Automatic Differentiation by Program Transformation

Laurent Hascoët

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

Ecole d'été CEA-EDF-INRIA,
Juin 2006

Outline

So you need derivatives ?...

Given a program P computing a function F

$$\begin{array}{rcccc} F & : & \mathbb{R}^m & \rightarrow & \mathbb{R}^n \\ & & X & \mapsto & Y \end{array}$$

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.

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, ...

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

Computer Programs as Functions

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:

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

Tangent mode and Reverse mode

Full $f'(X)$ is expensive and often useless.
We'd better compute useful "projections".

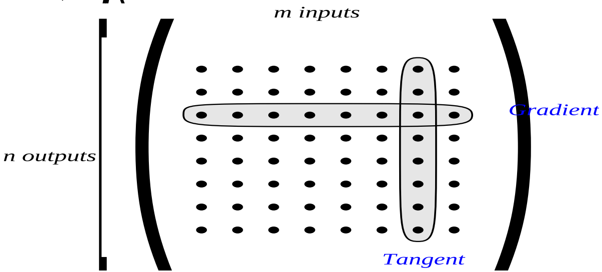
$$\begin{aligned} \text{tangent AD :} & \\ \dot{Y} = f'(X).\dot{X} = & \; f'_p(W_{p-1}).f'_{p-1}(W_{p-2}) \dots f'_1(W_0).\dot{X} \\ \text{reverse AD :} & \\ \overline{X} = f'^t(X).\overline{Y} = & \; f'^t_1(W_0) \dots . f'^t_{p-1}(W_{p-2}).f'^t_p(W_{p-1}).\overline{Y} \end{aligned}$$

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

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



- $f'(X) \sim$ costs $(m+1?) * \mathrm{P}$ using Divided Differences
- $f'(X)$ costs $m * 4 * \mathrm{P}$ using the tangent mode
  Good if $m <= n$
- $f'(X)$ 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

Focus on the Reverse mode

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

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

Focus on the Reverse mode

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

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

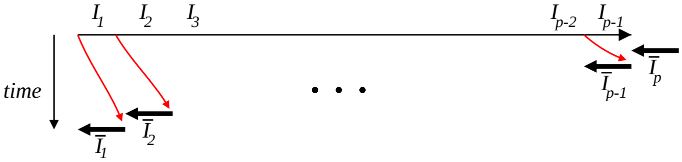
Focus on the Reverse mode

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

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

Instructions differentiated in the reverse order !

Reverse mode: graphical interpretation



Bottleneck: memory usage ("Tape").

Back to the example

```
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)
endif
...
```

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
```

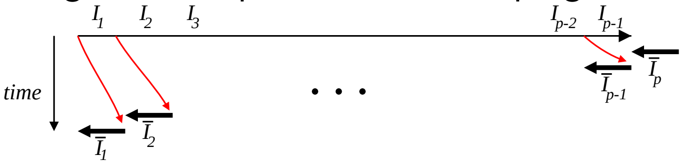
Outline

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:



$\Rightarrow$ How can we restore previous values?

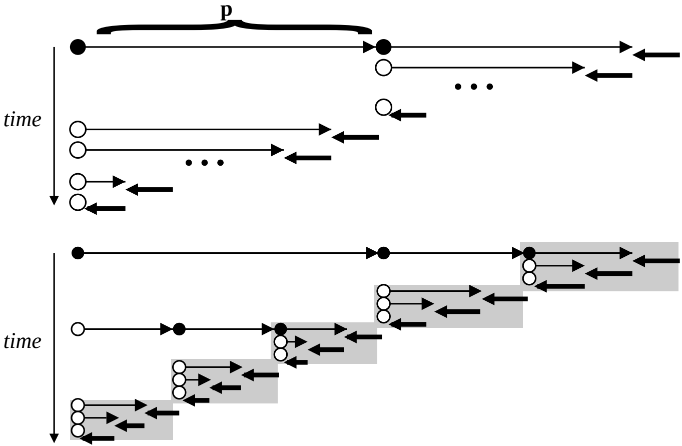
Restoration by recomputation (RA: Recompute-All)

Restart execution from a stored initial state:

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

Checkpointing (RA strategy)

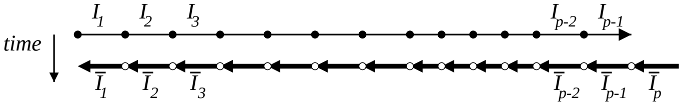
On selected pieces of the program, possibly nested, remember the output state to avoid recomputation.



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

Restoration by storage (SA: Store-All)

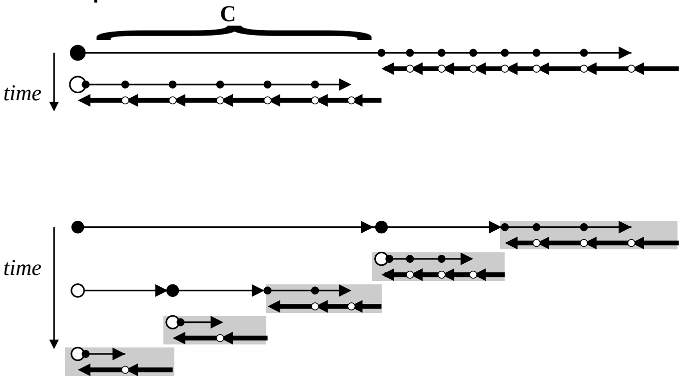
Progressively undo the assignments made by the forward sweep



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

Checkpointing (SA strategy)

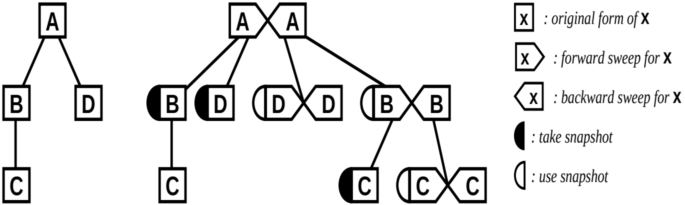
On selected pieces of the program, possibly nested, don't store intermediate values and re-execute the piece when values are required.



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

Applications to Minimization

From a simulation program P :

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

$$\mathtt{P} : \textit{(parameters to estimate)} \gamma \mapsto \textit{(misfit function)} j(\gamma)$$

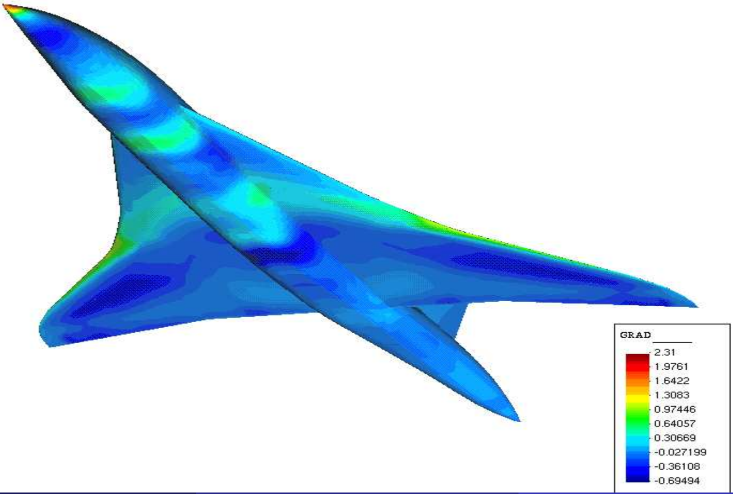
it takes a gradient $j'(\gamma)$ to obtain a minimization program.

Reverse mode AD builds program $\overline{\mathtt{P}}$ that computes $j'(\gamma)$

Minimization algorithms (Gradient descent, SQP, ...) may also use 2nd derivatives. AD can provide them too.

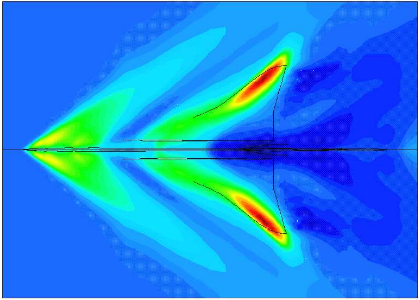
A color picture *(at last !...)*

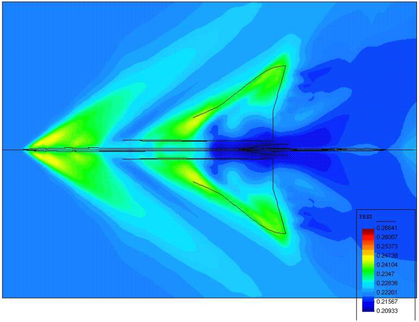
AD-computed gradient of a scalar cost (sonic boom) with respect to skin geometry:



… and after a few optimization steps

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

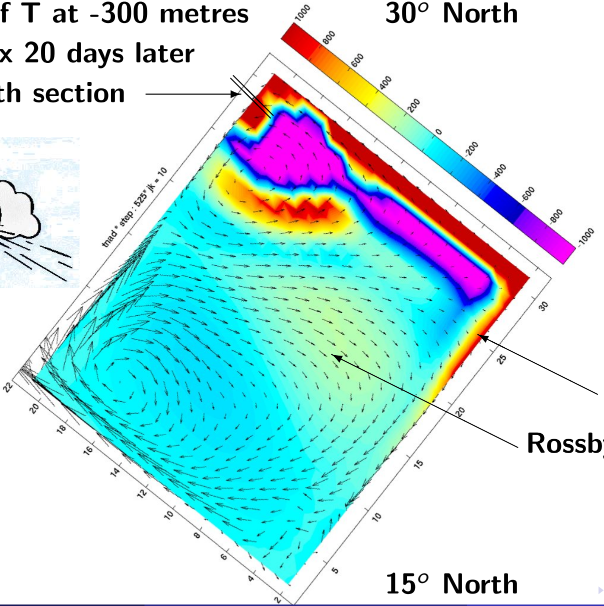




(*Plane geometry provided by Dassault Aviation*)

Data Assimilation (OPA 9.0/GYRE)

Influence of T at -300 metres on heat flux 20 days later across North section



Outline

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

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

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)

“To Be Recorded” analysis

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

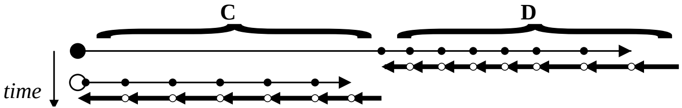
Variables occurring only in linear expressions do not appear in the differentiated instructions.
$\Rightarrow$ not To Be Recorded.

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

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

Snapshots

Taking small snapshots saves a lot of memory:



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

Outline

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

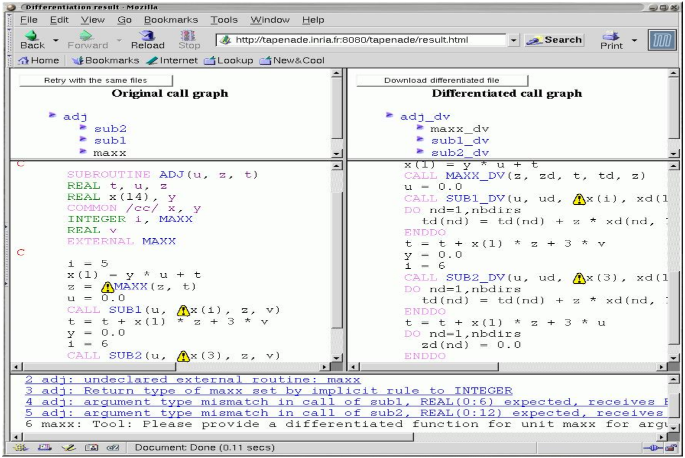
Implementation Languages: 90% JAVA, 10% C

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

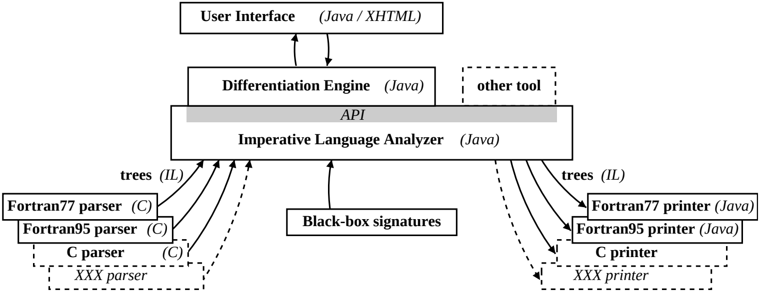
http://www-sop.inria.fr/tropics



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*



Outline

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 $g(h \in \boldsymbol{R}) = F(X + h.Xd)$:

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

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 $\dot{Y}$

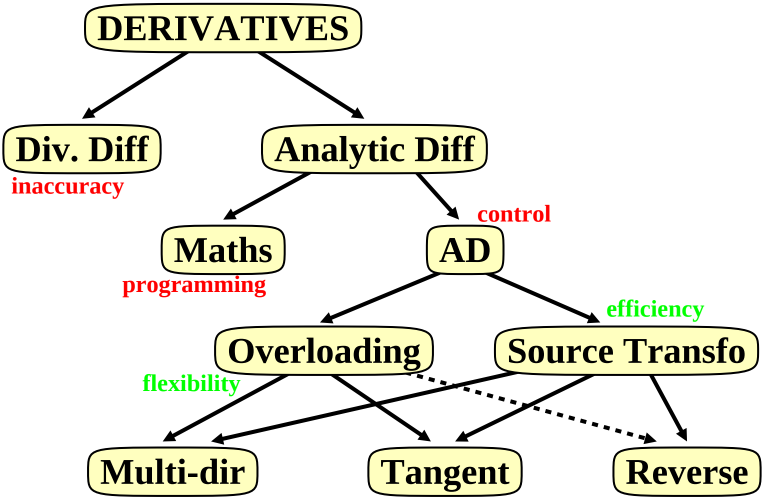
Initialize $\overline{Y} = \dot{Y}$ and run the reverse code, yielding $\overline{X}$.
We have :

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

Often called the “dot-product test”

Outline

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.