Derivative Evaluation by Automatic Differentiation of Programs

Laurent Hascoët
Laurent.Hascoet@sophia.inria.fr
http://www-sop.inria.fr/tropics

Ecole d'été CEA-EDF-INRIA,
Juillet 2005

Outline

1. Introduction
2. Formalization
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. Some AD Tools
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

So you need derivatives ?...

Given a program P computing a function F

$$\begin{array}{rcccc} F & : & \boldsymbol{R}^m & \longrightarrow & \boldsymbol{R}^n \\ & & X & \longmapsto & Y \end{array}$$

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 $= \overline{Y}^t \times J$
- Higher-order derivative tensors
- Taylor coefficients
- Intervals ?

Divided Differences

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

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

- Pros: immediate; no thinking required !
- Cons: approximation; what ε ?

  $\Rightarrow$ Not so cheap after all !

Most applications require inexpensive and accurate derivatives.

$\Rightarrow$ Let's go for exact, analytic derivatives !

Automatic Differentiation

Augment program P to make it compute the analytic derivatives

```
P: a = b*T(10) + c
```

The differentiated program must somehow compute:

```
P': da = db*T(10) + b*dT(10) + dc
```

How can we achieve this?

- AD by Overloading
- AD by Program transformation

AD by overloading

Tools: ADOL-C, ADTAGEO,...

Few manipulations required:

- `DOUBLE` $\longrightarrow$ `ADOUBLE` ;
- link with provided overloaded `+`,`-`,`*`,...

Easy extension to higher-order, Taylor series, intervals,

...but not so easy for gradients.

Anecdote?:

- real $\longrightarrow$ complex
- `x = a*b` $\longrightarrow$

  `(x , dx) = (a*b-da*db , a*db+da*b)`

AD by Program transformation

Tools: ADIFOR, TAF, TAPENADE,...

Complex transformation required:

- Build a new program that computes the analytic derivatives explicitly.
- Requires a compiler-like, sophisticated tool
  1. PARSING,
  2. ANALYSIS,
  3. DIFFERENTIATION,
  4. REGENERATION

Overloading *vs* Transformation

Overloading is versatile,

Transformed programs are efficient:

- Global program analyses are possible and most welcome !
- The compiler can optimize the generated program.

Example: 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
```

Example: 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
```

Example: Tangent differentiation by Program transformation

```
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

Outline

1 Introduction
2 Formalization
3 Reverse AD
4 Alternative formalizations
5 Memory issues in Reverse AD: Checkpointing
6 Multi-directional
7 Reverse AD for Optimization
8 AD for Sensitivity to Uncertainties
9 Some AD Tools
10 Static Analyses in AD tools
11 The TAPENADE AD tool
12 Validation of AD results
13 Expert-level AD
14 Conclusion

Dealing with the Programs' Control

Programs contain control:

discrete ⇒ non-differentiable.

```
if (x <= 1.0) then
   printf("x too small");
else {
   y = 1.0;
   while (y <= 10.0) {
      y = y*x;
      x = x+0.5;
   }
}
```

Not differentiable for `x=1.0`

Not differentiable for `x=2.9221444`

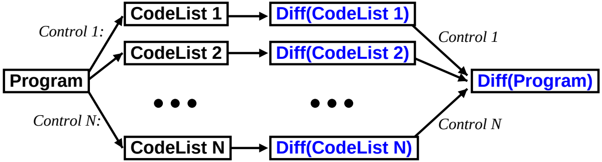
Take control away!

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

Freeze the current control:
For one given control, the program becomes a simple list of instructions $\Rightarrow$ differentiable:

```
printf("x too small");
y = 1.0; y = y*x; x = x+0.5;
```

AD differentiates these lists of instructions:



Caution: the program is only piecewise differentiable !

Computer Programs as Functions

- Identify sequences of instructions

$$\{I_1; I_2; \ldots I_{p-1}; I_p; \}$$

  with composition of functions.
- Each simple instruction

$$I_k : \quad \texttt{v4 = v3 + v2/v3}$$

  is a function $f_k : \boldsymbol{R}^q \longrightarrow \boldsymbol{R}^q$ where
  - The output `v4` is built from the input `v2` and `v3`
  - All other variable are passed unchanged
- Thus we see `P` : $\{I_1; I_2; \ldots I_{p-1}; I_p; \}$ as

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

Using the Chain Rule

We see program P as:

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

We define for short:

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

The chain rule yields:

$$f'(X) = f'_p(W_{p-1}).f'_{p-1}(W_{p-2}).\ldots.f'_1(W_0)$$

The Jacobian Program

$$f'(X) = f'_p(W_{p-1}).f'_{p-1}(W_{p-2}).\ldots.f'_1(W_0)$$

translates immediately into a program that computes the Jacobian J:

$$
\begin{array}{ll}
I_1 \;; & /*\; W = f_1(W) \;*/ \\
I_2 \;; & /*\; W = f_2(W) \;*/ \\
\ldots & \\
I_p \;; & /*\; W = f_p(W) \;*/
\end{array}
$$

The Jacobian Program

$$f'(X) = f'_p(W_{p-1}).f'_{p-1}(W_{p-2}).\ldots.f'_1(W_0)$$

translates immediately into a program that computes the Jacobian J:

$$
\begin{array}{ll}
W = X \; ; & \\
J = f'_1(W) \; ; & \\
I_1 \; ; & /*\ W = f_1(W)\ */ \\
J = f'_2(W) * J \; ; & \\
I_2 \; ; & /*\ W = f_2(W)\ */ \\
\ldots & \\
J = f'_p(W) * J \; ; & \\
I_p \; ; & /*\ W = f_p(W)\ */ \\
Y = W \; ; &
\end{array}
$$

Tangent mode and Reverse mode

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

tangent AD :

$$\dot{Y} = f'(X).\dot{X} = f'_p(W_{p-1}).f'_{p-1}(W_{p-2}) \ldots f'_1(W_0).\dot{X}$$

reverse AD :

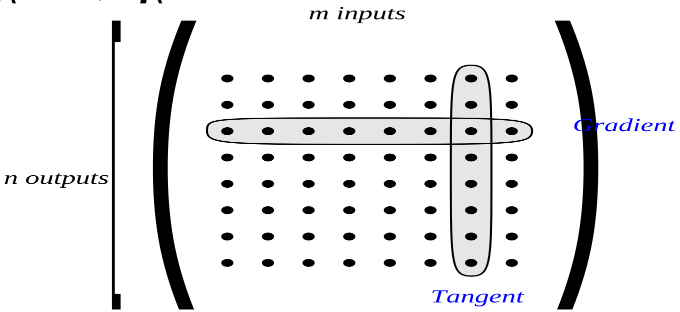
$$\overline{X} = f'^t(X).\overline{Y} = f'^t_1(W_0). \ldots f'^t_{p-1}(W_{p-2}).f'^t_p(W_{p-1}).\overline{Y}$$

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

$F \quad : \quad \boldsymbol{R}^m \quad \rightarrow \quad \boldsymbol{R}^n$



- J costs $m * 4 * \mathrm{P}$ using the tangent mode
  Good if $m <= n$
- J costs $n * 4 * \mathrm{P}$ using the reverse mode
  Good if $m >> n$ (e.g $n = 1$ in optimization)

Back to the Tangent Mode example

```
v3 = 2.0*v1 + 5.0
v4 = v3 + p1*v2/v3
```

Elementary Jacobian matrices:

$$f'(X) = \ldots \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ 0 & \frac{p_1}{v_3} & 1 - \frac{p_1 * v_2}{v_3^2} & 0 \end{pmatrix} \begin{pmatrix} 1 & & & \\ & 1 & & \\ 2 & & 0 & \\ & & & 1 \end{pmatrix} \ldots$$

$$\begin{aligned} \dot{v}_3 &= 2 * \dot{v}_1 \\ \dot{v}_4 &= \dot{v}_3 * (1 - p_1 * v_2/v_3^2) + \dot{v}_2 * p_1/v_3 \end{aligned}$$

Tangent Mode example continued

Tangent AD keeps the structure of P:

```
          ...
v3d = 2.0*v1d
v3 = 2.0*v1 + 5.0
v4d = v3d*(1-p1*v2/(v3*v3)) + v2d*p1/v3
v4 = v3 + p1*v2/v3

          ...
```

Differentiated instructions inserted into P's original control flow.

Outline

1 Introduction
2 Formalization
3 Reverse AD
4 Alternative formalizations
5 Memory issues in Reverse AD: Checkpointing
6 Multi-directional
7 Reverse AD for Optimization
8 AD for Sensitivity to Uncertainties
9 Some AD Tools
10 Static Analyses in AD tools
11 The TAPENADE AD tool
12 Validation of AD results
13 Expert-level AD
14 Conclusion

Focus on the Reverse mode

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0)\ldots f_p'^t(W_{p-1}).\overline{Y}$$

I_{p-1} ;
$\overline{W} = \overline{Y}$;
$\overline{W} = f_p'^t(W_{p-1}) * \overline{W}$;

Focus on the Reverse mode

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0)\dots f_p'^t(W_{p-1}).\overline{Y}$$

I_{p-2} ;
I_{p-1} ;
$\overline{W} = \overline{Y}$;
$\overline{W} = f_p'^t(W_{p-1}) * \overline{W}$;
Restore W_{p-2} before I_{p-2} ;
$\overline{W} = f_{p-1}'^t(W_{p-2}) * \overline{W}$;

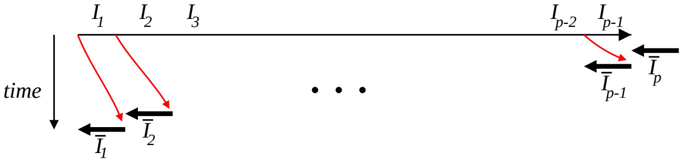
Focus on the Reverse mode

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0) \dots f_p'^t(W_{p-1}).\overline{Y}$$

$$\begin{array}{l}
I_1 \;\; ; \\
\dots \\
I_{p-2} \;\; ; \\
I_{p-1} \;\; ; \\
\overline{W} \; = \; \overline{Y} \;\; ; \\
\overline{W} \; = \; f_p'^t(W_{p-1}) \; * \; \overline{W} \;\; ; \\
\mathit{Restore}\ W_{p-2}\ \mathit{before}\ I_{p-2} \; ; \\
\overline{W} \; = \; f_{p-1}'^t(W_{p-2}) \; * \; \overline{W} \;\; ; \\
\dots \\
\mathit{Restore}\ W_0\ \mathit{before}\ I_1 \; ; \\
\overline{W} \; = \; f_1'^t(W_0) \; * \; \overline{W} \;\; ; \\
\overline{X} \; = \; \overline{W} \;\; ;
\end{array}$$

Instructions differentiated in the reverse order !

Reverse mode: graphical interpretation



Bottleneck: memory usage ("Tape").

```
v3 = 2.0*v1 + 5.0
v4 = v3 + p1*v2/v3
```

Transposed Jacobian matrices:

$$f'^t(X) = \ldots \begin{pmatrix} 1 & & 2 & \\ & 1 & & \\ & & 0 & \\ & & & 1 \end{pmatrix} \begin{pmatrix} 1 & & & 0 \\ & 1 & & \frac{p_1}{v_3} \\ & & 1 & 1 - \frac{p_1 * v_2}{v_3^2} \\ & & & 0 \end{pmatrix} \ldots$$

$$\begin{aligned} \overline{v}_2 &= \overline{v}_2 + \overline{v}_4 * p_1 / v_3 \\ &\ldots \\ \overline{v}_1 &= \overline{v}_1 + 2 * \overline{v}_3 \\ \overline{v}_3 &= 0 \end{aligned}$$

Reverse Mode example continued

Reverse AD inverses the structure of P:

```
          ...
 v3 = 2.0*v1 + 5.0
 v4 = v3 + p1*v2/v3
          ...
          ...
........................../*restore previous state*/
 v2b = v2b + p1*v4b/v3
 v3b = v3b + (1-p1*v2/(v3*v3))*v4b
 v4b = 0.0
........................../*restore previous state*/
 v1b = v1b + 2.0*v3b
 v3b = 0.0
........................../*restore previous state*/
          ...
```

Differentiated instructions inserted
into the inverse of P's original control flow.

Control Flow Inversion : conditionals

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)
```

Control Flow Inversion : loops

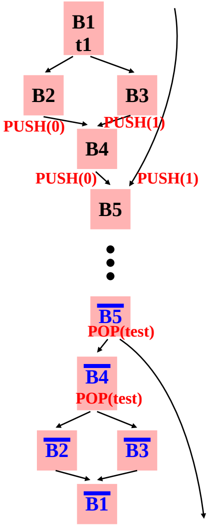
Reversed loops run in the inverse order

```
...
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

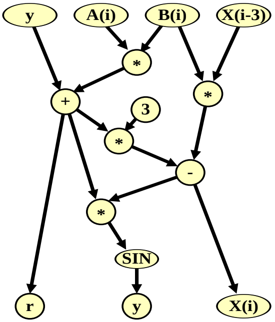
1. Introduction
2. Formalization
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. Some AD Tools
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

Yet another formalization using computation graphs

A sequence of instructions corresponds to a computation graph

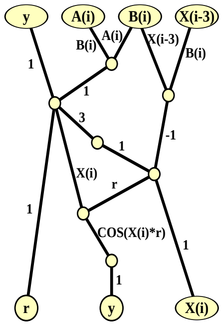
```
DO i=1,n
  IF (B(i).gt.0.0) THEN
    r = A(i)*B(i) + y
    X(i) = 3*r - B(i)*X(i-3)
    y = SIN(X(i)*r)
  ENDIF
ENDDO
```

Source program

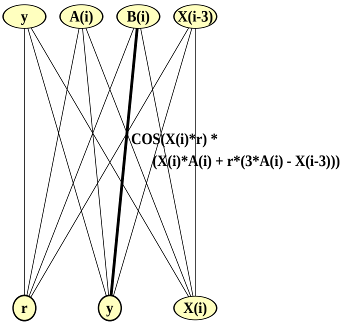


Computation Graph

Jacobians by Vertex Elimination



Jacobian Computation Graph



Bipartite Jacobian Graph

- Forward vertex elimination $\Rightarrow$ tangent AD.
- Reverse vertex elimination $\Rightarrow$ 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 $\mathcal{L}(v_1, v_2, v_3, v_4, \overline{v_3}, \overline{v_4}) =$
$v_4 + \overline{v_3}.(-v_3 + 2.v_1 + 5) + \overline{v_4}.(-v_4 + v_3 + p_1 * v_2/v_3)$
is such that:

$$\overline{v_1} = \frac{\partial v_4}{\partial v_1} = \frac{\partial \mathcal{L}}{\partial v_1} \quad \text{and} \quad \overline{v_2} = \frac{\partial v_4}{\partial v_2} = \frac{\partial \mathcal{L}}{\partial v_2}$$

provided

$$\frac{\partial \mathcal{L}}{\partial v_3} = \frac{\partial \mathcal{L}}{\partial v_4} = \frac{\partial \mathcal{L}}{\partial \overline{v_3}} = \frac{\partial \mathcal{L}}{\partial \overline{v_4}} = 0$$

The $\overline{v_i}$ are the Lagrange multipliers associated to the instruction that sets v_i.

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

$$\overline{v_4}.(1 - p_1.v_2/(v_3.v_3)) - \overline{v_3} = 0$$

To be compared with instruction

```
v3b = v3b + (1-p1*v2/(v3*v3))*v4b
```

(initial `v3b` is set to `0.0`)

Outline

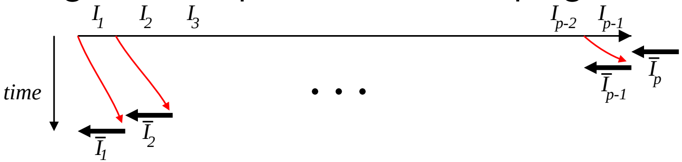
1. Introduction
2. Formalization
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. Some AD Tools
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

Time/Memory tradeoffs for reverse AD

From the definition of the gradient $\overline{X}$

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0)\dots f_p'^t(W_{p-1}).\overline{Y}$$

we get the general shape of reverse AD program:



⇒ How can we restore previous values?

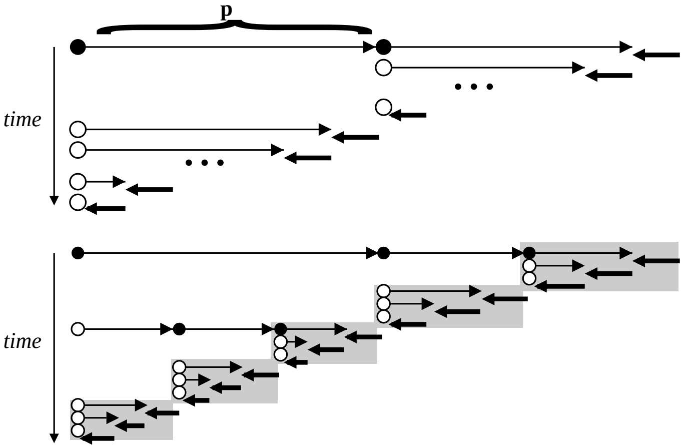
Restoration by recomputation (RA: Recompute-All)

Restart execution from a stored initial state:

Memory use low, CPU use high $\Rightarrow$ trade-off needed !

Checkpointing (RA strategy)

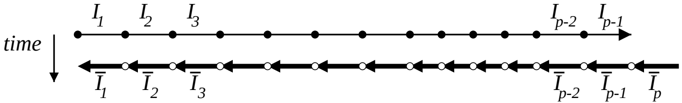
On selected pieces of the program, possibly nested, remember the output state to avoid recomputation.



Memory and CPU grow like $log(size(\mathtt{P}))$

Restoration by storage (SA: Store-All)

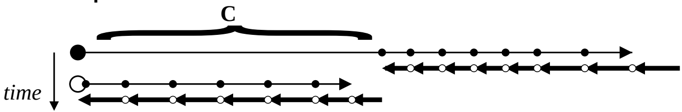
Progressively undo the assignments made by the forward sweep

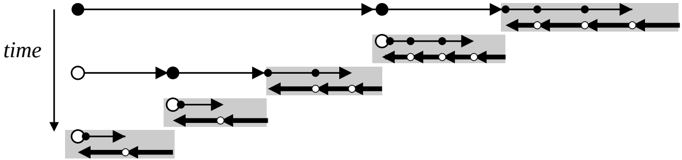


Memory use high, CPU use low $\Rightarrow$ trade-off needed !

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.

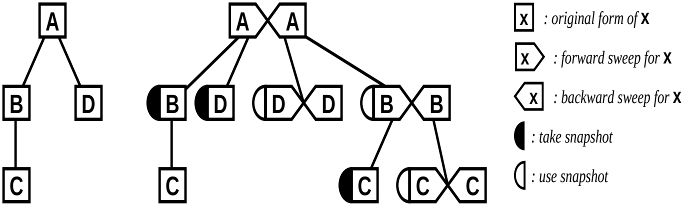




Memory and CPU grow like *log*(*size*(P))

Checkpointing on calls (SA)

A classical choice: checkpoint procedure calls !



Memory and CPU grow like *log*(*size*(P)) when call tree is well balanced.

Ill-balanced call trees require not checkpointing some calls

Careful analysis keeps the snapshots small.

Outline

1. Introduction
2. Formalization
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. Some AD Tools
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

Multi-directional mode and Jacobians

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 Jacobian is sparse,
use "seed matrices" to propagate fewer $\dot{X}$ or $\overline{Y}$

- 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:

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

Outline

1. Introduction
2. Formalization
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. Some AD Tools
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

Applications to Optimization

From a simulation program $\mathtt{P}$:

$$\mathtt{P} : \textit{(design parameters)}\gamma \mapsto \textit{(cost function)}J(\gamma)$$

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

Reverse mode AD builds program $\overline{\mathtt{P}}$ 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: $W_0, W_1, W_2, ..., W_\infty$

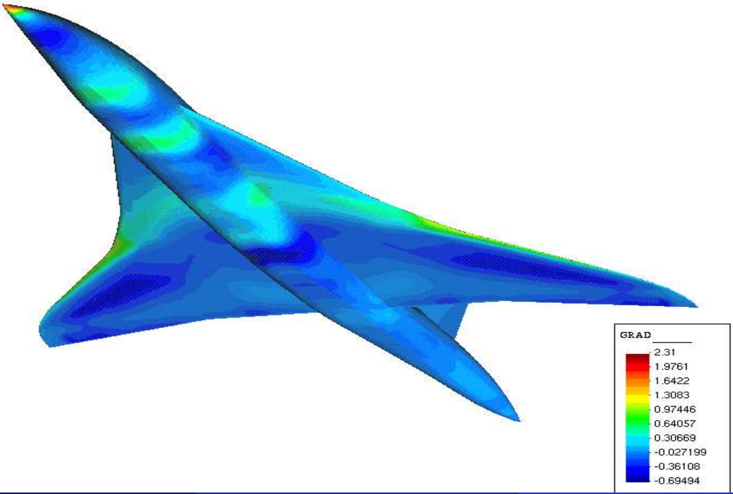
then pure reverse AD of P may prove too expensive (memory...)

Solutions exist:

- reverse AD on the final steady state only.
- *Andreas Griewank's* "Piggy-backing"
- reverse AD on Ψ alone + hand-coding

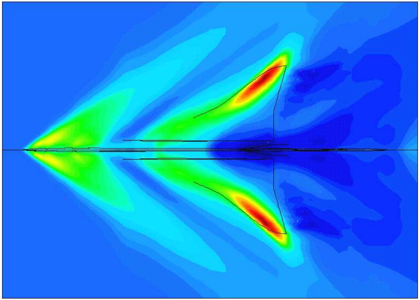
A color picture *(at last !...)*

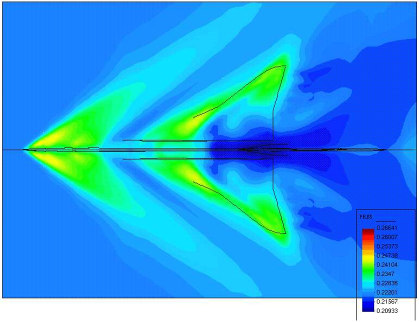
AD-computed gradient of a scalar cost (sonic boom) with respect to skin geometry:



… and after a few optimization steps

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





(*Plane geometry provided by Dassault Aviation*)

Outline

1. Introduction
2. Formalization
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. Some AD Tools
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

Studying Uncertainties

Assume a state W is defined as a function $W(c)$ of uncertain parameters c.
Assume a scalar cost function $J(W)$ is defined on W.

To model the influence of c on $J(W(c))$, numericians want

$$\frac{dJ}{dc} \quad \textbf{and also} \quad \frac{d^2J}{dc^2}$$

Repeated application of AD, Tangent-on-Reverse

Given the program `W` that computes (solves?) $W(c)$
and the program `J` that computes the cost $j = J(W)$
we may very well apply AD to $\mathtt{Q}(c) = \mathtt{J}(\mathtt{W}(c)) = j$!

$$\mathtt{Q} : \quad c \quad \mapsto \quad j \qquad time : t$$

$$\overline{\mathtt{Q}} : \quad c, (\bar{j} \doteq 1) \quad \mapsto \quad \bar{c} \doteq \left(\frac{\partial j}{\partial c_i}\right)_{\forall i} \qquad time : 4t$$

$$\dot{\overline{\mathtt{Q}}} : \quad c, \dot{c} \doteq e_k \quad \mapsto \quad \dot{\bar{c}}_k \doteq \left(\frac{\partial^2 j}{\partial c_i \partial c_k}\right)_{\forall i} \qquad time : 16t$$

$$\dot{\overline{\mathtt{Q}}}^* : \quad c, (\dot{c}) \doteq (e_k)_{\forall k} \quad \mapsto \quad (\dot{\bar{c}}_k)_{\forall k} \doteq \left(\frac{\partial^2 j}{\partial c_i \partial c_k}\right)_{\forall i,k} \qquad time : 16mt$$

The problem of Implicit Formulations

The cost function $J(W)$ is explicit and relatively simple but the state W is often defined implicitely by

$$\Psi(W, c) = 0$$

Program `W` includes an iterative solver !

$\Rightarrow$ Do we really want to differentiate this? *(No!...)*

$\Rightarrow$ Let's go back up to the math level !

First derivative

Differentiating the implicit state equation wrt c, we get:

$$\frac{\partial\Psi}{\partial W} \cdot \frac{\partial W}{\partial c} + \frac{\partial\Psi}{\partial c} = 0 \;\Rightarrow\; \frac{\partial W}{\partial c} = -\left[\frac{\partial\Psi}{\partial W}\right]^{-1} \cdot \frac{\partial\Psi}{\partial c}$$

So we can write the gradient:

$$\frac{dJ}{dc} = \frac{\partial J}{\partial W} \cdot \frac{\partial W}{\partial c} = -\frac{\partial J}{\partial W} \cdot \left[\frac{\partial\Psi}{\partial W}\right]^{-1} \cdot \frac{\partial\Psi}{\partial c}$$

For efficiency reasons, it's best to solve for Π first:

$$\frac{\partial\Psi}{\partial W}^{*} \cdot \Pi = \frac{\partial J}{\partial W}^{*}$$

How to solve an adjoint equation

Π is often called an adjoint state. Its adjoint equation is of the general shape:

$$\frac{\partial \Psi}{\partial W}^* \cdot \Pi = Y$$

We can solve it iteratively ("matrix-free resolution"), provided repeated computations, for various X's, of

$$\frac{\partial \Psi}{\partial W}^* \cdot X$$

Calling `Psi` the procedure that computes $\Psi(W, c)$, $\overline{\texttt{Psi}}_W$, reverse AD of `Psi` wrt W, computes just that !

Second derivatives

Differentiating $\frac{dJ}{dc}$ again, we get

$$\frac{d^2J}{dc^2} = -\frac{d\Pi}{dc} \cdot \frac{\partial \Psi}{\partial c} - \Pi \cdot \frac{d}{dc}\left(\frac{\partial \Psi}{\partial W}\right)$$

AD can help computing every term of this formula.
Let's focus for example on $\frac{d\Pi}{dc}$:
$\Rightarrow$ we can play the adjoint trick again!

Solving for $\frac{d\Pi}{dc}$

Again we go back to an implicit equation, now for Π:

$$\frac{\partial \Psi}{\partial W}^* \cdot \Pi = \frac{\partial J}{\partial W}^*$$

Differentiating it wrt c, we get:

$$\left[\frac{d}{dc}\left(\frac{\partial \Psi}{\partial W}^*\right)\right] \cdot \Pi + \frac{\partial \Psi}{\partial W}^* \cdot \frac{d\Pi}{dc} = \frac{d}{dc}\left(\frac{\partial J}{\partial W}^*\right)$$

which rewrites as

$$\frac{\partial \Psi}{\partial W}^* \cdot \frac{d\Pi}{dc} = \frac{d}{dc}\left(\frac{\partial J}{\partial W}\right) - \frac{d}{dc}\left(\frac{\partial \Psi}{\partial W}^* \cdot \Pi_{c_0}\right)$$

Solving for $\frac{\partial \Pi}{\partial c}$ using AD

$$\frac{\partial \Psi}{\partial W}^{*} \cdot \frac{d\Pi}{dc} = \frac{d}{dc}\left(\frac{\partial J}{\partial W}\right) - \frac{d}{dc}\left(\frac{\partial \Psi}{\partial W}^{*} \cdot \Pi_{c_0}\right)$$

matrix-free resolution, using $\overline{\texttt{Psi}}_W$

$\overline{\texttt{J}}_W$

$\dot{\overline{\texttt{J}}}_W$

$\overline{\texttt{Psi}}_W$

on $\overline{\Psi} \doteq \Pi_{c_0}$

$\dot{\overline{\texttt{Psi}}}_W$

Outline

1. Introduction
2. Formalization
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. **Some AD Tools**
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

Tools for overloading-based AD

If language supports overloading (F95, C++)
Tool provides:

- help for "re-typing" diff variables
- a library of overloaded operations

The reverse mode, or cross-country elimination, cannot be done on the fly. Tools use

- a tape recording of partial derivatives and execution trace
- a special program to compute the derivatives from the tape.

Tools for source-transformation AD

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

parsing	**→analysis**	**→differentiation**
F77	type-checking	tangent
F9X	use/kill	reverse
C	dependencies	multi-directional
MATLAB	activity	...
...	...	

Some AD tools

- NAGWARE F95 Compiler: Overloading, tangent, reverse
- ADOL-C : Overloading+Tape; tangent, reverse, higher-order
- ADIFOR : 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 very large codes

Technical Difficulties:

- Pointers and memory allocation
- Objects
- Inversion or Duplication of random control (communications, random,...)

Outline

1. Introduction
2. Formalization
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. Some AD Tools
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

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 ⇒ simplifications

orig. prog	tangent mode	w/activity analysis
	`cd = a*bd + ad*b`	`cd = a*bd + ad*b`
`c = a*b`	`c = a*b`	`c = a*b`
	`ad = 0.0`	
`a = 5.0`	`a = 5.0`	`a = 5.0`
	`dd = a*cd + ad*c`	`dd = a*cd`
`d = a*c`	`d = a*c`	`d = a*c`
	`ed=ad/c-a*cd/c**2`	
`e = a/c`	`e = a/c`	`e = a/c`
	`ed = 0.0`	`ed = 0.0`
`e=floor(e)`	`e = floor(e)`	`e = floor(e)`

Adjoint Liveness

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 $\overline{\texttt{P}}$.
- Recursively, other instructions of `P` can be removed too.

orig. program	reverse mode	Adjoint Live code
IF(a.GT.0.)THEN	IF(a.GT.0.)THEN	IF (a.GT.0.) THEN
	CALL PUSH(a)	
a = LOG(a)	a = LOG(a)	
	CALL POP(a)	
	ab = ab/a	ab = ab/a
ELSE	ELSE	ELSE
a = LOG(c)	a = LOG(c)	a = LOG(c)
CALL SUB(a)	CALL PUSH(a)	
ENDIF	CALL SUB(a)	
END	CALL POP(a)	
	CALL SUB_B(a,ab)	CALL SUB_B(a,ab)
	cb = cb + ab/c	cb = cb + ab/c
	ab = 0.0	ab = 0.0
	END IF	END IF

"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.
$\Rightarrow$ not To Be Restored.

```
x = x + EXP(a)
y = x + a**2
a = 3*z
```

reverse mode: naive backward sweep	reverse mode: backward sweep with TBR
`CALL POP(a)` `zb = zb + 3*ab` `ab = 0.0` `CALL POP(y)` `ab = ab + 2*a*yb` `xb = xb + yb` `yb = 0.0` `CALL POP(x)` `ab = ab + EXP(a)*xb`	`CALL POP(a)` `zb = zb + 3*ab` `ab = 0.0` `ab = ab + 2*a*yb` `xb = xb + yb` `yb = 0.0` `ab = ab + EXP(a)*xb`

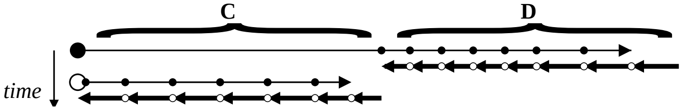
Aliasing

In reverse AD, it is important to know whether two variables in an instruction are the same.

`a[i] = 3*a[i+1]`	`a[i] = 3*a[i]`	`a[j] = 3*a[j]`
variables certainly different	variables certainly equal	? $\Rightarrow$ `tmp = 3*a[j]` `a[i] = tmp`
`ab[i+1]= ab[i+1] + 3*ab[i]` `ab[i] = 0.0`	`ab[i] = 3* ab[i]`	`tmpb = ab[i]` `ab[i] = 0.0` `ab[j] = ab[j] + 3*tmpb`

Snapshots

Taking small snapshots saves a lot of memory:



$$\mathit{Snapshot}(\mathrm{C}) = \mathit{Use}(\overline{\mathrm{C}}) \cap (\mathit{Write}(\mathrm{C}) \cup \mathit{Write}(\overline{\mathrm{D}}))$$

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"
- $\Rightarrow$ 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
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. Some AD Tools
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

A word on TAPENADE

Automatic Differentiation Tool

Name: TAPENADE version 2.1

Date of birth: January 2002

Ancestors: Odyssée 1.7

Address: www.inria.fr/tropics/tapenade.html

Specialties: AD Reverse, Tangent, Vector Tangent, Restructuration

Reverse mode Strategy: Store-All, Checkpointing on calls

Applicable on: FORTRAN95, FORTRAN77, and older

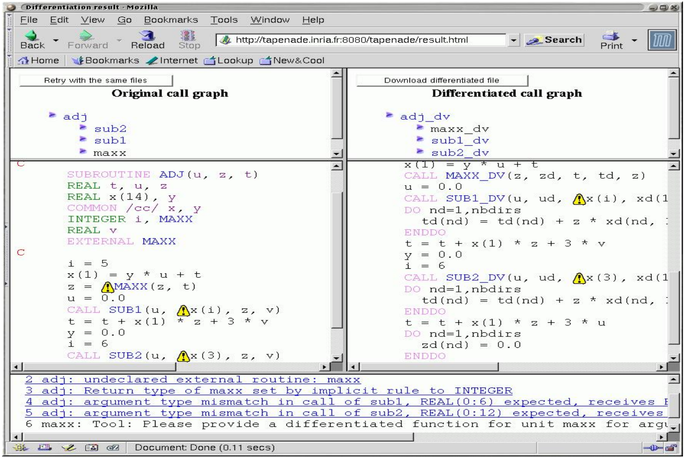
Implementation Languages: 90% JAVA, 10% C

Availability: Java classes for Linux and Windows, or Web server

Internal features: Type-Checking, Read-Written Analysis, Fwd and Bwd Activity, Adjoint Liveness analysis, TBR, . . .

TAPENADE on the web

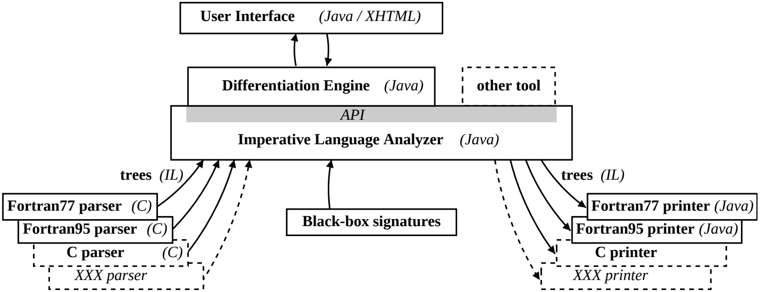
http://www-sop.inria.fr/tropics



applied to industrial and academic codes:
Aeronautics, Hydrology, Chemistry, Biology, Agronomy...

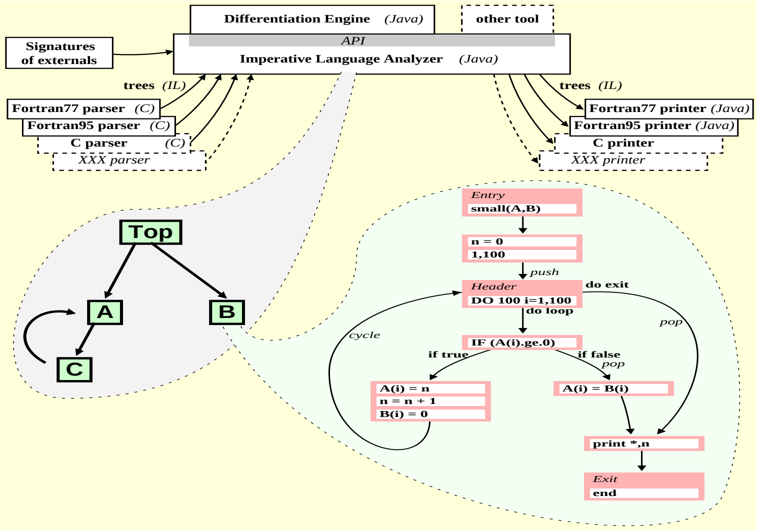
TAPENADE Architecture

- Use a general abstract *Imperative Language (IL)*
- Represent programs as *Call Graphs* of *Flow Graphs*



TAPENADE Program Internal Representation

using Calls-Graphs and Flow-Graphs:



Outline

1. Introduction
2. Formalization
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. Some AD Tools
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

Validation methods

From a program P that evaluates

$$F : \mathbf{R}^m \rightarrow \mathbf{R}^n$$
$$X \mapsto Y$$

tangent AD creates

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

and reverse AD creates

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

Wow can we validate these programs ?

- Tangent wrt Divided Differences
- Reverse wrt Tangent

Validation of Tangent *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 :

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

Often called the "dot-product test"

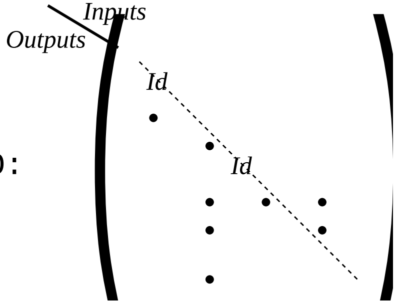
Outline

1. Introduction
2. Formalization
3. Reverse AD
4. Alternative formalizations
5. Memory issues in Reverse AD: Checkpointing
6. Multi-directional
7. Reverse AD for Optimization
8. AD for Sensitivity to Uncertainties
9. Some AD Tools
10. Static Analyses in AD tools
11. The TAPENADE AD tool
12. Validation of AD results
13. Expert-level AD
14. Conclusion

Black-box routines

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

FOO:



After AD, provide required hand-coded derivative (`FOO_D` or `FOO_B`)

Linear or auto-adjoint procedures

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

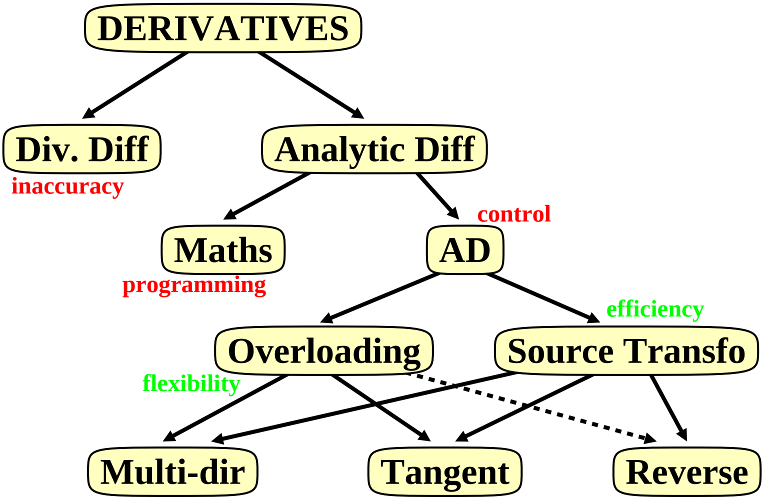
Standard:		*Improved:*
`do i = 1,N` *body(i)* `end` `do i = N,1` $\overleftarrow{body(i)}$ `end`	$\Longleftrightarrow$	`do i = 1,N` *body(i)* $\overleftarrow{body(i)}$ `end`

In the Recompute-All context, this reduces the memory consumption by a factor `N`

Outline

1 Introduction
2 Formalization
3 Reverse AD
4 Alternative formalizations
5 Memory issues in Reverse AD: Checkpointing
6 Multi-directional
7 Reverse AD for Optimization
8 AD for Sensitivity to Uncertainties
9 Some AD Tools
10 Static Analyses in AD tools
11 The TAPENADE AD tool
12 Validation of AD results
13 Expert-level AD
14 Conclusion

AD: Context


DERIVATIVES
Div. Diff
inaccuracy
Analytic Diff
Maths
programming
control
AD
efficiency
Overloading
Source Transfo
flexibility
Multi-dir
Tangent
Reverse

AD: To Bring Home

- 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 sensitivity analysis or optimization.
- Reverse AD is a discrete equivalent of the adjoint methods from control theory: gives a gradient at remarkably low cost.

AD tools: To Bring Home

- 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.