Diff
Analytic Diff
Maths
Overloading
Source Transfo
Multi-dir
Tangent
Reverse

inaccuracy
control
programming
flexibility

Laurent Hascoët (INRIA)
Trieste, april 18th, 2005
65 / 68
If you want the derivatives of an implemented math function, you should seriously consider AD.

Divided Differences aren’t good for you (nor for others…)

Especially think of AD when you need higher order (taylor coefficients) for simulation or gradients (reverse mode) for optimization.

Reverse AD is a discrete equivalent of the adjoint methods from control theory: gives a gradient at remarkably low cost.
AD tools provide you with highly optimized derivative programs in a matter of minutes.

AD tools are making progress steadily, but the best AD will always require end-user intervention.
Thank you for your attention!

1. Introduction
2. Formalization
3. .......... Multi-directional
4. Reverse AD
5. .......... Alternative formalizations
6. Reverse AD for Optimization
7. Performance issues
8. .......... Static Analyses in AD tools
9. Some AD Tools
10. Validation
11. .......... Expert-level AD
12. Conclusion