Algorithmic Differentiation
(by Source Transformation):
achievements and challenges

Laurent Hascoët

INRIA Sophia-Antipolis, France

GDR Calcul, Jan 24, 2019
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
See any (straight-line piece of) program $P: \{I_1; I_2; \ldots I_p; \}$ as:

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

Define for short:

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

The chain rule yields:

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

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 \textbf{in}) \Rightarrow \textbf{Tangent}
\Rightarrow start with a direction vector \textbf{in}, then progress leftwards:

\[ \dot{\textbf{out}} = f'(\textbf{in}) \cdot \dot{\textbf{in}} = f'_p(V_{p-1}) \cdot f'_{p-1}(V_{p-2}) \ldots f'_1(V_0) \cdot \dot{\textbf{in}} \]

We may start from the left (i.e. the inputs \textbf{out}) \Rightarrow \textbf{Adjoint}
\Rightarrow start with a weighting vector \textbf{out}, then progress rightwards:

\[ \overline{\textbf{in}} = \overline{\textbf{out}} \cdot f'(\textbf{in}) = \overline{\textbf{out}} \cdot f'_p(V_{p-1}) \cdot f'_{p-1}(V_{p-2}) \ldots f'_1(V_0) \]

(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**:

\[
\begin{align*}
\text{in} & \rightarrow v1 & \rightarrow v2 & \rightarrow \ldots & \rightarrow v9 & \rightarrow \text{out} \\
\end{align*}
\]

One can propagate \( \frac{dv}{din} \) forward \( \Rightarrow \) **Tangent**:

\[
1.0 = \frac{d\text{in}}{d\text{in}} \rightarrow \frac{dv1}{d\text{in}} \rightarrow \frac{dv2}{d\text{in}} \rightarrow \ldots
\]

One can propagate \( \frac{dout}{dv} \) backward \( \Rightarrow \) **Adjoint**:

\[
\begin{align*}
\frac{d\text{out}}{d\text{in}} & \rightarrow \frac{d\text{out}}{dv8} & \rightarrow \frac{d\text{out}}{dv9} & \rightarrow \frac{d\text{out}}{d\text{out}} = 1.0 \\
\end{align*}
\]

Same result, different cost:

\[
\ldots \text{depending of the sizes of in and out}
\]
Full Jacobian with Tangent or Adjoint AD

\[ f : \text{in} \in \mathbb{R}^m \rightarrow \text{out} \in \mathbb{R}^n \]

\[ \frac{d\text{out}}{d\text{in}} = \begin{pmatrix} \cdots & \cdots & \cdots & \cdots \end{pmatrix} \]

- \( \frac{d\text{out}}{d\text{in}} \) costs \( m \times 4? \times P \) using the tangent mode
  - Good if \( m \leq n \)

- \( \frac{d\text{out}}{d\text{in}} \) costs \( n \times 4? \times P \) using the adjoint mode
  - Good if \( m \gg n \) (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 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

\[ \dot{\text{out}} = f'(\text{in}) \cdot \dot{\text{in}} = f'_p(V_{p-1}) \cdot f'_{p-1}(V_{p-2}) \cdots f'_1(V_0) \cdot \dot{\text{in}} \]

Implementation:

Tangent-diff instructions \textit{interleaved} with the original instructions.

almost no problem...
Implementing Adjoint AD

\[ \overline{\text{in}} = \overline{\text{out}}. f'(\text{in}) = \overline{\text{out}}. f'_p(V_{p-1}). f'_{p-1}(V_{p-2}) \cdots f'_1(V_0) \]

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 $l_k$: $c := a*b$ i.e. function:

$$f_k : \mathbb{R}^3 \rightarrow \mathbb{R}^3$$

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \mapsto \begin{pmatrix} a \\ b \\ a*b \end{pmatrix}$$

Its adjoint code must compute:

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

And therefore its adjoint “code” is:

$$\bar{a} := \bar{a} + b*\bar{c}$$
$$\bar{b} := \bar{b} + a*\bar{c}$$
$$\bar{c} := 0.0$$

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].

\[
\begin{align*}
a &:= i_1 \\
b &:= i_2 \\
c &:= a \cdot b \\
d &:= a \cdot c \\
r &:= c + d \\
\end{align*}
\]

\[\text{Adjoint AD} \rightarrow \]

\[
\begin{align*}
\bar{d} &:= \bar{r} \\
\bar{c} &:= \bar{r} + a \cdot \bar{d} \\
\bar{b} &:= a \cdot \bar{c} \\
\bar{a} &:= c \cdot \bar{d} + b \cdot \bar{c} \\
\end{align*}
\]

\[\downarrow \text{?Lagrangian?}\]

\[
\mathcal{L} = \bar{r}(c+d-r)+\bar{d}(ac-d)+\bar{c}(ab-c)+\bar{b}(i_2-b)+\bar{a}(i_1-a)
\]

\[\downarrow\]

\[
\begin{align*}
\frac{d\mathcal{L}}{d\bar{d}} &= 0 = \bar{r}-\bar{d} \\
\frac{d\mathcal{L}}{d\bar{c}} &= 0 = \bar{r}+a\bar{d}-\bar{c} \\
\frac{d\mathcal{L}}{d\bar{b}} &= 0 = a\bar{c}-\bar{b} \\
\frac{d\mathcal{L}}{d\bar{a}} &= 0 = c\bar{d}+b\bar{c}-\bar{a} \\
\end{align*}
\]

\[\rightarrow\]

\[
\begin{align*}
\bar{d} &:= \bar{r} \\
\bar{c} &:= \bar{r} + a \cdot \bar{d} \\
\bar{b} &:= a \cdot \bar{c} \\
\bar{a} &:= c \cdot \bar{d} + b \cdot \bar{c} \\
\end{align*}
\]
Outline

1. AD principle
2. AD tools
3. Challenges of Adjoint AD
4. Data-Flow Analysis
5. Checkpointing
6. Profitable Situations
7. Validation of Adjoint AD
8. The fun of Adjoint AD
9. Commercial break
10. Applications and performance
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

Overloading-based

single-language

CppAD
FADBAD
MAD
AD for Matlab

multi-language

Adol-C
dco
complex-step

Source transformation

single-language

ADIFOR
ADiMat

data-flow reversal

by recomputation

multi-language

data-flow reversal

by storage

Compiler-embedded

OpenAD
Tapenade

NAG compiler
In the sequel we are mostly concerned with Source-Transformation AD

Wait for Uwe’s talk for details on Operator-Overloading AD
Outline

1. AD principle
2. AD tools
3. Challenges of Adjoint AD
4. Data-Flow Analysis
5. Checkpointing
6. Profitable Situations
7. Validation of Adjoint AD
8. The fun of Adjoint AD
9. Commercial break
10. Applications and performance
Challenges of adjoint AD

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

There are many other challenges around AD:

- non-smoothness \cite{Griewank et al.}
- stochastic or chaotic parts \cite{Wang}
- higher derivatives (cost, size...) \cite{Walther, Wang, Pothen}
- . . .
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**:

```
B1
  t1
B2
  PUSH(0)
B3
  PUSH(1)
B4
  PUSH(0)
B5
  PUSH(1)
```

```
B5
  POP(test)
B4
  PUSH(nf)
B2
  PUSH(i-3)
B3
  POP(ni)
B1
  nf=f
```

The same applies to loops and any other construct:

```
DO i=f,t,3
  B2
  PUSH(nf)
  PUSH(i-3)
```

```
DO i=ni,nf,-3
  B2
  POP(ni)
  POP(nf)
```

Works with a stack. Memory cost is negligible.
Adjoint second difficulty: data flow reversal

\[ \overline{\text{in}} = f'^{t}(\text{in}). \overline{\text{out}} = f_1'^{t}(V_{0}) \ldots f_{p-1}'^{t}(V_{p-2}) \cdot f_{p}'^{t}(V_{p-1}) \cdot \overline{\text{out}} \]

In most codes, \( V_{0}, V_{1}, \ldots V_{p-1} \) 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

\[
x = \text{op}_1 \quad y = \text{op}_2 \quad z = \text{op}_3
\]

\[
x = x + y + z
\]

\[
\overline{x} = \overline{\text{op}_1} \quad \overline{y} = \overline{\text{op}_2} \quad \overline{z} = \overline{\text{op}_3}
\]

\[
\overline{x} = \overline{x} + \overline{y} + \overline{z}
\]

Hascoët (INRIA)

ST-AD

GDR Calcul, 2019
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
  → can gain 40 to 70%... still linear memory cost
- trade recomputation/storage ("Checkpointing")
  → achieves logarithmic growth
- exploit profitable situations, (math or algorithm) e.g.
  - Linear solvers
  - Parallel loops
  - Fixed-Point iterations
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 $v$ is useless
  $\Rightarrow$ slice out computation of $v$

- **useful**:
  
  if current $v$ influences no "dependent output", then $v$ is zero
  $\Rightarrow$ propagate constant $v$ 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
These are just special cases of classic code optim.

Agressive compiler optim \cite{Pearlmutter, Siskind} may be more systematic (⇒ 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.

\[
\begin{align*}
TBR^+(I) &= \begin{cases}
(TBR^-(I) \cup \text{use}(I')) \setminus \text{kill}(I) & \text{if } I \text{ live} \\
TBR^-(I) \cup \text{use}(I') & \text{otherwise}
\end{cases}
\end{align*}
\]
<table>
<thead>
<tr>
<th>naïve</th>
<th>Diff-live</th>
<th>TBR</th>
<th>Recompute</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL PUSHINTEGER4(n) n = ind1(i)</td>
<td>CALL PUSHINTEGER4(n) n = ind1(i)</td>
<td>n = ind1(i)</td>
<td>n = ind1(i)</td>
</tr>
<tr>
<td>CALL PUSHREAL4(b(n)) b(n)=SIN(a(n)) - b(n)</td>
<td>CALL PUSHREAL4(b(n)) b(n)=SIN(a(n)) - b(n)</td>
<td>b(n)=SIN(a(n)) - b(n)</td>
<td>b(n)=SIN(a(n)) - b(n)</td>
</tr>
<tr>
<td>CALL PUSHREAL4(a(n)) a(n) = a(n) + x</td>
<td>CALL PUSHREAL4(a(n)) a(n) = a(n) + x</td>
<td>a(n) = a(n) + x</td>
<td>a(n) = a(n) + x</td>
</tr>
<tr>
<td>CALL PUSHREAL4(c) c = a(n)*b(n)</td>
<td>CALL PUSHREAL4(a(n)) a(n) = a(n)*a(n+1)</td>
<td>cb = zb(n)</td>
<td>cb = zb(n)</td>
</tr>
<tr>
<td>CALL POPREAL4(z(n)) z(n) = z(n) + c</td>
<td>CALL POPINTEGER4(z(n))</td>
<td>CALL POPINTEGER4(n)</td>
<td>n = ind2(i+2)</td>
</tr>
<tr>
<td>cb = zb(n)</td>
<td>CALL POPREAL4(z(n))</td>
<td>cb = zb(n)</td>
<td>cb = zb(n)</td>
</tr>
<tr>
<td>CALL POPINTEGER4(n)</td>
<td>CALL POPINTEGER4(n)</td>
<td>CALL POPINTEGER4(n)</td>
<td>n = ind1(i)</td>
</tr>
<tr>
<td>ab(n+1) = ab(n+1) + a(n)*ab(n)</td>
<td>ab(n+1) = ab(n+1) + a(n)*ab(n)</td>
<td>ab(n+1) = ab(n+1) + a(n)*ab(n)</td>
<td>ab(n+1) = ab(n+1) + a(n)*ab(n)</td>
</tr>
<tr>
<td>ab(n) = b(n)*cb + a(n+1)*ab(n)</td>
<td>ab(n) = b(n)*cb + a(n+1)*ab(n)</td>
<td>ab(n) = b(n)*cb + a(n+1)*ab(n)</td>
<td>ab(n) = b(n)*cb + a(n+1)*ab(n)</td>
</tr>
<tr>
<td>CALL POPREAL4(c) bb(n) = bb(n) + a(n)*cb</td>
<td>CALL POPREAL4(a(n))</td>
<td>CALL POPREAL4(a(n))</td>
<td>CALL POPREAL4(a(n))</td>
</tr>
<tr>
<td>bb(n) = bb(n) + a(n)*cb</td>
<td>CALL POPREAL4(b(n))</td>
<td>CALL POPREAL4(b(n))</td>
<td>CALL POPREAL4(b(n))</td>
</tr>
<tr>
<td>xb = xb + ab(n)</td>
<td>xb = xb + ab(n)</td>
<td>xb = xb + ab(n)</td>
<td>xb = xb + ab(n)</td>
</tr>
<tr>
<td>ab(n) = ab(n) + COS(a(n))*bb(n)</td>
<td>ab(n) = ab(n) + COS(a(n))*bb(n)</td>
<td>ab(n) = ab(n) + COS(a(n))*bb(n)</td>
<td>ab(n) = ab(n) + COS(a(n))*bb(n)</td>
</tr>
<tr>
<td>+COS(a(n))*bb(n)</td>
<td>+COS(a(n))*bb(n)</td>
<td>+COS(a(n))*bb(n)</td>
<td>+COS(a(n))*bb(n)</td>
</tr>
</tbody>
</table>
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

⇒ we need something else...
Outline

1. AD principle
2. AD tools
3. Challenges of Adjoint AD
4. Data-Flow Analysis
5. Checkpointing
6. Profitable Situations
7. Validation of Adjoint AD
8. The fun of Adjoint AD
9. Commercial break
10. Applications and performance
Checkpointing: elementary pattern

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

\[
\text{Snapshot} \subset \text{use}(\overline{C}) \cap (\text{out}(C) \cup \text{out}(\overline{D}))
\]
Nesting checkpoints

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

- Peak memory storage grows like $\log(\text{runtime})$
- Execution duplication grows like $\log(\text{runtime})$
- In real life, storage is fixed to $q$ snapshots, execution duplication grows like $q^{\text{th-root}}(\text{runtime})$

[Griewank, Walther]
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
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
Avoid differentiation inside the source of linear solvers
⇒ write their adjoint by hand, calling the solver itself!

```c
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
}
```

[Giles]
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 reads of a variable.
- No DDG arrow either between successive increments of a variable. (assuming increments are atomic, or assuming memory is not shared)
- The adjoint of a $\text{read}(x)$ is an $\text{increment}(\overline{x})$
- The adjoint of an $\text{increment}(x)$ is a $\text{read}(\overline{x})$

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!
// 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

until $z$ converges:
\[ z = \phi(z, x) \]

as many times:
\[ \bar{x} = \bar{x} + \bar{z} \frac{\partial \phi}{\partial x}; \bar{z} = \bar{z} \frac{\partial \phi}{\partial z} \]

You should **not** do that!
- all values from intermediate iterations are stored
- poor convergence guarantees of the adjoint sweep
Two-Phases Adjoint

until \( z \) converges:
\[
z = \phi(z, x)
\]

end

\[
t = \bar{z}
\]

until \( \bar{z} \) converges:
\[
\bar{z} = \bar{z} \frac{\partial \phi}{\partial z} + t
\]

end

\[
\bar{x} = \bar{x} + \bar{z} \frac{\partial \phi}{\partial x}
\]

- 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

[Christianson]
Outline

1. AD principle
2. AD tools
3. Challenges of Adjoint AD
4. Data-Flow Analysis
5. Checkpointing
6. Profitable Situations
7. Validation of Adjoint AD
8. The fun of Adjoint AD
9. Commercial break
10. Applications and performance
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 $\varepsilon$. 
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

1. AD principle
2. AD tools
3. Challenges of Adjoint AD
4. Data-Flow Analysis
5. Checkpointing
6. Profitable Situations
7. Validation of Adjoint AD
8. The fun of Adjoint AD
9. Commercial break
10. Applications and performance
Reading and writing variables

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

<table>
<thead>
<tr>
<th>primal</th>
<th>adjoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>... = ... x ...</td>
<td>xb = xb + ...</td>
</tr>
<tr>
<td>s = s + 2.1*x</td>
<td>xb = xb + 2.1*sb</td>
</tr>
</tbody>
</table>

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

\[ B = \text{malloc}(\text{size}); \]
\[ \text{table} \leftarrow [B, \text{size}, \_] \]
\[ B = \text{FW\_ADMM\_Allocate}(\text{size}); \]
\[ \ldots \]
\[ p = B+9; \]
\[ \ldots \]
\[ *z = \sin(*p); \]
\[ \ldots \]
\[ \text{push}(p); \ p = \ldots; \]
\[ \text{FW\_ADMM\_Deallocate}(B); \]

\[ *p' = *p' + \cos(*p)*(*z'); \]
\[ \ldots \]
\[ \text{ADMM\_Rebase}(&p); \text{pop}(&p); \]
\[ B = \text{BW\_ADMM\_Deallocate}(); \]

\[ \text{free}(B); \]
\[ \text{table} \rightarrow [\text{oldB}, \text{size}, B] \]

\[ \text{free}(B); \]
\[ \text{table} \leftarrow [\text{oldB}, \text{size}, B] \]

\[ p = B+(p-\text{oldB}); \]
\[ \text{find} [\text{oldB}, \text{size}, B] \text{ in table such that } p \text{ in } [\text{oldB}, \text{size}] \]

\[ \text{table} \rightarrow [\text{oldB}, \text{size}, B] \]
\[ B = \text{malloc}(\text{size}); \]
\[ \text{pop}(&\text{size}); \text{pop}(&\text{oldB}); \]
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
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

1. AD principle
2. AD tools
3. Challenges of Adjoint AD
4. Data-Flow Analysis
5. Checkpointing
6. Profitable Situations
7. Validation of Adjoint AD
8. The fun of Adjoint AD
9. Commercial break
10. Applications and performance
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:
  \[
  \text{\$> tapenade -b -head "mod1.foo(d)/(b x y)" file1.f90 file2.f90 aux.f ...<} \text{options>}
  \]
- Free for academic use
- Decent popularity ... despite limitations and bugs
In the sequel, applications images, performance measurements... are made with Tapenade
Outline

1. AD principle
2. AD tools
3. Challenges of Adjoint AD
4. Data-Flow Analysis
5. Checkpointing
6. Profitable Situations
7. Validation of Adjoint AD
8. The fun of Adjoint AD
9. Commercial break
10. Applications and performance
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

30° North

15° North

Kelvin wave
Rossby wave
Data Assimilation (OPA 9.0/NEMO)

2° grid cells, one year simulation
Inverse problem (ALIF/ISSM)

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

- First guess velocity
- Final velocity
- Observed velocity
- First guess drag
- Inferred drag (ALIF)
- Inferred drag (ISSM)

Friction coefficient (s/m): 1/2

Ice velocity (m/a):
- 500
- 1000
- 1500
- 2000
- 2500
- 3000

Hascoët (INRIA)
<table>
<thead>
<tr>
<th>Application</th>
<th>Size</th>
<th>n → m</th>
<th>(A_t)</th>
<th>(R_t)</th>
<th>(A_a)</th>
<th>(R_a)</th>
<th>Peak Traffic (Mb)</th>
<th>Traffic (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uns2d (2,000*F77)</td>
<td>14000 → 3</td>
<td>3.4</td>
<td>2.4</td>
<td>15.1</td>
<td>5.9</td>
<td>241</td>
<td>1243</td>
<td></td>
</tr>
<tr>
<td>nsc2ke (3,500*F77)</td>
<td>1602 → 5607</td>
<td>1.9</td>
<td>2.4</td>
<td>4.5</td>
<td>16.2</td>
<td>168</td>
<td>2806</td>
<td></td>
</tr>
<tr>
<td>lidar (330*F90)</td>
<td>37 → 37</td>
<td>6.7</td>
<td>1.1</td>
<td>14.4</td>
<td>2.0</td>
<td>11</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>nemo (55,000*F90)</td>
<td>9100 → 1</td>
<td>3.0</td>
<td>2.0</td>
<td>8.1</td>
<td>6.5</td>
<td>1591</td>
<td>85203</td>
<td></td>
</tr>
<tr>
<td>gyre (21,000*F90)</td>
<td>21824 → 1</td>
<td>4.5</td>
<td>1.9</td>
<td>13.3</td>
<td>7.9</td>
<td>481</td>
<td>48602</td>
<td></td>
</tr>
<tr>
<td>winnie (3,700*F90)</td>
<td>3 → 1</td>
<td>1.4</td>
<td>1.7</td>
<td>13.7</td>
<td>5.9</td>
<td>421</td>
<td>614</td>
<td></td>
</tr>
<tr>
<td>stics (17,000*F77)</td>
<td>739 → 1467</td>
<td>8.6</td>
<td>2.4</td>
<td>15.3</td>
<td>3.9</td>
<td>155</td>
<td>186</td>
<td></td>
</tr>
<tr>
<td>smac-sail (3,500*F77)</td>
<td>1321 → 7801</td>
<td>5.9</td>
<td>1.0</td>
<td>10.5</td>
<td>3.1</td>
<td>2</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>traces (19,800*F90)</td>
<td>8 → 1</td>
<td>4.0</td>
<td>1.3</td>
<td>12.9</td>
<td>3.8</td>
<td>159</td>
<td>4390</td>
<td></td>
</tr>
<tr>
<td>mit-gcm (258,225*F77)</td>
<td>4704 → 1</td>
<td>8.5</td>
<td>2.0</td>
<td>14.5</td>
<td>6.6</td>
<td>260</td>
<td>5709</td>
<td></td>
</tr>
<tr>
<td>alif (6,755*C)</td>
<td>1413 → 1</td>
<td>6.0</td>
<td>1.6</td>
<td>14.0</td>
<td>4.3</td>
<td>729</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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!
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.
3 to 4 phases,
mostly sequential,
needs interaction with AD tool developers...
## Overloading AD: pros and cons

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>light-weight, versatile</td>
<td>(mildly)hand-modified source</td>
</tr>
<tr>
<td>adapts to exotic control</td>
<td>overloading required,</td>
</tr>
<tr>
<td>and constructs</td>
<td>restricted data-flow analysis</td>
</tr>
<tr>
<td></td>
<td>no global analysis</td>
</tr>
<tr>
<td>higher-order, Taylor,</td>
<td>not-so-efficient adjoints</td>
</tr>
<tr>
<td>intervals</td>
<td>(trajectory storage on tape)</td>
</tr>
</tbody>
</table>
Splitting and merging differentiated instructions

- Split common subexpressions in derivatives
- Merge unnecessary intermediate derivatives

<table>
<thead>
<tr>
<th>naïve adjoint</th>
<th>split and merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>resb = v(j)*gb(i, j)</td>
<td>resb = v(j)*gb(i, j)</td>
</tr>
<tr>
<td>vb(j) = vb(j) + res*gb(i, j)</td>
<td>temp = (z(j)-2.0)/v(j)</td>
</tr>
<tr>
<td>gb(i, j) = 0.0</td>
<td>tempb0 = temp*g(i, j)*resb</td>
</tr>
<tr>
<td>taub = taub</td>
<td>tempb = (tau-w(i, j))</td>
</tr>
<tr>
<td>+(z(j)-2.0)*g(i, j)*resb/v(j)</td>
<td>*g(i, j)*resb/v(j)</td>
</tr>
<tr>
<td>wb(i, j) = wb(i, j)</td>
<td>vb(j) = vb(j)</td>
</tr>
<tr>
<td>-g(i, j)*(z(j)-2.0)*resb/v(j)</td>
<td>+res<em>gb(i, j) -temp</em>tempb</td>
</tr>
<tr>
<td>gb(i, j) = gb(i, j)</td>
<td>gb(i, j) = temp</td>
</tr>
<tr>
<td>+(z(j)-2.0)*(tau-w(i, j))*resb/v(j)</td>
<td>*(tau-w(i, j))*resb</td>
</tr>
<tr>
<td>zb(j) = zb(j)</td>
<td>taub = taub + tempb0</td>
</tr>
<tr>
<td>+(tau-w(i, j))*g(i, j)*resb/v(j)</td>
<td>wb(i, j) = wb(i, j) - tempb0</td>
</tr>
<tr>
<td>vb(j) = vb(j)</td>
<td>zb(j) = zb(j) + tempb</td>
</tr>
</tbody>
</table>
By the way: Combining Checkpointing and TBR

The Snapshot may take care of TBR coming from $U$
The TBR sent to $D$ can take care of the Snapshot

A range of “optimal” combinations exist.
E.g., given $tbr_U$ coming from $U$, “lazy” snapshot:

- **Snapshot** = $\text{out}(C) \cap (\text{use}(\overline{C}) \cup tbr_U)$
- $tbr$ to $D = (\text{use}(\overline{C}) \cup tbr_U) \setminus \text{out}(C)$
- $tbr$ to $C = tbr_U$