Automatic Differentiation of programs and its applications to Scientific Computing

Laurent Hascoët

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

September 2010

Outline

This is AD !

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

  REAL v1,v2,v3,v4,p1


  v3 = 2.0*v1 + 5.0


  v4 = v3 + p1*v2/v3
END
```

This is AD !

```
SUBROUTINE FȮO(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

Computer Programs as Functions

See any program $\mathtt{P}:\{I_1; I_2; \dots I_p; \}$ as:

$$f : \mathbb{R}^m \to \mathbb{R}^n \qquad f = f_p \circ f_{p-1} \circ \cdots \circ f_1$$

Define for short:

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

The chain rule yields:

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

Tangent mode and Reverse mode

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

tangent AD :

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

reverse AD :

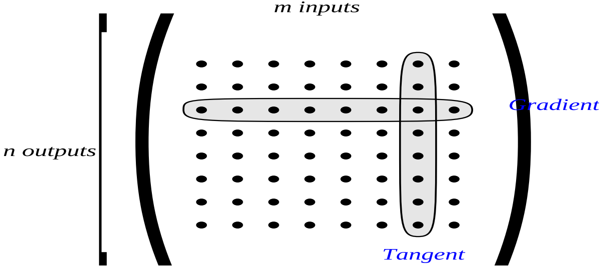
$$\overline{X} = f'^t(X).\overline{Y} = f'^t_1(W_0) \dots f'^t_{p-1}(W_{p-2}).f'^t_p(W_{p-1}).\overline{Y}$$

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

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



- $f'(X)$ 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)

Focus on the Reverse mode (Gradients)

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0) \;\ldots\; f_{p-1}'^t(W_{p-2}) \;.\; f_p'^t(W_{p-1}) \;.\; \overline{Y}$$

I_1 ;

...

I_{p-2} ;

I_{p-1} ;

$\overline{W}$ = $\overline{Y}$;

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

Focus on the Reverse mode (Gradients)

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0) \; \ldots \; f_{p-1}'^t(W_{p-2}) \; . \; f_p'^t(W_{p-1}) \; . \overline{Y}$$

I_1 ;
...
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 (Gradients)

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0) \ \ldots \ f_{p-1}'^t(W_{p-2}) \ . \ f_p'^t(W_{p-1}) \ . \ \overline{Y}$$

I_1 ;

...

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} ;

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

...

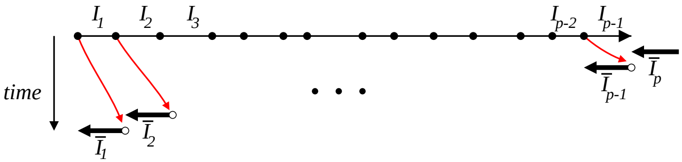
Restore W_0 *before* I_1 ;

$\overline{W} = f_1'^t(W_0) * \overline{W}$;

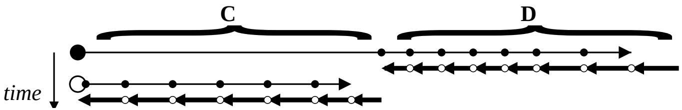
$\overline{X} = \overline{W}$;

Instructions differentiated in the reverse order !

Reverse mode: graphical interpretation



- A Forward sweep followed by Backward sweep
- Bottleneck: Uses a large memory "Tape"
- Trade-off strategy: "Checkpointing"



Outline

So you need derivatives ?...

Given a program P computing a function F

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

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 $= \overline{Y}^t \times J$
- 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 !

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
```

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
```

AD Example: analytic 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
```

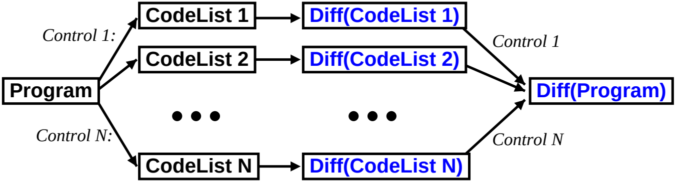
Just inserts "differentiated instructions" into FOO

Outline

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

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

  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 $\texttt{P} : \{I_1; I_2; \ldots I_{p-1}; I_p;\}$ as

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

Using the Chain Rule

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

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

tangent AD :

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

reverse AD :

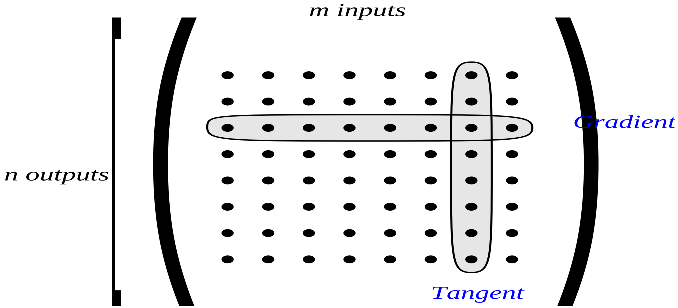
$$\overline{X} = f'^t(X).\overline{Y} = f'^t_1(W_0)\dots f'^t_{p-1}(W_{p-2}).f'^t_p(W_{p-1}).\overline{Y}$$

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

$F \quad : \quad R^m \quad \rightarrow \quad R^n$



- 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:

$$f'(X) = \ldots \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ 0 & \frac{p_1}{v_3} & 1 - \frac{p_1 * v_2}{v_3^2} & 0 \end{pmatrix} \begin{pmatrix} 1 & & & \\ & 1 & & \\ 2 & & 0 & \\ & & & 1 \end{pmatrix} \begin{pmatrix} \dot{v}_1 \\ \dot{v}_2 \\ \dot{v}_3 \\ \dot{v}_4 \end{pmatrix}$$

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

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

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 sparse Jacobian, 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

Focus on the Reverse mode

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0) \ \ldots \ f_{p-1}'^t(W_{p-2}) \ . \ f_p'^t(W_{p-1}) \ . \ \overline{Y}$$

I_1 ;

...

I_{p-2} ;

I_{p-1} ;

$\overline{W} = \overline{Y}$;

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

Focus on the Reverse mode

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0)\ \dots\ f_{p-1}'^t(W_{p-2})\ .\ f_p'^t(W_{p-1})\ .\ \overline{Y}$$

I_1 ;
...
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} ;
$\overline{W} = f_{p-1}'^t(W_{p-2}) * \overline{W}$;

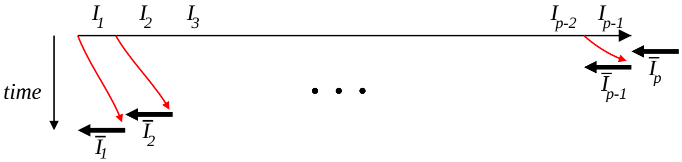
Focus on the Reverse mode

$$\overline{X} = f'^t(X).\overline{Y} = f_1'^t(W_0)\ \ldots\ f_{p-1}'^t(W_{p-2})\ .\ f_p'^t(W_{p-1})\ .\ \overline{Y}$$

I_1 ;
...
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} ;
$\overline{W} = f_{p-1}'^t(W_{p-2}) * \overline{W}$;
...
Restore W_0 before I_1 ;
$\overline{W} = f_1'^t(W_0) * \overline{W}$;
$\overline{X} = \overline{W}$;

Instructions differentiated in the reverse order !

Reverse mode: graphical interpretation



Bottleneck: memory usage (“Tape”).

Still searching for optimal combinations of **storage**, **recomputation** and even **inversion**.

Back to the example

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

Transposed Jacobian matrices:

$$f'^{t}(X) = \ldots \begin{pmatrix} 1 & & 2 & \\ & 1 & & \\ & & 0 & \\ & & & 1 \end{pmatrix} \begin{pmatrix} 1 & & & 0 \\ & 1 & & \frac{p_1}{v_3} \\ & & 1 & 1 - \frac{p_1 * v_2}{v_3^2} \\ & & & 0 \end{pmatrix} \begin{pmatrix} \overline{v}_1 \\ \overline{v}_2 \\ \overline{v}_3 \\ \overline{v}_4 \end{pmatrix}$$

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

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

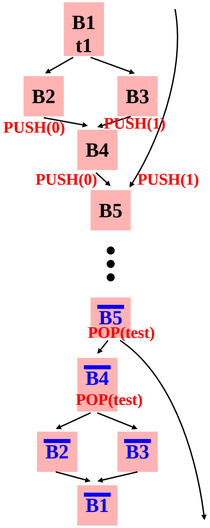
Reversed loops run in the inverse order

```
...
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
```

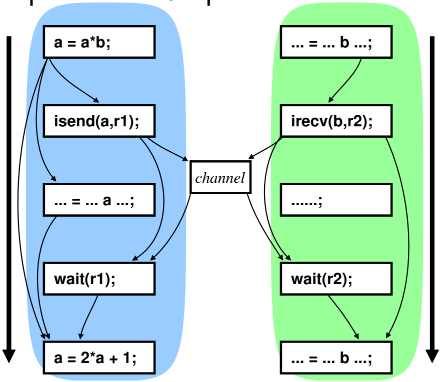
Control Flow Inversion : spaghetti

Remember original Control Flow when it merges



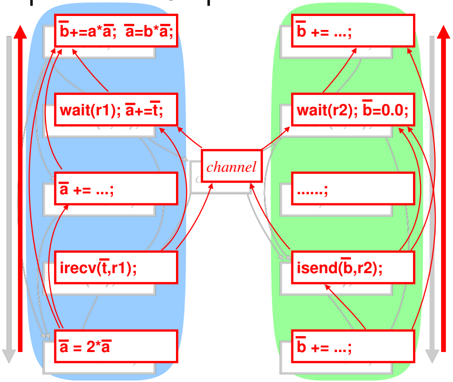
Data Flow Inversion: message-passing parallelism

Consider the Data Dependence Graph of an MPI communication.



Data Flow Inversion: message-passing parallelism

Consider the Data Dependence Graph of an MPI communication.



The reversed communication pattern is designed to inverse data-flow
⇒ and therefore does not introduce deadlocks.

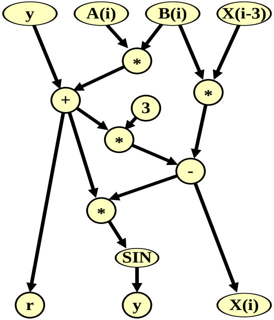
Outline

Yet another formalization using computation graphs

A sequence of instructions corresponds to a computation graph

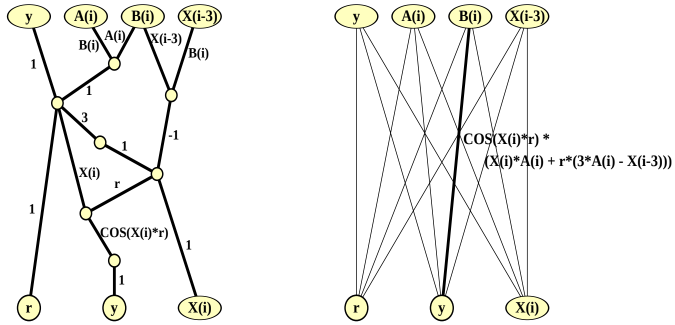
```
DO i=1,n
  IF (B(i).gt.0.0) THEN
    r = A(i)*B(i) + y
    X(i) = 3*r - B(i)*X(i-3)
    y = SIN(X(i)*r)
  ENDIF
ENDDO
```

Source program



Computation Graph

Jacobians by Vertex Elimination



Jacobian Computation Graph *Bipartite Jacobian Graph*

- Forward vertex elimination ⇒ 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 $\mathcal{L}(v_1, v_2, v_3, v_4, \overline{v_3}, \overline{v_4}) =$ $v_4 + \overline{v_3}.(-v_3 + 2.v_1 + 5) + \overline{v_4}.(-v_4 + v_3 + p_1 * v_2/v_3)$ 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`)

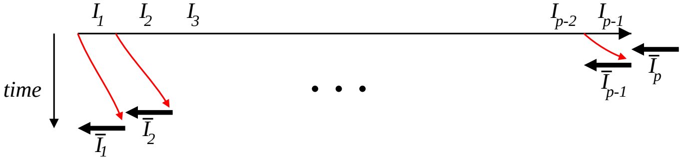
Outline

Time/Memory tradeoffs for reverse AD

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

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

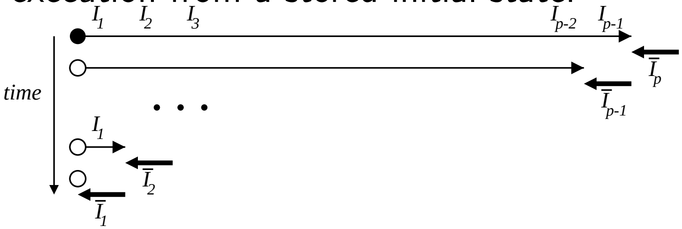
we get the general shape of reverse AD program:



$\Rightarrow$ How can we restore previous values?

Restoration by recomputation (RA: Recompute-All)

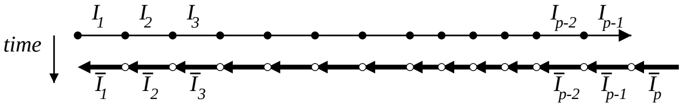
Restart execution from a stored initial state:



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

Restoration by storage (SA: Store-All)

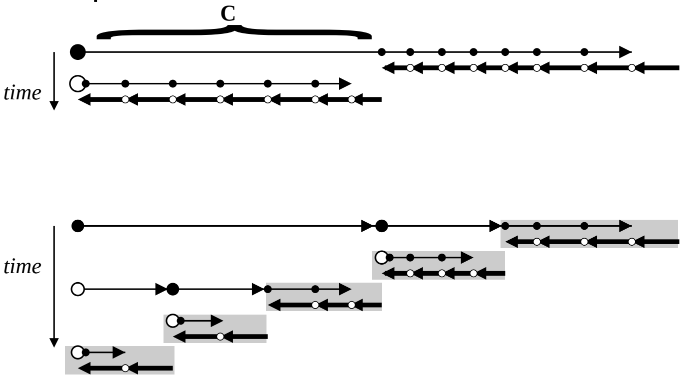
Progressively undo the assignments made by the forward sweep



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

Checkpointing (SA strategy)

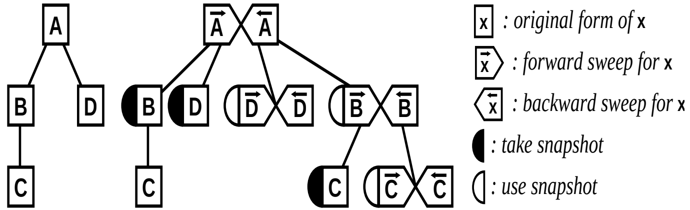
On selected pieces of the program, possibly nested, don't store intermediate values and re-execute the piece when values are required.



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 well balanced.

Ill-balanced call trees require not checkpointing some calls

Careful analysis keeps the snapshots small.

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 ⇒ 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)	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
xb = xb + yb	xb = xb + yb
yb = 0.0	yb = 0.0
CALL POP(x)	
ab = ab + EXP(a)*xb	ab = ab + EXP(a)*xb

Aliasing

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

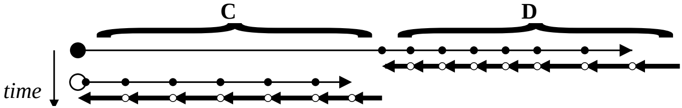
`a[i] = 3*a[i+1]`	`a[i] = 3*a[i]`	`a[i] = 3*a[j]`

variables certainly different	variables certainly equal	? ⇒
		 `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:



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

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

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 $\overline{\mathrm{P}}$ that computes $J'(\gamma)$

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

Taking advantage of 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: $W_0, W_1, W_2, ..., W_\infty$

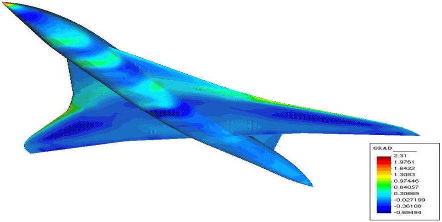
then pure reverse AD of P may prove too expensive (memory...)

Solutions exist:

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

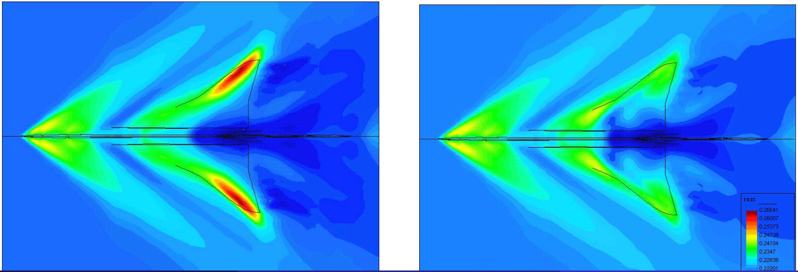
CFD optimization: color pictures...

AD gradient of the cost function on the skin geometry:



(Dassault Aviation)

Sonic boom under the plane after 8 optimization cycles:



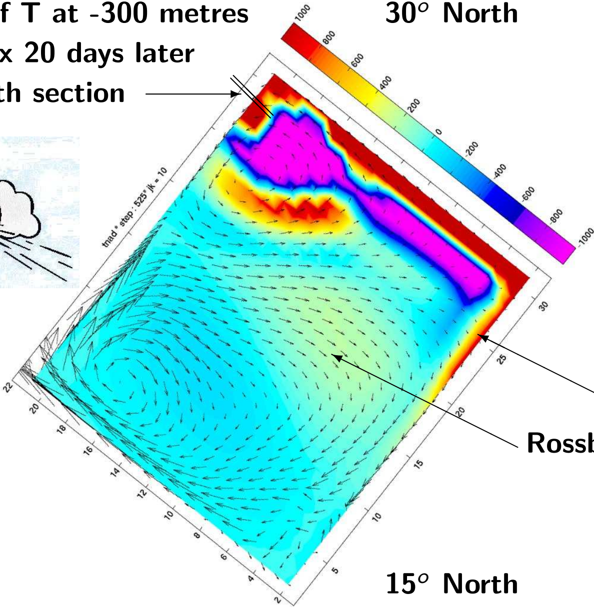
CFD optimization: figures

- Cost function: sonic boom below + lift + drag
- Design parameters: plane skin, (2000 `REAL*8`)
- Specific strategy for a stationnary simulation: assembly of the adjoint linear system through AD, then specific solver.
- Performances:
  - Differentiation time: 2 s.
  - Reverse AD slowdown: 7
  - Adjoint slowdown: 4
  - Reverse AD memory use: 58 `REAL*8` per mesh node

Data Assimilation (OPA 9.0/GYRE)

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

30^o North

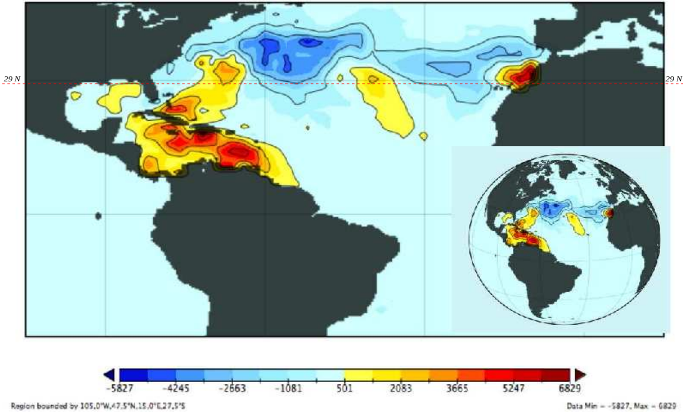


Kelvin wave

Rossby wave

15^o North

Data Assimilation (OPA 9.0/NEMO)



2^o grid cells, one year simulation

Data Assimilation: figures

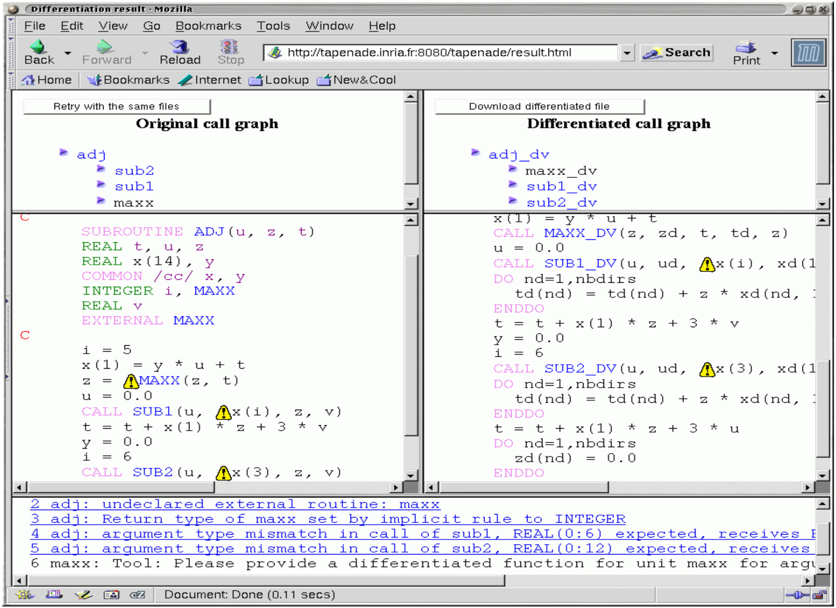
- Code : OPA 9.0. 120000 lines of FORTRAN 95
- Cost function: e.g. heat flux at the end
  vs. temperature, salinity. . . at initial state
- Standard reverse AD of complete simulation
- Differentiation time: 20 s.
- Reverse AD slowdown: 7

Outline

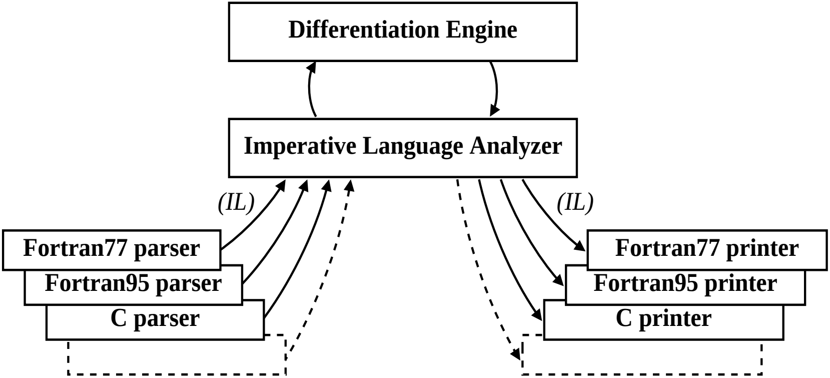
TAPENADE support and directions

- Team's website, tutorial, FAQ:
  http://www-sop.inria.fr/tropics
- Tapenade download site:
  ftp://ftp-sop.inria.fr/tropics/tapenade
- TAPENADE 2.1 user's guide:
  http://www.inria.fr/rrrt/rt-0300.html
- Mailing list:
  tapenade-users@lists-sop.inria.fr

Tapenade Web Interface



Tapenade Architecture



- Language-independent kernel
- Written in Java (100 000 lines)
- Accepts Fortran (77 and 95) and C (August 2008)

Outline

A very simple program

Original program	Tapenade reverse: fwd sweep

<pre>SUBROUTINE S(x, y, r)
  REAL*8 x,y,r
  r = x*y
  r = SQRT(r)
END</pre>
	<pre>SUBROUTINE S_B(x,xb,y,yb,r,rb)
	  REAL*8 x,xb,y,yb,r,rb
	  r = x*y
	  CALL PUSHREAL8(r)
	  r = SQRT(r)
	  ...</pre>

Tapenade tangent	**Tapenade reverse: bwd sweep**

<pre>SUBROUTINE S_D(x,xd,y,yd...)
  REAL*8 x,xd,y,yd,r,rd
  rd = xd*y + x*yd
  r = x*y
  rd = rd/(2.0*SQRT(r))
  r = SQRT(r)
END</pre>
	<pre> ...
	  CALL POPREAL8(r)
	  rb = rb/(2.0*SQRT(r))
	  xb = xb + y*rb
	  yb = yb + x*rb
	  rb = 0.0
	END</pre>

Control structures

Original program

```
SUBROUTINE S1(a, n, x)
...
DO i=2,n,7
 IF (a(i).GT.1.0) THEN
  a(i) = LOG(a(i)) + a(i-1)
 END IF
ENDDO
```

Tapenade reverse: fwd sweep

```
DO i=2,n,7
 IF (a(i).GT.1.0) THEN
  CALL PUSHREAL4(a(i))
  a(i) = LOG(a(i))+a(i-1)
  CALL PUSHINTEGER4(1)
 ELSE
  ...
```

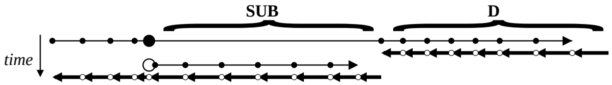
Tapenade tangent

```
SUBROUTINE S1_D(a,ad,n,x)
...
DO i=2,n,7
 IF (a(i).GT.1.0) THEN
  ad(i)=ad(i)/a(i)+ad(i-1)
  a(i) = LOG(a(i)) + a(i-1)
 END IF
```

Tapenade reverse: bwd sweep

```
CALL POPINTEGER4(adTo)
DO i=adTo,2,-7
 CALL POPINTEGER4(branch)
 IF (branch .GE. 1) THEN
  CALL POPREAL4(a(i))
  ab(i-1) = ab(i-1) + ab(i)
  ab(i) = ab(i)/a(i)
```

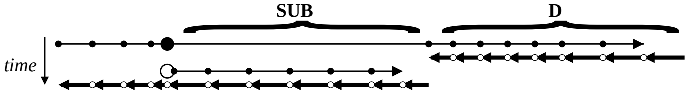
Procedure calls and Checkpointing



Original program	Tapenade reverse: fwd sweep
x = x**3 CALL SUB(a, x, 1.5, z) x = x*y	CALL PUSHREAL4(x) x = x**3 CALL PUSHREAL4(x) CALL SUB(a, x, 1.5, z) x = x*y
Tapenade tangent	**Tapenade reverse: bwd sweep**
xd = 3*x**2*xd x = x**3 CALL SUB_D(a, ad, x, xd, 1.5, 0.0, z) xd = y*xd x = x*y	xb = y*xb CALL POPREAL4(x) CALL SUB_B(a, ab, x, xb, 1.5, arg2b, z) CALL POPREAL4(x) xb = 3*x**2*xb

Snapshots for Checkpointing

Snapshots must be as small as possible:



$$\textbf{Snapshot}(\mathrm{SUB}) \subseteq \textbf{Use}(\overline{\mathrm{SUB}}) \cap (\textbf{Out}(\mathrm{SUB}) \cup \textbf{Out}(\overline{\mathrm{D}}))$$

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)`

"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(y)`	
`ab = ab + EXP(a)*yb`	`ab = ab + EXP(a)*xb`

Tapenade does/doesn't

Tapenade does handle

- modules, overloading, renaming, interfaces
- structured types ("records")
- pointers and allocation

Tapenade does not handle

- fpp or cpp keys, templates
- deallocation in reverse more
- checkpointing of non-reentrant code
- classes and objects

Outline

Tools for source-transformation AD

http://www.autodiff.org

AD tools are based on
overloading or **source transformation**.

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/OPEN-AD : Transformation ; tangent, reverse?, Store-All + Checkpointing
- TAPENADE : Transformation ; tangent, reverse, Store-All + Checkpointing
- TAF : Transformation ; 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,...)

Outline

Validation methods

From a program P that evaluates

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

tangent AD creates

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

and reverse AD creates

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

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 :

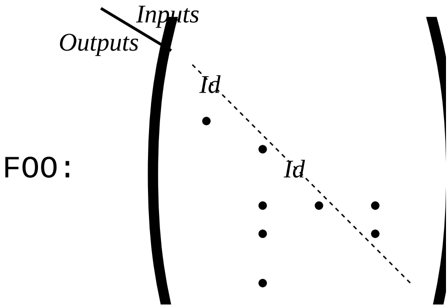
$$
\begin{aligned}
(\overline{X} \cdot \dot{X}) &= (F'^t(X) \times \dot{Y} \cdot \dot{X}) \\
&= \dot{Y}^t \times F'(X) \times \dot{X} \\
&= \dot{Y}^t \times \dot{Y} \\
&= (\dot{Y} \cdot \dot{Y})
\end{aligned}
$$

Often called the "dot-product test"

Outline

Black-box routines

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



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

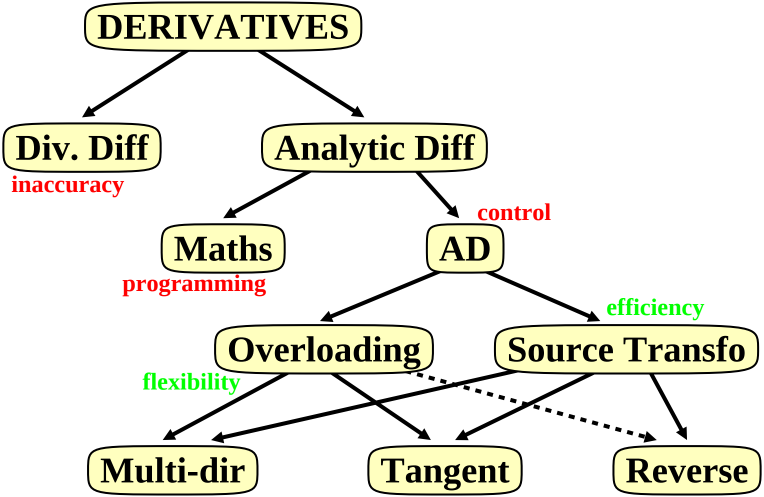
If a loop has independent iterations, possibly terminated by a sum-reduction, then

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

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 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.

Thank you for your attention !