Algorithmic Differentiation (by Source Transformation): achievements and challenges

Laurent Hascoët

INRIA Sophia-Antipolis, France

GDR Calcul, Jan 24, 2019

This is (Source-Transformation) 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 (Source-Transformation) AD

```
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
```

Inserts differentiated instructions into FOO, automatically

Computes derivatives with machine accuracy

Outline

Formalization: programs are functions

See any (straight-line piece of) program $\mathtt{P}:\{I_1; I_2; \dots I_p;\}$ as:

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

Define for short:

$$V_0 = \textbf{in} \;\text{ and }\; V_k = f_k(V_{k-1})$$

The chain rule yields:

$$f'(\textbf{in}) = f'_p(V_{p-1}).f'_{p-1}(V_{p-2}).\dots.f'_1(V_0)$$

In which order shall we multiply all these matrices?

Evaluate from the right or from the left?

We may start from the right (i.e. the inputs **in**) $\Rightarrow$ Tangent
$\Rightarrow$ start with a direction vector $\dot{\mathbf{in}}$, then progress leftwards:

$$\dot{\mathbf{out}} = f'(\mathbf{in}).\dot{\mathbf{in}} = f'_p(V_{p-1}).f'_{p-1}(V_{p-2})\dots f'_1(V_0).\dot{\mathbf{in}}$$

We may start from the left (i.e. the inputs **out**) $\Rightarrow$ Adjoint
$\Rightarrow$ start with an weighting vector $\overline{\mathbf{out}}$, then progress rightwards:

$$\overline{\mathbf{in}} = \overline{\mathbf{out}}.f'(\mathbf{in}) = \overline{\mathbf{out}}.f'_p(V_{p-1}).f'_{p-1}(V_{p-2})\dots f'_1(V_0)$$

(for the full Jacobian, replace the start vectors by identity matrices)

Take the time to figure out the sizes and costs wrt sizes m and n

Same idea, different words

A (straight-line) program computes **out** from **in**:

$$\textbf{in} \longrightarrow \mathtt{v1} \longrightarrow \mathtt{v2} \longrightarrow \cdots \longrightarrow \mathtt{v9} \longrightarrow \textbf{out}$$

One can propagate $\frac{d\mathtt{v}}{d\textbf{in}}$ forward ⇒ Tangent:

$$1.0 = \frac{\mathrm{d}\,\textbf{in}}{\mathrm{d}\,\textbf{in}} \longrightarrow \frac{\mathrm{d}\,\mathtt{v1}}{\mathrm{d}\,\textbf{in}} \longrightarrow \frac{\mathrm{d}\,\mathtt{v2}}{\mathrm{d}\,\textbf{in}} \longrightarrow \cdots \qquad \frac{\mathrm{d}\,\textbf{out}}{\mathrm{d}\,\textbf{in}}$$

One can propagate $\frac{d\textbf{out}}{d\mathtt{v}}$ backward ⇒ Adjoint:

$$\frac{\mathrm{d}\,\textbf{out}}{\mathrm{d}\,\textbf{in}} \qquad \cdots \longleftarrow \frac{\mathrm{d}\,\textbf{out}}{\mathrm{d}\,\mathtt{v8}} \longleftarrow \frac{\mathrm{d}\,\textbf{out}}{\mathrm{d}\,\mathtt{v9}} \longleftarrow \frac{\mathrm{d}\,\textbf{out}}{\mathrm{d}\,\textbf{out}} = 1.0$$

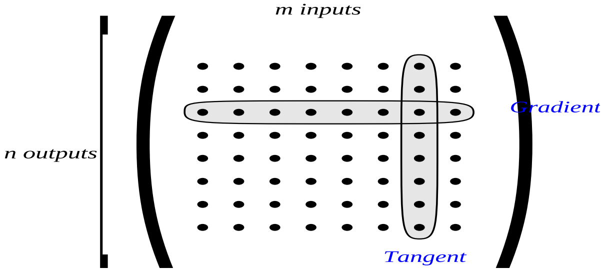
Same result, different cost:

...depending of the sizes of **in** and **out**

Full Jacobian with Tangent or Adjoint AD

$f \quad : \quad \mathbf{in} \in R^m \;\; \rightarrow \;\; \mathbf{out} \in R^n$

$\dfrac{\mathbf{d\,out}}{\mathbf{d\,in}} =$



- $\dfrac{\mathbf{d\,out}}{\mathbf{d\,in}}$ costs $m * 4? * \mathrm{P}$ using the tangent mode
  Good if $m <= n$
- $\dfrac{\mathbf{d\,out}}{\mathbf{d\,in}}$ costs $n * 4? * \mathrm{P}$ using the adjoint mode
  Good if $m >> n$ (e.g $n = 1$ for a gradient)

By the way: beware of control

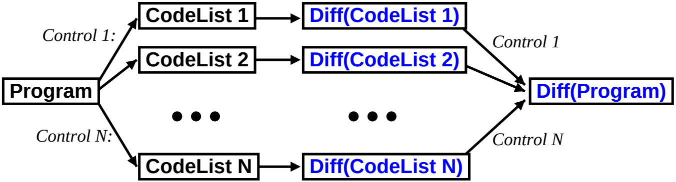
Function f must be differentiable,

but implementation may require control $\Rightarrow$ creates non-differentiability !

Freeze the current control:

$\Rightarrow$ the program becomes a simple sequence of instructions

$\Rightarrow$ AD differentiates these sequences:



$\Rightarrow$ and replaces them into the control.

Caution: the diff program is only a piecewise diff !

$\Rightarrow$ see [Griewank] about the *Abs-Normal-Form*

Adjoint AD in a nutshell

Adjoint derivatives by Algorithmic Differentiation (AD):

- compute gradients of numerical models,
- from the models source program,
- more or less automatically,
- at a cost independant of #inputs,

...but there are serious challenges

Implementing Tangent AD

$$\dot{\mathbf{out}} = f'(\mathbf{in}).\dot{\mathbf{in}} = f'_p(V_{p-1}).f'_{p-1}(V_{p-2})\ldots f'_1(V_0).\dot{\mathbf{in}}$$

Implementation:

$$\begin{matrix}\dot{\mathbf{in}}\\ \mathbf{in}\end{matrix} \xrightarrow{\dot{I_1}} \xrightarrow{I_1} \xrightarrow{\dot{I_2}} \xrightarrow{I_2} \xrightarrow{\dot{I_3}} \xrightarrow{I_3} \quad \ldots \quad \xrightarrow{\dot{I}_{p\text{-}2}} \xrightarrow{I_{p\text{-}2}} \xrightarrow{\dot{I}_{p\text{-}1}} \xrightarrow{I_{p\text{-}1}} \xrightarrow{\dot{I_p}} \begin{matrix}\dot{\mathbf{out}}\\ \mathbf{out}\end{matrix}$$

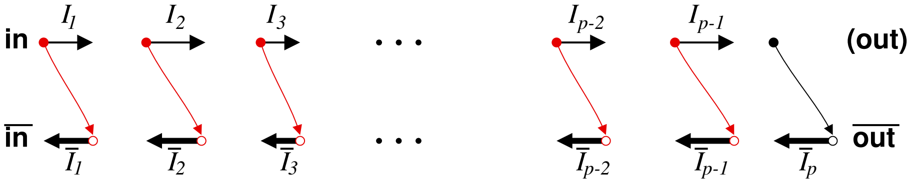
Tangent-diff instructions interleaved with the original instructions.

almost no problem...

Implementing Adjoint AD

$$\overline{\mathbf{in}} = \overline{\mathbf{out}}.f'(\mathbf{in}) = \overline{\mathbf{out}}.f'_p(V_{p-1}).f'_{p-1}(V_{p-2})\ldots f'_1(V_0)$$

Implementation:



Adjoint-diff instructions form the backward sweep.
There is a forward sweep and then the backward sweep.
Mechanism required to make the V_k available in reverse order.

This is hard, but it is worth the effort

By the way: Adjoint code is weird

Consider instruction I_k: `c := a*b` i.e. function:

$$
\begin{array}{cccc}
f_k : & \boldsymbol{R}^3 & \rightarrow & \boldsymbol{R}^3 \\
& \begin{pmatrix} \mathtt{a} \\ \mathtt{b} \\ \mathtt{c} \end{pmatrix} & \mapsto & \begin{pmatrix} \mathtt{a} \\ \mathtt{b} \\ \mathtt{a} * \mathtt{b} \end{pmatrix}
\end{array}
$$

Its adjoint code must compute:

$$
\begin{pmatrix} \overline{\mathtt{a}} & \overline{\mathtt{b}} & \overline{\mathtt{c}} \end{pmatrix} := \begin{pmatrix} \overline{\mathtt{a}} & \overline{\mathtt{b}} & \overline{\mathtt{c}} \end{pmatrix} \times f_k' == \begin{pmatrix} \overline{\mathtt{a}} & \overline{\mathtt{b}} & \overline{\mathtt{c}} \end{pmatrix} \times \begin{pmatrix} 1 & & \\ & 1 & \\ \mathtt{b} & \mathtt{a} & 0 \end{pmatrix}
$$

And therefore its adjoint "code" is:

$$
\begin{array}{l}
\overline{\mathtt{a}} := \overline{\mathtt{a}} + \mathtt{b} * \overline{\mathtt{c}} \\
\overline{\mathtt{b}} := \overline{\mathtt{b}} + \mathtt{a} * \overline{\mathtt{c}} \\
\overline{\mathtt{c}} := 0.0
\end{array}
$$

This is not a problem: all you need is a tool

By the way: why the name "Adjoint AD"?

Code instructions can be seen as equality constraints *[Giles, Pironneau]*.

```
a := i1
b := i2
c := a*b
d := a*c
r := c + d
```

Adjoint AD $\rightarrow$

$\overline{d} := \overline{r}$
$\overline{c} := \overline{r} + a*\overline{d}$
$\overline{b} := a*\overline{c}$
$\overline{a} := c*\overline{d} + b*\overline{c}$

$\downarrow$?Lagrangian?

$$\mathcal{L} = \overline{r}(c+d-r)+\overline{d}(ac-d)+\overline{c}(ab-c)+\overline{b}(i_2-b)+\overline{a}(i_1-a)$$

$\downarrow$

$$\frac{d\mathcal{L}}{d\mathrm{d}} = 0 = \overline{r}-\overline{d}$$
$$\frac{d\mathcal{L}}{d\mathrm{c}} = 0 = \overline{r}+a\overline{d}-\overline{c}$$
$$\frac{d\mathcal{L}}{d\mathrm{b}} = 0 = a\overline{c}-\overline{b}$$
$$\frac{d\mathcal{L}}{d\mathrm{a}} = 0 = c\overline{d}+b\overline{c}-\overline{a}$$

$\rightarrow$

$\overline{d} := \overline{r}$
$\overline{c} := \overline{r} + a*\overline{d}$
$\overline{b} := a*\overline{c}$
$\overline{a} := c*\overline{d} + b*\overline{c}$

Outline

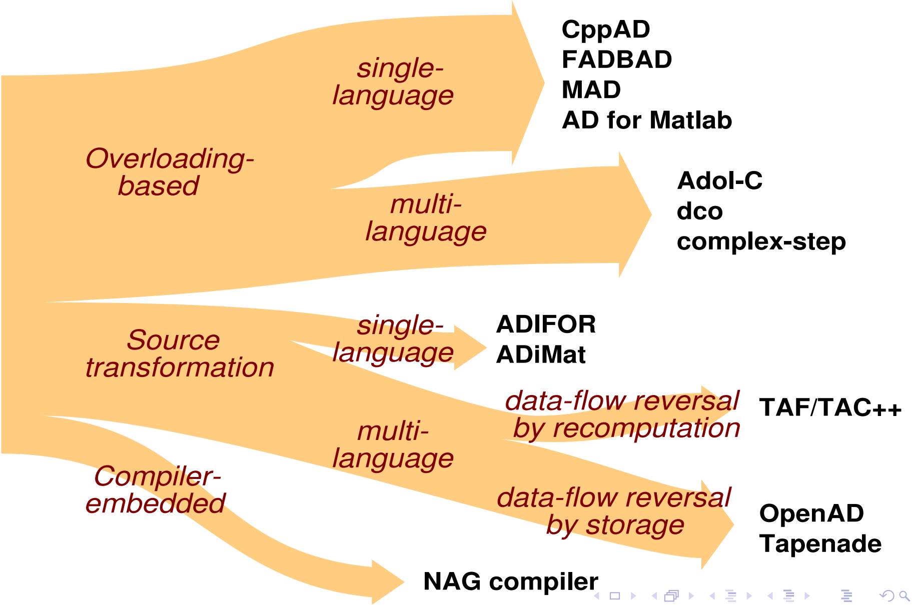
AD by Overloading, AD by Source-Transformation

Roughly, AD tools are based either on Source-Transformation, or on Operator-Overloading.

Overloading (available in F90, Object languages, ...) lets one redefine arithmetic operations to compute derivatives on the fly:
Change active `float, real` to `aDouble`, and link with a library that

- for Tangent: computes derivatives on `aDouble`'s
- for Adjoint: stores instructions on a "tape", for later backward derivative computation

A taxonomy of AD tools



In the sequel we are mostly concerned with
Source-Tranformation AD

Wait for Uwe's talk for details on
Operator-Overloading AD

Outline

Challenges of adjoint AD

Gradients are propagated backwards,
using info from the (forward) primal code
$\Rightarrow$ Instruction flow reversal
$\Rightarrow$ Data flow reversal

There are many other challenges around AD:

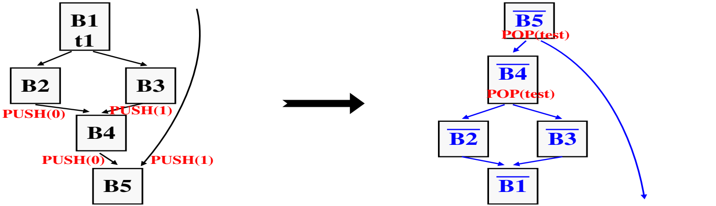
- non-smoothness *[Griewank et al.]*
- stochastic or chaotic parts *[Wang]*
- higher derivatives (cost, size...) *[Walther, Wang, Pothen]*
- ...

Adjoint first difficulty: instruction flow reversal

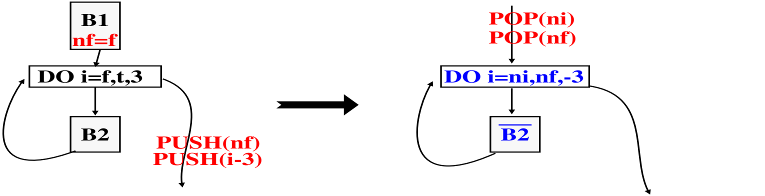
- Differentiated instructions follow the inverse of P's original control flow.
- The forward sweep must record its control-flow choices
- The backward sweep must use the recorded choices
- ...and all this must remain cheap

Instruction flow reversal with bookkeeping

The key is to store flow decisions at merging point:



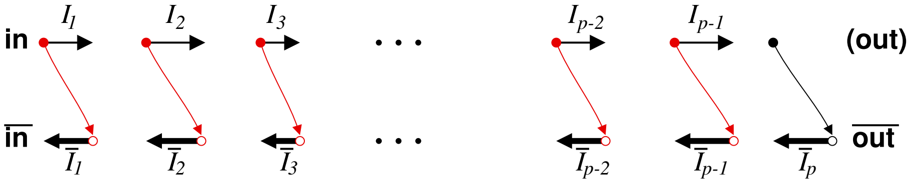
The same applies to loops and any other construct:



Works with a stack. Memory cost is negligible.

Adjoint second difficulty: data flow reversal

$$\overline{\textbf{in}} = f'^t(\textbf{in}).\overline{\textbf{out}} = f_1'^t(V_0)\ \ldots\ f_{p-1}'^t(V_{p-2})\ .\ f_p'^t(V_{p-1})\ .\overline{\textbf{out}}$$



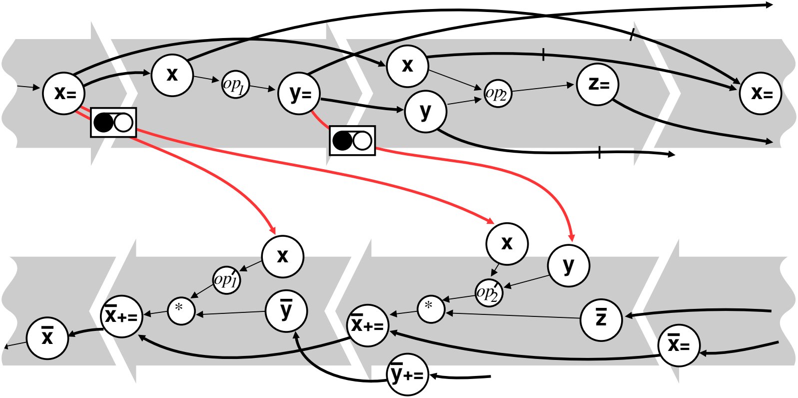
In most codes, V_0, V_1,... V_{p-1} successively overwrite one another. Most likely V_{p-2} is lost, overwritten by I_{p-1}, etc.

One can either store (our basic choice), or recompute
In practice, one always ends up using both!

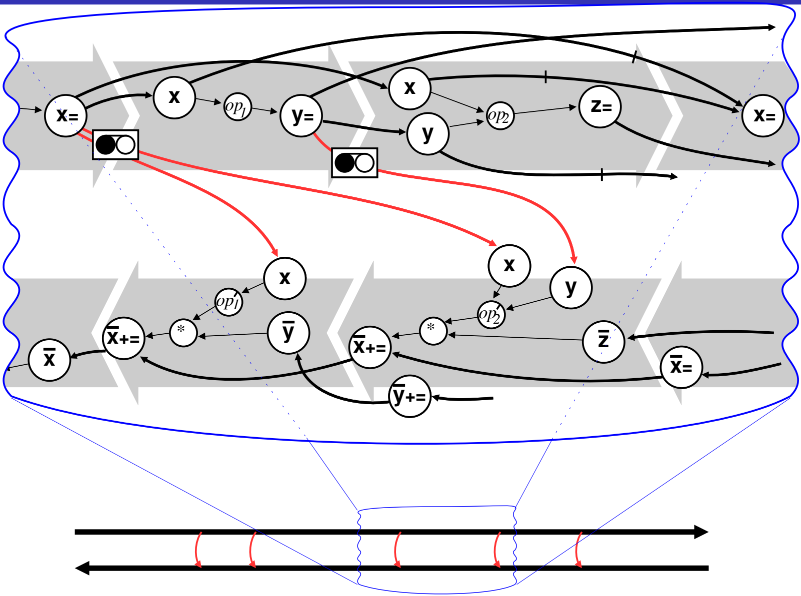
In the sequel, data-flow reversal is based on storage
Recomputation only comes as an extra

See tool TAF/TAC++ for
data-flow reversal by recomputation

Store forwards; Retrieve backwards



Store forwards; Retrieve backwards



The memory challenge

The memory cost of storing intermediate values grows linearly with runtime.

Can we master memory consumption ?

- use every possible Data-Flow analysis
  $\rightarrow$ can gain 40 to 70%... still linear memory cost
- trade recomputation/storage ("Checkpointing")
  $\rightarrow$ achieves logarithmic growth
- exploit profitable situations, (math or algorithm) e.g.
  - Linear solvers
  - Parallel loops
  - Fixed-Point iterations

Outline

Data-Flow Analysis

Naïve application of the adjoint AD model would

- execute all primal instructions
- store every value before it is overwritten
- execute the complete adjoint of each instruction

Forward constant propagation & backward slicing, specialized for the particular structure of adjoint codes

Use static data-flow analysis (classic $+$ and $-$), on the primal code, then produce an optimized adjoint code

4 classic AD Data-Flow analyses

- **varied**: *[Fagan, Carle]*

  if current v depends on no "independent input", then $\overline{\mathrm{v}}$ is useless

  $\Rightarrow$ slice out computation of $\overline{\mathrm{v}}$

- **useful**:

  if current v influences no "dependent output", then $\overline{\mathrm{v}}$ is zero

  $\Rightarrow$ propagate constant $\overline{\mathrm{v}}$ and remove its initialization

- **diff-live**:

  if current v influences no useful derivative (may influence orig. result)

  $\Rightarrow$ slice out computation of v

- **TBR**: *[Naumann]*

  if current v not used in any derivative (e.g. only linear uses of v)

  $\Rightarrow$ slice out storage of v before it is overwritten

Discussion

- These are just special cases of classic code optim.
- Agressive compiler optim *[Pearlmutter, Siskind]* may be more systematic ($\Rightarrow$ are we missing adjoint data-flow analyses?)
- ... but there's a limit to the window of code that the compiler can examine, whereas fwd and bwd code are arbitrarily far apart
- Adjoint data-flow analyses use structural knowledge of adjoint codes, and run on the primal code. E.g.

$$\mathbf{TBR}^{+}(I) = \begin{cases} (\mathbf{TBR}^{-}(I) \cup \mathbf{use}(I')) \setminus \mathbf{kill}(I) & \text{if } I \text{ live} \\ \mathbf{TBR}^{-}(I) \cup \mathbf{use}(I') & \text{otherwise} \end{cases}$$

Diff-Live, TBR, Recompute

naïve	Diff-live	TBR	Recompute
CALL PUSHINTEGER4(n)	CALL PUSHINTEGER4(n)		
n = ind1(i)	n = ind1(i)	n = ind1(i)	n = ind1(i)
CALL PUSHREAL4(b(n))	CALL PUSHREAL4(b(n))		
b(n)=SIN(a(n))-b(n)	b(n)=SIN(a(n))-b(n)	b(n)=SIN(a(n))-b(n)	b(n)=SIN(a(n))-b(n)
CALL PUSHREAL4(a(n))	CALL PUSHREAL4(a(n))	CALL PUSHREAL4(a(n))	CALL PUSHREAL4(a(n))
a(n) = a(n) + x	a(n) = a(n) + x	a(n) = a(n) + x	a(n) = a(n) + x
CALL PUSHREAL4(c)			
c = a(n)*b(n)			
CALL PUSHREAL4(a(n))			
a(n) = a(n)*a(n+1)			
CALL PUSHINTEGER4(n)	CALL PUSHINTEGER4(n)	CALL PUSHINTEGER4(n)	
n = ind2(i+2)	n = ind2(i+2)	n = ind2(i+2)	n = ind2(i+2)
CALL PUSHREAL4(z(n))			
z(n) = z(n) + c			
CALL POPREAL4(z(n))			
cb = zb(n)	cb = zb(n)	cb = zb(n)	cb = zb(n)
CALL POPINTEGER4(n)	CALL POPINTEGER4(n)	CALL POPINTEGER4(n)	n = ind1(i)
CALL POPREAL4(a(n))			
ab(n+1) = ab(n+1)	ab(n+1) = ab(n+1)	ab(n+1) = ab(n+1)	ab(n+1) = ab(n+1)
+a(n)*ab(n)	+a(n)*ab(n)	+a(n)*ab(n)	+a(n)*ab(n)
ab(n) = b(n)*cb	ab(n) = b(n)*cb	ab(n) = b(n)*cb	ab(n) = b(n)*cb
+a(n+1)*ab(n)	+a(n+1)*ab(n)	+a(n+1)*ab(n)	+a(n+1)*ab(n)
CALL POPREAL4(c)			
bb(n) = bb(n)	bb(n) = bb(n)	bb(n) = bb(n)	bb(n) = bb(n)
+a(n)*cb	+a(n)*cb	+a(n)*cb	+a(n)*cb
CALL POPREAL4(a(n))	CALL POPREAL4(a(n))	CALL POPREAL4(a(n))	CALL POPREAL4(a(n))
xb = xb + ab(n)	xb = xb + ab(n)	xb = xb + ab(n)	xb = xb + ab(n)
CALL POPREAL4(b(n))	CALL POPREAL4(b(n))		
ab(n) = ab(n)	ab(n) = ab(n)	ab(n) = ab(n)	ab(n) = ab(n)
+COS(a(n))*bb(n)	+COS(a(n))*bb(n)	+COS(a(n))*bb(n)	+COS(a(n))*bb(n)

Summary: good, but not sufficient

Adjoint data-flow analyses

- are classical compiler analyses/optims specialized for adjoint codes.
- bring substantial benefit
  - 20% to 50% in runtime
  - 40% to 70% in memory space

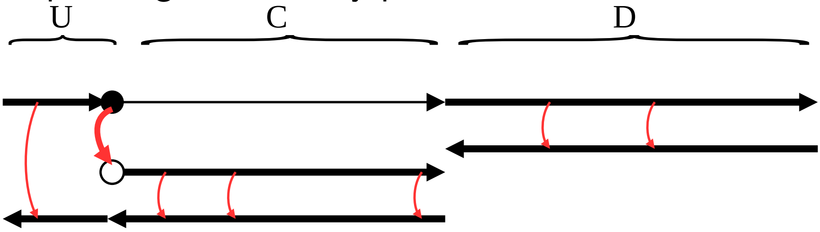
But memory still grows linearly with runtime

$\Rightarrow$ we need something else...

Outline

Trading recomputation (CPU) for storage (memory)

Checkpointing: elementary pattern

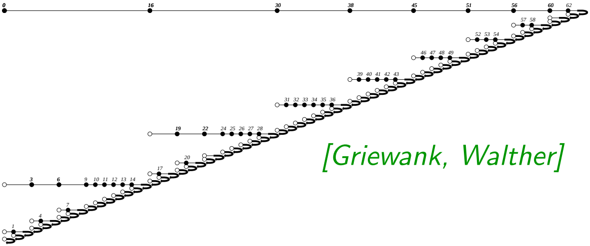


- reduces peak storage
- at the cost of duplicate execution
- also costs a memory "Snapshot", small enough:

$$\text{Snapshot} \subset \textbf{use}(\overline{\mathtt{C}}) \cap \left(\textbf{out}(\mathtt{C}) \cup \textbf{out}(\overline{\mathtt{D}})\right)$$

Nesting checkpoints

Checkpoints must be (carefully) nested.
Optimal nesting (binomial) exists for time-stepping loops:

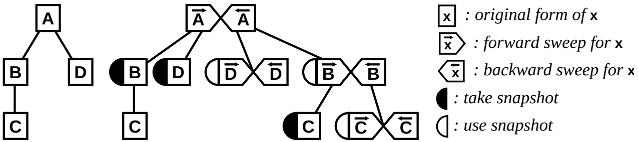


[Griewank, Walther]

- peak memory storage grows like log(runtime)
  execution duplication grows like log(runtime)
- in real life, storage is fixed to q snapshots,
  execution duplication grows like qth-root(runtime)

Checkpointing on calls

Nested checkpointing can be applied on procedure calls:



Not optimal(?), but still logarithmic if call tree is balanced.

Applies also to code sections that *could* be procedures.

A few limitations

- Checkpoints must respect code structure:
  - no checkpoint across procedures
  - no checkpoint across structured statements
  - *...well you could, but you need a flattened instruction tape*
- Checkpoints must contain both ends of system resources lifespan:

  `read/write`, `alloc/free`, `send/recv`, `isend/wait`...
- Checkpointed code must be reentrant

All in all, nested checkpointing is the answer

Outline

Profitable Situations

Take advantage of algorithmic or mathematic knowledge on parts of the code.

A selection:

- Adjoint of Linear Solvers
- Adjoint of Parallel Loops
- Adjoint of Fixed-Point iterations

Adjoint of Linear Solvers

Avoid differentiation inside the source of linear solvers

⇒ write their adjoint by hand, calling the solver itself!

[Giles]

```
SOLVE_B(A,Ab,y,yb,b,bb) {
  At = TRANSPOSE(A)
  SOLVE(At,tmp,yb)
  bb[:]  = bb[:]  + tmp[:]
  SOLVE(A,y,b)
  for each i and each j {
    Ab[i,j] = Ab[i,j] - y[j]*tmp[i]
  }
  yb[:]  = 0.0
}
```

Data-Dependence Graph of Adjoints

Data-Dependence Graph is key to loop rescheduling.
Fewer arrows in the DDG $\Rightarrow$ more rescheduling allowed.

- (classical) No DDG arrow between successive `read`s of a variable.
- No DDG arrow either between successive `increment`s of a variable. (assuming increments are atomic, or assuming memory is not shared)
- The adjoint of a `read(x)` is an $\texttt{increment(}\bar{\texttt{x}}\texttt{)}$
- The adjoint of an `increment(x)` is a $\texttt{read(}\bar{\texttt{x}}\texttt{)}$

The DDG of the backward sweep is a subset of the DDG of the primal code, only with arrows reversed

Therefore adjoint AD preserves most parallel properties!

Application to Parallel Loops

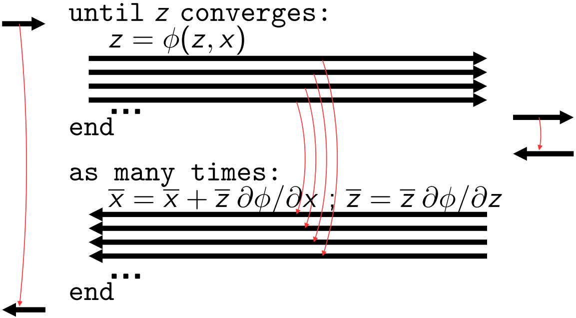
// Parallel loop:
```
for (i=0 ; i<=N ; ++i) {
  forward sweep iteration i
}
for (i=N ; i>=0 ; --i) {
  backward sweep iteration i
}
```

Loop #2 is parallel: reverse iterations, fuse with loop #1:

```
for (i=0 ; i<=N ; ++i) {
  forward sweep iteration i
  backward sweep iteration i
}
```

⇒ Reduces peak memory usage dramatically!

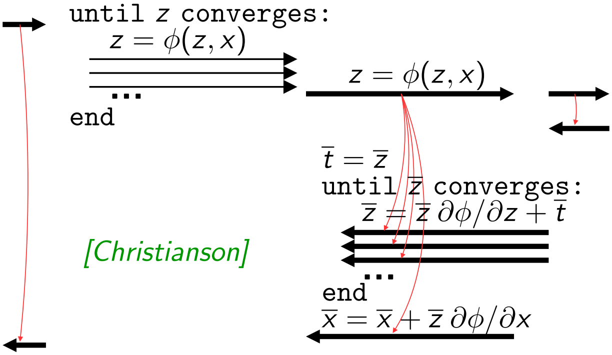
Adjoint of Fixed-Point iterations



You should not do that!

- all values from intermediate iterations are stored
- poor convergence guarantees of the adjoint sweep

Two-Phases Adjoint



- Only the converged primal iteration is stored, then is used several times.
- The adjoint iteration has its own convergence control
- Converges in one step if primal has quadratic convergence

Outline

Validate Tangent first!

For any function/code F, with Jacobian J:

- For any $\dot{X}$, tangent code returns $\dot{Y} = J \times \dot{X}$
- For any $\dot{X}$, $\dot{Y}$ is also the limit:

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

So we can approximate $\dot{Y}$ by running P twice, at points X and $X + \varepsilon \dot{X}$ for a small ε.

Validate Adjoint wrt Tangent

- For any $\dot{X}$, tangent code returns $\dot{Y} = J \times \dot{X}$
- For any $\overline{Y}$, adjoint code returns $\overline{X} = \overline{Y} \times J$

Observe that $\overline{X} \times \dot{X} = \overline{Y} \times J \times \dot{X} = \overline{Y} \times \dot{Y}$

If the adjoint code is correct, then the above must hold for any $\dot{X}$ and any $\overline{Y}$.

Moreover, at any "point" of the code, calling W the set of all active variables at that point:

$$\overline{X} \times \dot{X} = \overline{W} \times \dot{W} = \overline{Y} \times \dot{Y}$$

Outline

Reading and writing variables

The adjoint of a use is an increment
The adjoint of an increment is a use

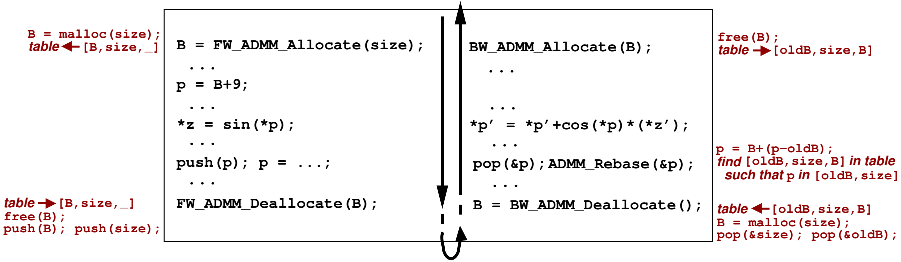
primal	**adjoint**
`... = ... x ...`	`xb = xb + ...`
`s = s + 2.1*x`	`xb = xb + 2.1*sb`

Assuming increments are atomic, they are independent
⇒ The adjoint of a parallel loop is (almost) a parallel loop

Dynamic memory

The adjoint of a `malloc` is a `free`

The adjoint of a `free` is a `malloc`



Parallel collective operations

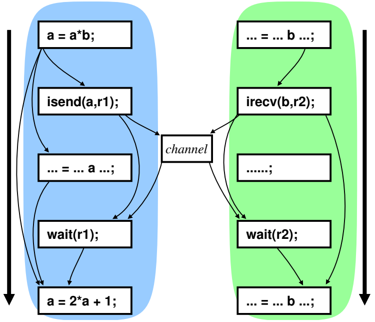
The adjoint of a sum is a spread
The adjoint of a spread is a sum

The adjoint of a `MPI_Bcast` is a (SUM)`MPI_Reduce`
The adjoint of a (SUM)`MPI_Reduce` is a `MPI_Bcast`
The adjoint of a `MPI_Gather` is a `MPI_Scatter`
The adjoint of a `MPI_Scatter` is a `MPI_Gather`

Message Passing

The adjoint of a SEND is a RECEIVE
The adjoint of a RECEIVE is a SEND

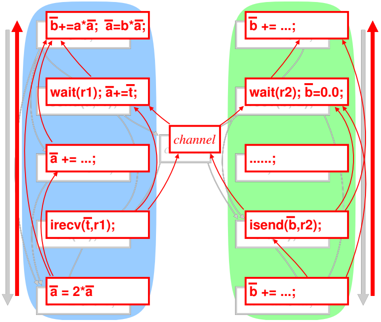
The adjoint of a `MPI_Isend/MPI_Wait` is a `MPI_Irecv/MPI_Wait`
The adjoint of a `MPI_Irecv/MPI_Wait` is a `MPI_Isend/MPI_Wait`



Message Passing

The adjoint of a SEND is a RECEIVE
The adjoint of a RECEIVE is a SEND

The adjoint of a `MPI_Isend/MPI_Wait` is a `MPI_Irecv/MPI_Wait`
The adjoint of a `MPI_Irecv/MPI_Wait` is a `MPI_Isend/MPI_Wait`



⇒ Good news: adjoint AD introduces no deadlock

Outline

Tapenade

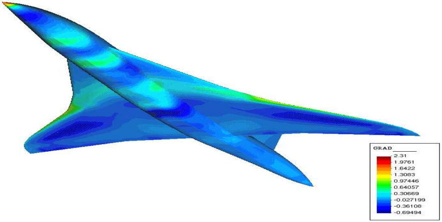
- Tapenade is the AD tool that our team develops.
- Source-Transformation, data-flow reversal by storage, association-by-name
- **Tangent** and **Adjoint** AD, on **Fortran** (77 to current) and **C** (ANSI)
- Classically used from the command-line:
  `$> tapenade -b -head "mod1.foo(d)/(b x y)" file1.f90 file2.f90 aux.f` ...<*options*>
- Free for academic use
- Decent popularity ... despite limitations and bugs

In the sequel,
applications images, performance measurements...
are made with Tapenade

Outline

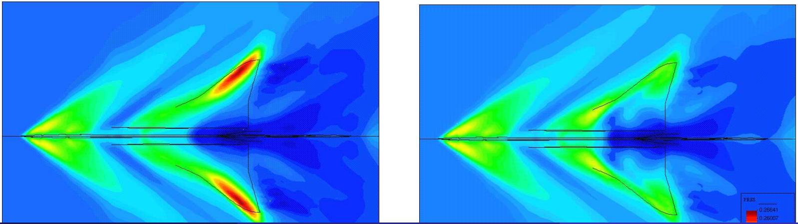
CFD optimization

AD gradient of the cost function (sonic boom under) on the skin geometry:



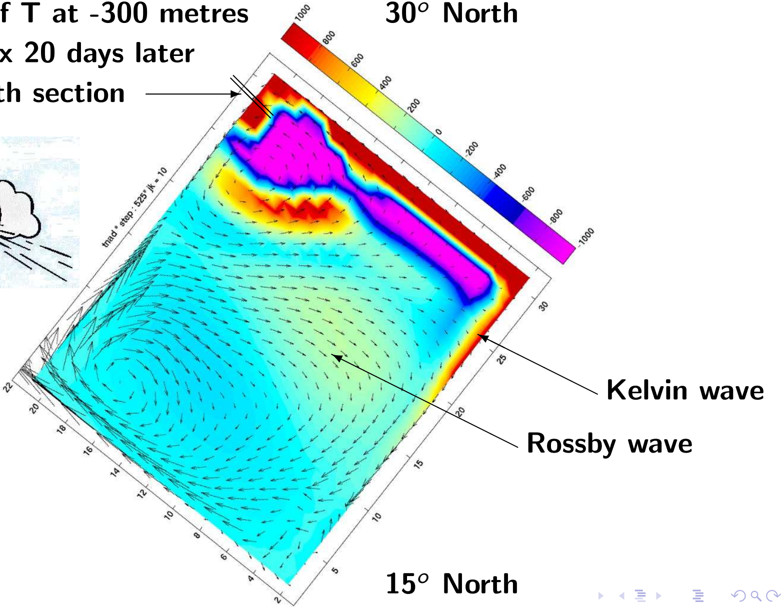
(Dassault Aviation)

Sonic boom under the plane after 8 optimization cycles:

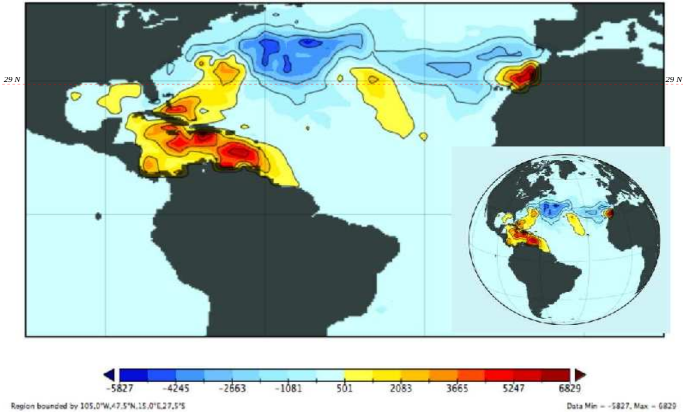


Data Assimilation (OPA 9.0/GYRE)

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



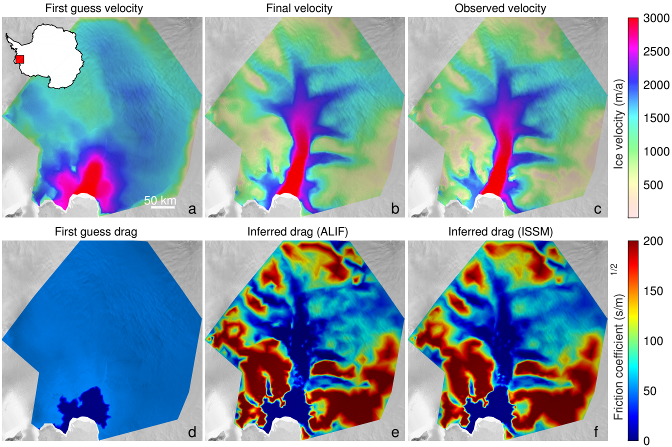
Data Assimilation (OPA 9.0/NEMO)



2^o grid cells, one year simulation

Inverse problem (ALIF/ISSM)

Infer the basal drag glacier/ground
by minimizing discrepancy on surface velocity



Performance statistics

	$n \rightarrow m$	tangent A_t	R_t	adjoint A_a	R_a	peak (Mb)	traffic (Mb)
uns2d *(2,000*F77)*	$14000 \rightarrow 3$	3.4	**2.4**	15.1	**5.9**	**241**	1243
nsc2ke *(3,500*F77)*	$1602 \rightarrow 5607$	1.9	**2.4**	4.5	**16.2**	**168**	2806
lidar *(330*F90)*	$37 \rightarrow 37$	6.7	**1.1**	14.4	**2.0**	**11**	11
nemo *(55,000*F90)*	$9100 \rightarrow 1$	3.0	**2.0**	8.1	**6.5**	**1591**	85203
gyre *(21,000*F90)*	$21824 \rightarrow 1$	4.5	**1.9**	13.3	**7.9**	**481**	48602
winnie *(3,700*F90)*	$3 \rightarrow 1$	1.4	**1.7**	13.7	**5.9**	**421**	614
stics *(17,000*F77)*	$739 \rightarrow 1467$	8.6	**2.4**	15.3	**3.9**	**155**	186
smac-sail *(3,500*F77)*	$1321 \rightarrow 7801$	5.9	**1.0**	10.5	**3.1**	**2**	21
traces *(19,800*F90)*	$8 \rightarrow 1$	4.0	**1.3**	12.9	**3.8**	**159**	4390
mit-gcm *(258,225*F77)*	$4704 \rightarrow 1$	8.5	**2.0**	14.5	**6.6**	**260**	5709
alif *(6,755*C)*	$1413 \rightarrow 1$	6.0	**1.6**	14.0	**4.3**	**729**	

Conclusion

- AD is now a mature technology
- If your function is implemented, consider AD
- Adjoint AD still requires more effort, but it's worth it
- Many researchers are building excellent AD tools, for you

Enjoy today's presentations !

Tapenade validation and debug

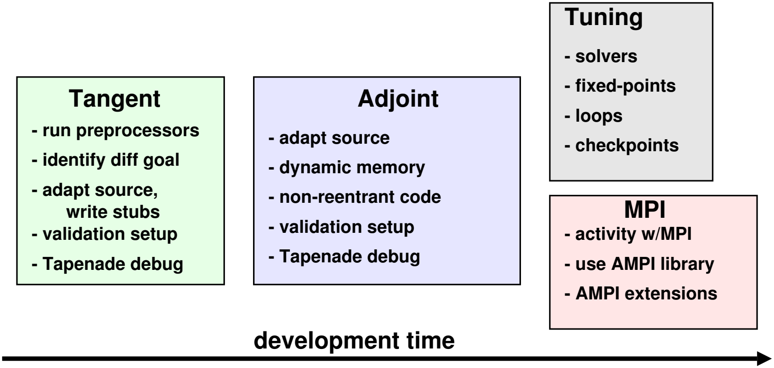
Automated validation:

- `-context` generates a context code to run diff code, to validate TGT against DD, and to validate ADJ against TGT.

When AD goes wrong:

- `-debugTGT, -debugADJ` insert debugging primitives at strategic places.
- `-nooptim NAME` turns off the AD optimization named `NAME`, for a less efficient but maybe more robust diff code.

Phases of an AD project



- 3 to 4 phases,
- mostly sequential,
- needs interaction with AD tool developers...

Overloading AD: pros and cons

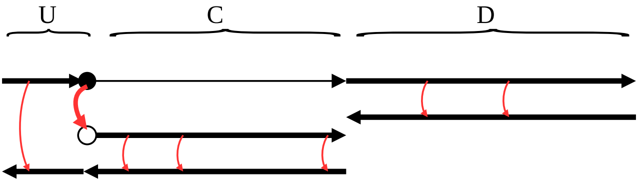
+	-
light-weight, versatile	(mildly)hand-modified source
adapts to exotic control and constructs	overloading required, restricted data-flow analysis no global analysis
higher-order, Taylor, intervals	not-so-efficient adjoints (trajectory storage on tape)

Splitting and merging differentiated instructions

- Split common subexpressions in derivatives
- Merge unnecessary intermediate derivatives

<table>
<tr><th>naïve adjoint</th><th>split and merge</th></tr>
<tr><td><pre>resb = v(j)*gb(i, j)
vb(j) = vb(j) + res*gb(i, j)
gb(i, j) = 0.0
taub = taub
   +(z(j)-2.0)*g(i, j)*resb/v(j)
wb(i, j) = wb(i, j)
   -g(i, j)*(z(j)-2.0)*resb/v(j)
gb(i, j) = gb(i, j)
   +(z(j)-2.0)*(tau-w(i, j))*resb/v(j)
zb(j) = zb(j)
   +(tau-w(i, j))*g(i, j)*resb/v(j)
vb(j) = vb(j)
   -(tau-w(i, j))*g(i, j)*(z(j)-2.0)*resb/v(j)**2</pre></td><td><pre>resb = v(j)*gb(i, j)
temp = (z(j)-2.0)/v(j)
tempb0 = temp*g(i, j)*resb
tempb = (tau-w(i, j))
        *g(i, j)*resb/v(j)
vb(j) = vb(j)
   +res*gb(i, j) -temp*tempb
gb(i, j) = temp
        *(tau-w(i, j))*resb
taub = taub + tempb0
wb(i, j) = wb(i, j) - tempb0
zb(j) = zb(j) + tempb</pre></td></tr>
</table>

By the way: Combining Checkpointing and TBR



- The Snapshot may take care of TBR coming from $\mathtt{U}$
- The TBR sent to $\mathtt{D}$ can take care of the Snapshot

A range of "optimal" combinations exist.
E.g., given **tbr**$\mathtt{U}$ coming from $\mathtt{U}$, "lazy" snapshot:

- Snapshot $= \textbf{out}(\mathtt{C}) \cap \left(\textbf{use}(\overline{\mathtt{C}}) \cup \textbf{tbr}\mathtt{U}\right)$
- **tbr** to $\mathtt{D}$ $= \left(\textbf{use}(\overline{\mathtt{C}}) \cup \textbf{tbr}\mathtt{U}\right) \setminus \textbf{out}(\mathtt{C})$
- **tbr** to $\mathtt{C}$ $= \textbf{tbr}\mathtt{U}$