Automatic Differentiation for Optimum Design, Applied to Sonic Boom Reduction

Laurent Hascoët, Mariano Vázquez, Alain Dervieux
Laurent.Hascoet@sophia.inria.fr

TROPICS Project, INRIA Sophia-Antipolis

AD Workshop, Cranfield, June 5-6, 2003

PLAN:

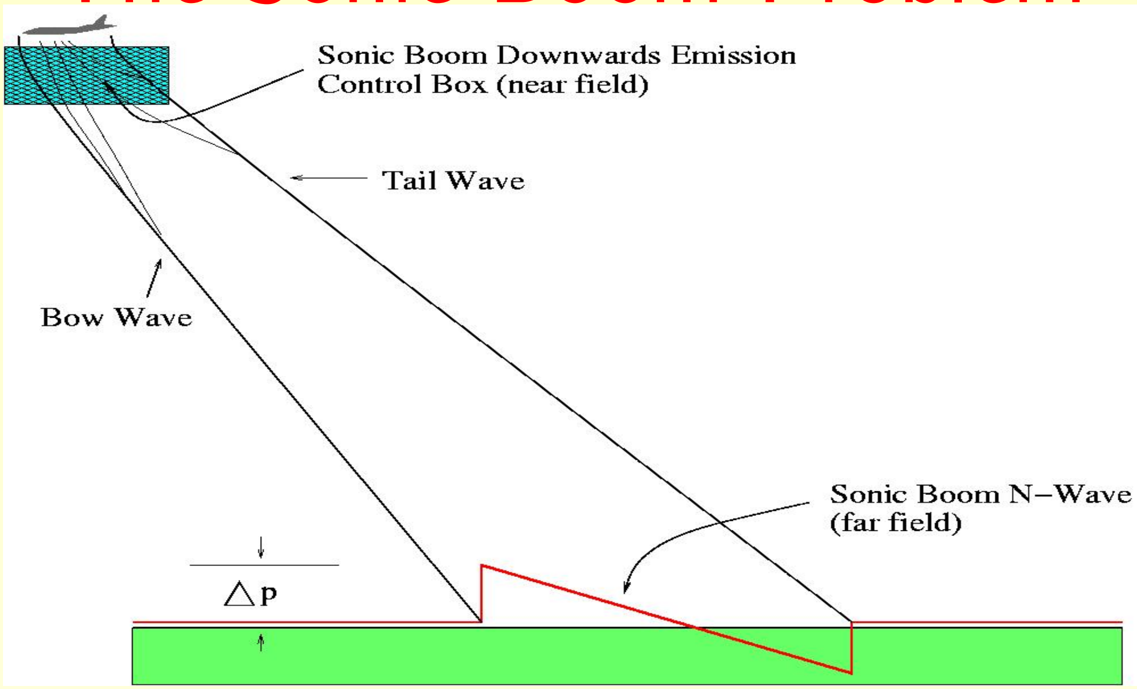
- Part 1: A gradient-based shape optimization to reduce the Sonic Boom
- Part 2: Utilization and improvements of reverse A.D to compute the Adjoint
- Conclusion

PART 1:

GRADIENT-BASED SONIC BOOM REDUCTION

- The Sonic Boom optimization problem
- A mixed Adjoint/AD strategy
- Resolution of the Adjoint and Gradient
- Numerical results

The Sonic Boom Problem



$$\begin{array}{ccccccccc} \text{Control} & & & & \text{Euler} & & \text{Pressure} & & \text{Cost}\\ \text{points} & \Rightarrow & \text{Geometry} & \Rightarrow & \text{flow} & \Rightarrow & \text{shock} & \Rightarrow & \text{function}\\ \gamma & & \Omega(\gamma) & & W(\gamma) & & \nabla p(W) & & j(\gamma)\end{array}$$

Gradient-based Optimization

$j(\gamma)$ models the strength of the downwards Sonic Boom

$\Rightarrow$ Compute the gradient $j'(\gamma)$ and use it in an optimization loop!

$\Rightarrow$ Use reverse-mode AD to compute this gradient

Differentiate the iterative solver?

$W(\gamma)$ is defined implicitly through the Euler equations

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

$\Rightarrow$ The program uses an iterative solver

$\Rightarrow$ Brute force reverse AD differentiates the whole iterative process

- Does it make sense?

- Is it efficient ?

A mixed Adjoint/AD strategy

Back to the mathematical adjoint system:

$$\left\{\begin{array}{rcll} \Psi(\gamma,W) & = & 0 & \textit{(state system)} \\ \frac{\partial J}{\partial W}(\gamma,W) - (\frac{\partial \Psi}{\partial W}(\gamma,W))^t \,.\, \Pi & = & 0 & \textit{(adjoint system)} \\ j'(\gamma) = \frac{\partial J}{\partial \gamma}(\gamma,W) - (\frac{\partial \Psi}{\partial \gamma}(\gamma,W))^t \,.\, \Pi & = & 0 & \textit{(optimality condition} \end{array}\right.$$

lower level ⇒ reverse AD *cf Part 2*

upper level ⇒ hand-coded specific solver for Π

Upper level

Solve $\dfrac{\partial \Psi}{\partial W}(\gamma, W))^t \;.\; \Pi = \dfrac{\partial J}{\partial W}(\gamma, W)$

⇒ Use a matrix-free solver

Preconditioning: “defect correction”

⇒ use the inverse of 1^{st} order $\dfrac{\partial \Psi}{\partial W}$ to precondition 2^{nd} order $\dfrac{\partial \Psi}{\partial W}$

Overall Optimization Loop

Loop:

- compute Π with a matrix-free solver
- use Π to compute $j'(\gamma)$
- 1-D search in the direction $j'(\gamma)$
- update γ (using transpiration conditions)

Overall Optimization Loop

Loop:

- compute Π with a matrix-free solver
- use Π to compute $j'(\gamma)$
- 1-D search in the direction $j'(\gamma)$
- update γ (using transpiration conditions)

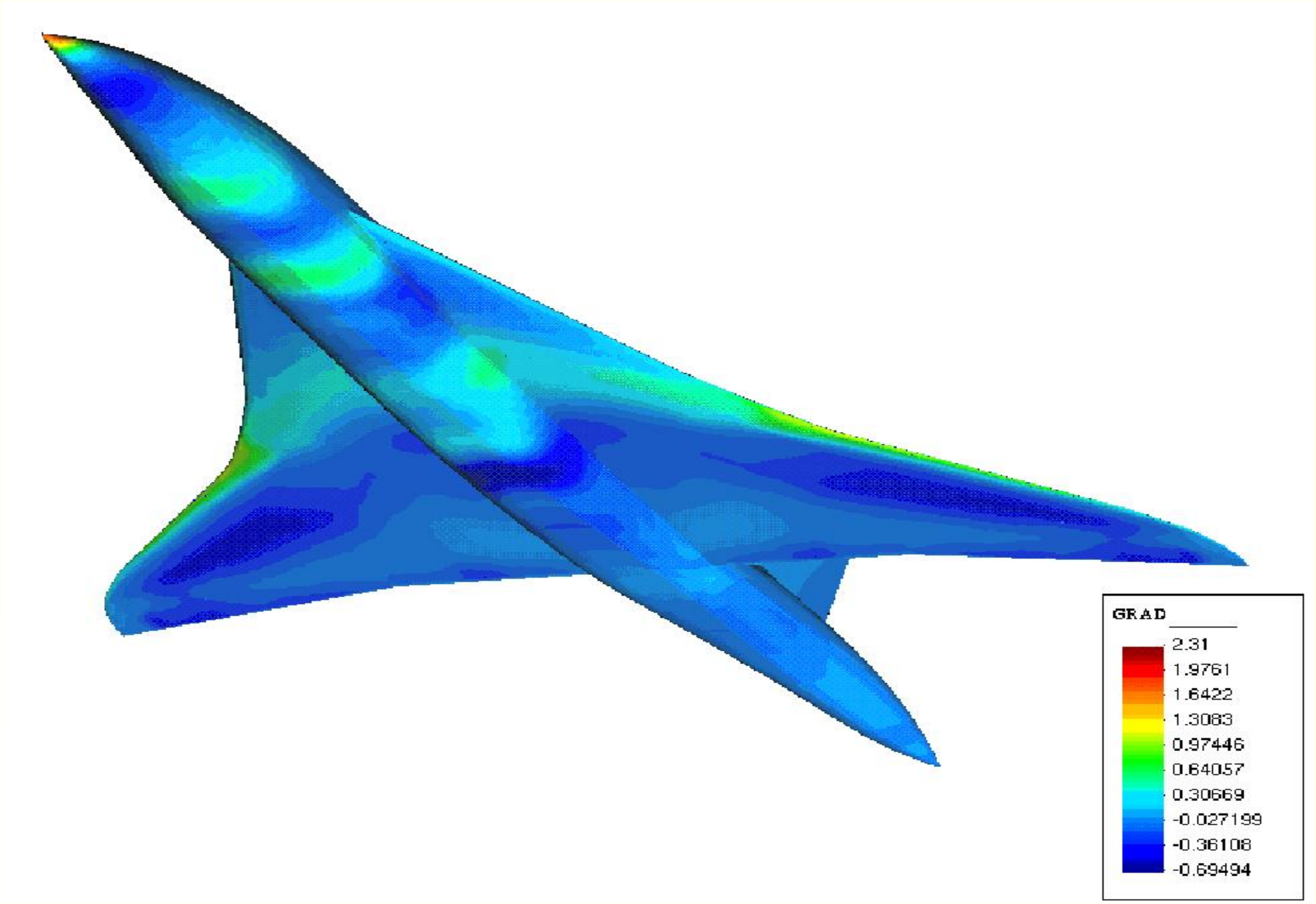
Performances:

Assembling $\frac{\partial \Psi}{\partial W}(\gamma, W))^t$. Π takes about 7 times as long as assembing the state residual $\Psi(\gamma, W))$

$\Rightarrow$ Solving for Π takes about 4 times as long as solving for W.

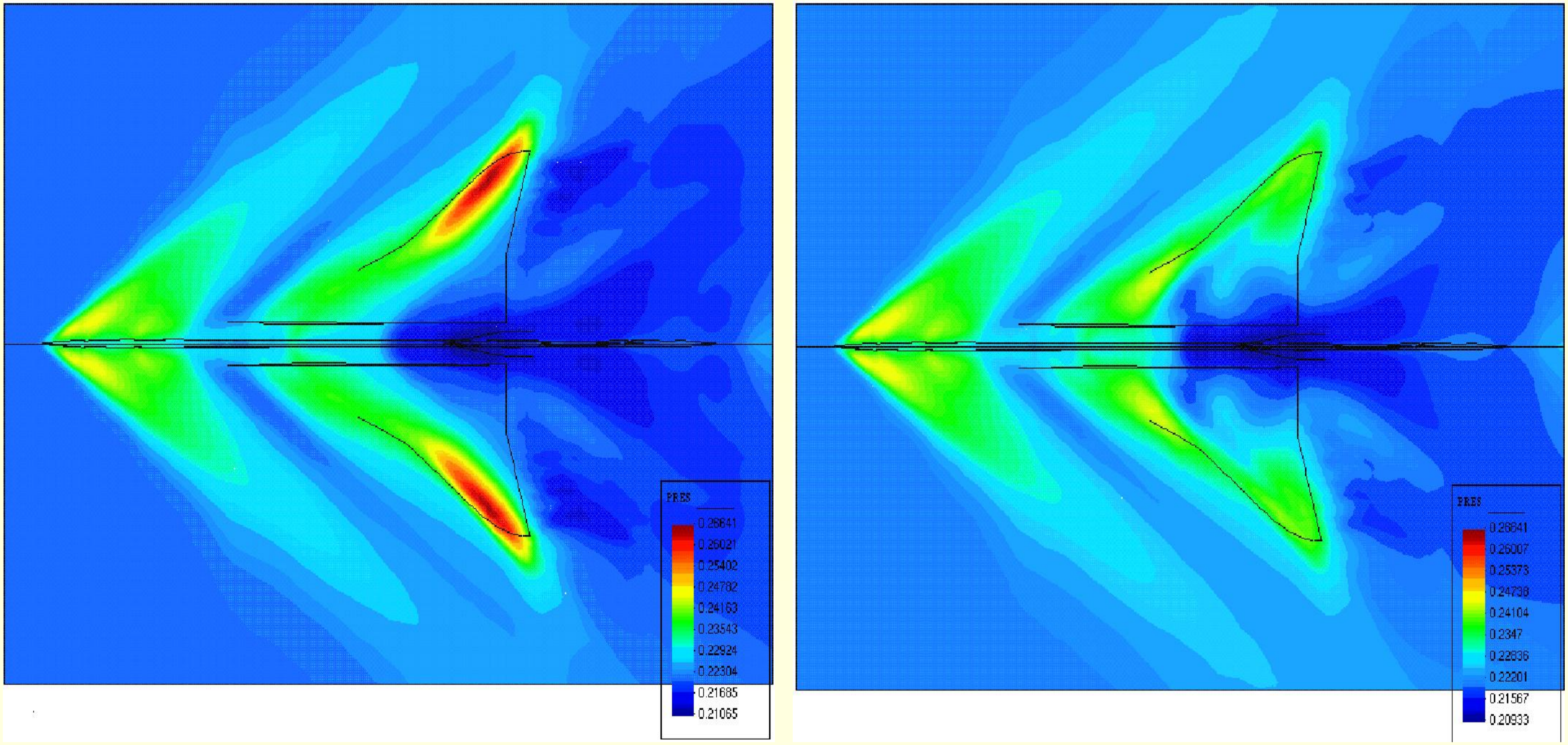
Numerical results 1

Gradient $j'(\gamma)$ on the skin of the plane:



Numerical results 2

Improvement of the sonic boom after 8 optimization cycles:



PART 2:

REVERSE AD TO COMPUTE THE ADJOINT

- Some principles of Reverse AD
- The “Tape” memory problem, the “Checkpointing” tactic
- Impact of Data Dependences Analysis
- Impact of In-Out Data Flow Analysis

Principles of reverse AD

AD rewrites source programs to make them compute derivatives.

consider: $P: \ \{I_1; I_2; \ldots I_p;\}$ implementing $f: \mathbb{R}^m \rightarrow \mathbb{R}^n$

Principles of reverse AD

AD rewrites source programs to make them compute derivatives.

consider: $P : \{I_1; I_2; \dots I_p;\}$ implementing $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$

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

name: $x_0 = x$ and $x_k = f_k(x_{k-1})$

Principles of reverse AD

AD rewrites source programs to make them compute derivatives.

consider: $P : \{I_1; I_2; \dots I_p;\}$ implementing $f : \mathbb{R}^m \to \mathbb{R}^n$

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

name: $x_0 = x$ and $x_k = f_k(x_{k-1})$

chain rule: $f'(x) = f'_p(x_{p-1}).f'_{p-1}(x_{p-2}).\ldots.f'_1(x_0)$

Principles of reverse AD

AD rewrites source programs to make them compute derivatives.

consider: $P : \{I_1; I_2; \dots I_p;\}$ implementing $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$

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

name: $x_0 = x$ and $x_k = f_k(x_{k-1})$

chain rule: $f'(x) = f'_p(x_{p-1}).f'_{p-1}(x_{p-2}).\ldots.f'_1(x_0)$

$f'(x)$ generally too large and expensive $\Rightarrow$ take useful views!

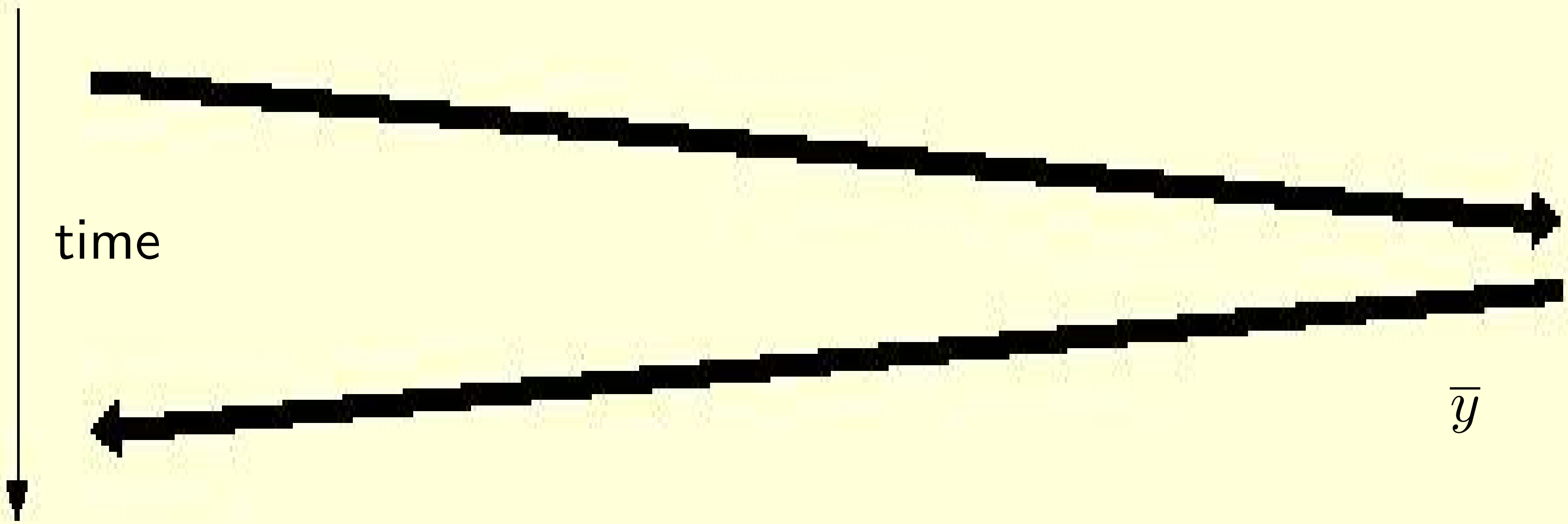
$\dot{y} = f'(x).\dot{x} = f'_p(x_{p-1}).f'_{p-1}(x_{p-2}).\ldots.f'_1(x_0).\dot{x}$ tangent AD

$\overline{x} = f'^*(x).\overline{y} = f'^*_1(x_0).\ldots f'^*_{p-1}(x_{p-2}).f'^*_p(x_{p-1}).\overline{y}$ reverse AD

Evaluate both from right to left !

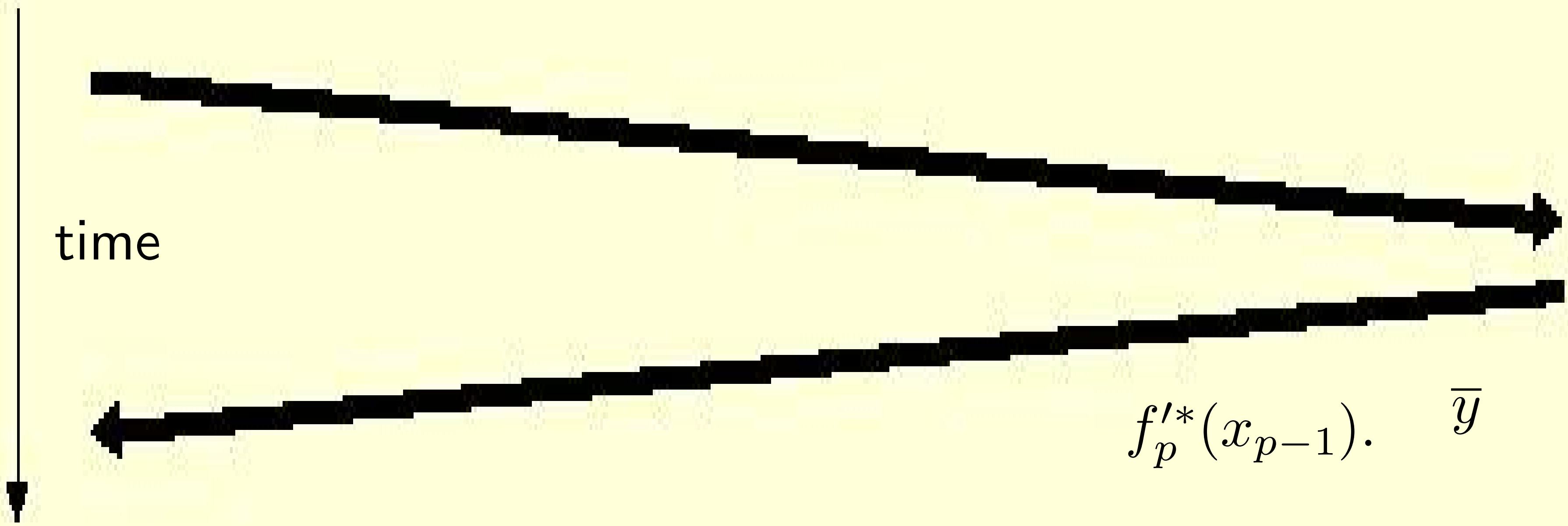
Reverse AD is more tricky than Tangent AD

$$\overline{x} = f'^*(x).\overline{y} = f_1'^*(x_0).\ldots.f_{p-1}'^*(x_{p-2}).f_p'^*(x_{p-1}).\overline{y}$$



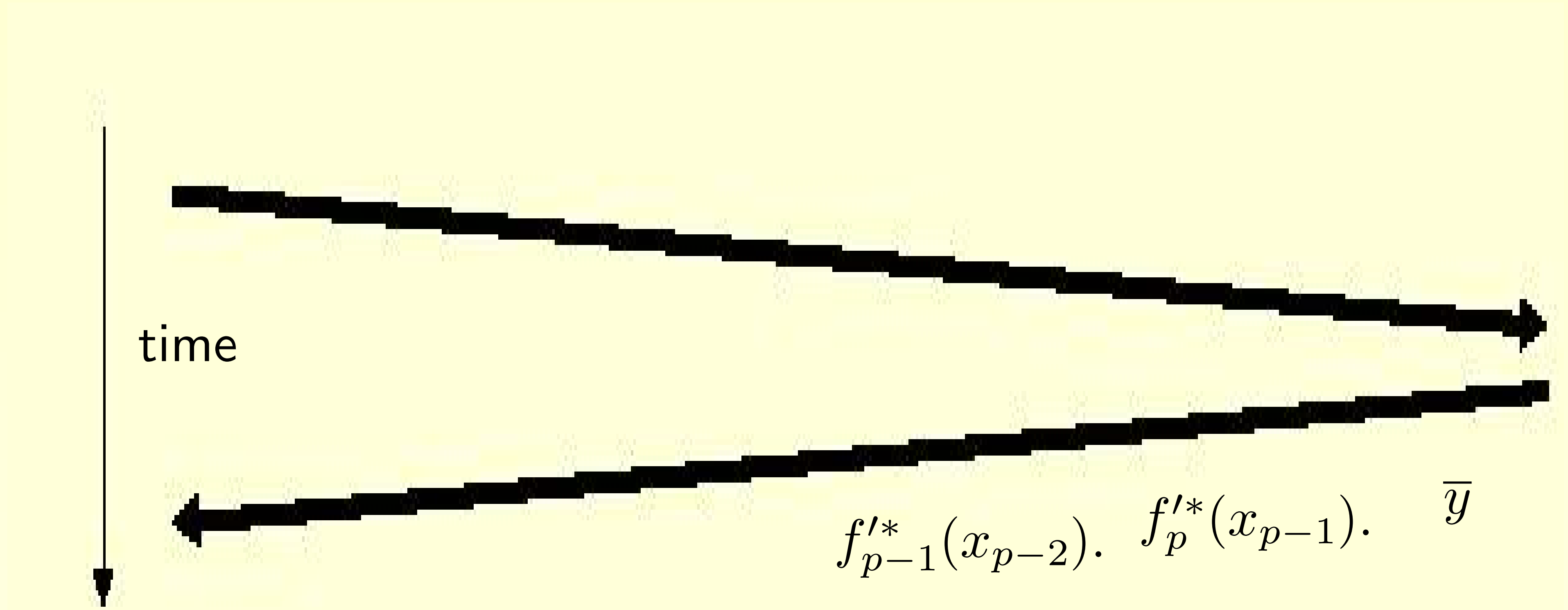
Reverse AD is more tricky than Tangent AD

$$\overline{x} = f'^{*}(x).\overline{y} = f_1'^{*}(x_0).\ldots.f_{p-1}'^{*}(x_{p-2}).f_p'^{*}(x_{p-1}).\overline{y}$$



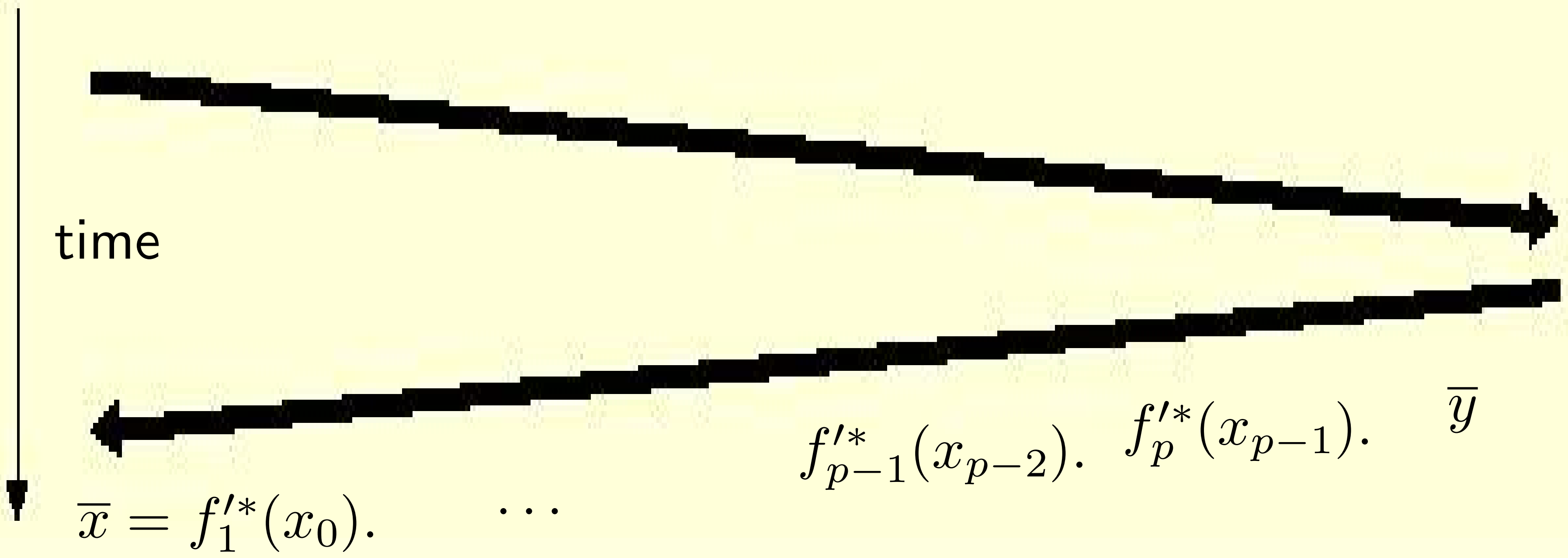
Reverse AD is more tricky than Tangent AD

$$\overline{x} = f'^{*}(x).\overline{y} = f_1'^{*}(x_0).\ldots.f_{p-1}'^{*}(x_{p-2}).f_p'^{*}(x_{p-1}).\overline{y}$$



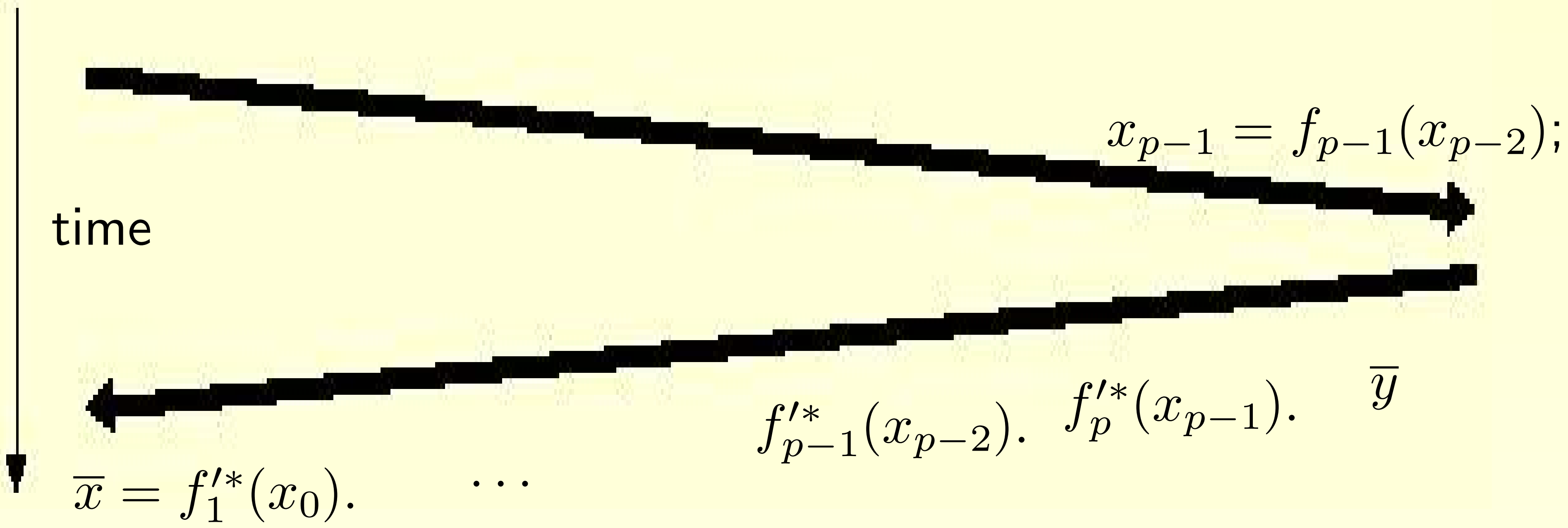
Reverse AD is more tricky than Tangent AD

$$\overline{x} = f'^{*}(x).\overline{y} = f_1'^{*}(x_0).\ldots.f_{p-1}'^{*}(x_{p-2}).f_p'^{*}(x_{p-1}).\overline{y}$$



Reverse AD is more tricky than Tangent AD

$$\overline{x} = f'^*(x).\overline{y} = f_1'^*(x_0).\ldots.f_{p-1}'^*(x_{p-2}).f_p'^*(x_{p-1}).\overline{y}$$



time

$x_{p-1} = f_{p-1}(x_{p-2});$

$\overline{y}$

$f_p'^*(x_{p-1}).$

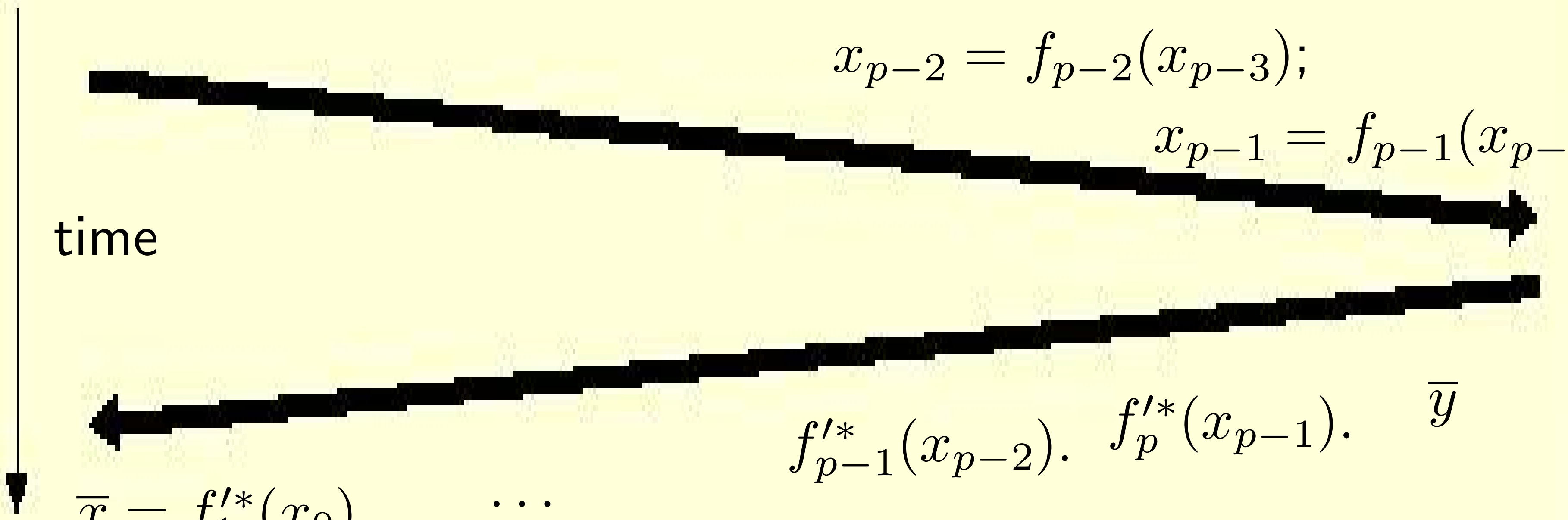
$f_{p-1}'^*(x_{p-2}).$

$\ldots$

$\overline{x} = f_1'^*(x_0).$

Reverse AD is more tricky than Tangent AD

$$\overline{x} = f'^*(x).\overline{y} = f_1'^*(x_0).\ldots.f_{p-1}'^*(x_{p-2}).f_p'^*(x_{p-1}).\overline{y}$$



time

$x_{p-2} = f_{p-2}(x_{p-3});$

$x_{p-1} = f_{p-1}(x_{p-2});$

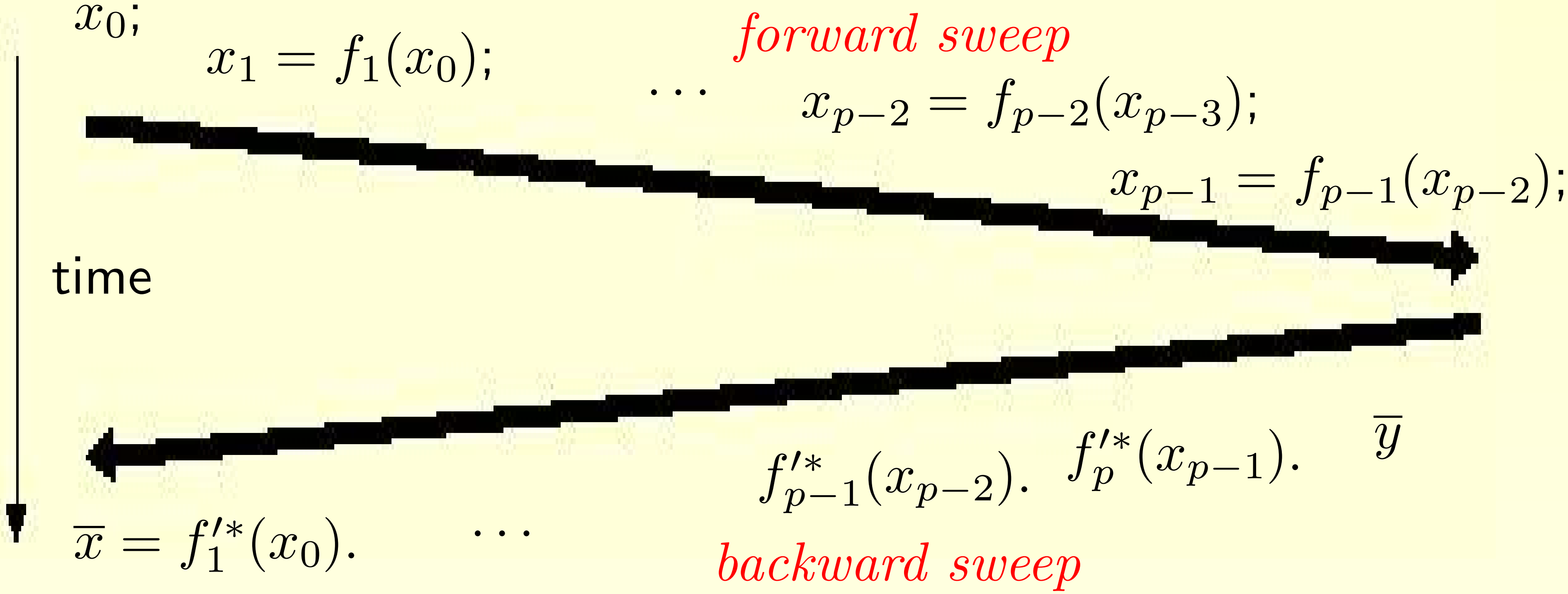
$\overline{y}$

$f_{p-1}'^*(x_{p-2}).\quad f_p'^*(x_{p-1}).$

$\overline{x} = f_1'^*(x_0).\quad \ldots$

Reverse AD is more tricky than Tangent AD

$$\overline{x} = f'^*(x).\overline{y} = f_1'^*(x_0).\ldots.f_{p-1}'^*(x_{p-2}).f_p'^*(x_{p-1}).\overline{y}$$



$x_0;$

$x_1 = f_1(x_0);$

forward sweep

$\ldots$ $x_{p-2} = f_{p-2}(x_{p-3});$

$x_{p-1} = f_{p-1}(x_{p-2});$

time

$\overline{y}$

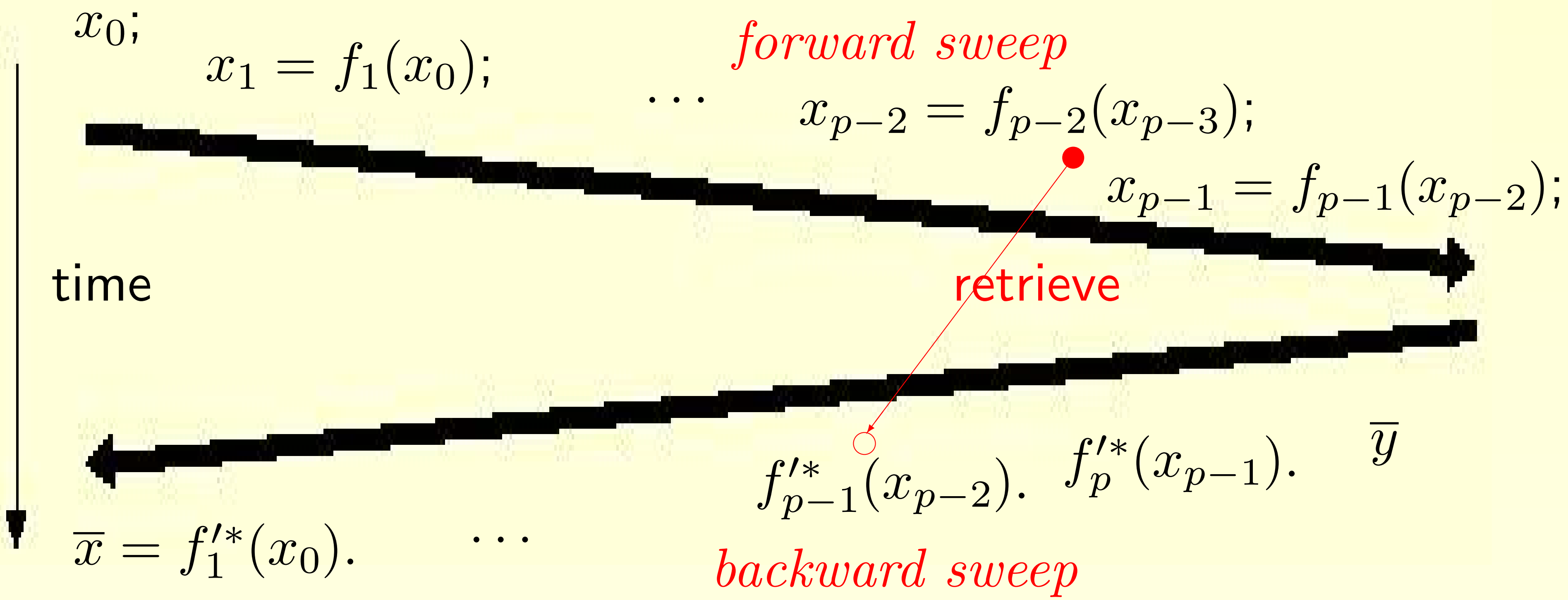
$f_{p-1}'^*(x_{p-2}).$ $f_p'^*(x_{p-1}).$

$\overline{x} = f_1'^*(x_0).$ $\ldots$

backward sweep

Reverse AD is more tricky than Tangent AD

$$\overline{x} = f'^*(x).\overline{y} = f_1'^*(x_0).\ldots.f_{p-1}'^*(x_{p-2}).f_p'^*(x_{p-1}).\overline{y}$$



$x_0;$

$x_1 = f_1(x_0);$

$\ldots$

forward sweep

$x_{p-2} = f_{p-2}(x_{p-3});$

$x_{p-1} = f_{p-1}(x_{p-2});$

time

retrieve

$\overline{y}$

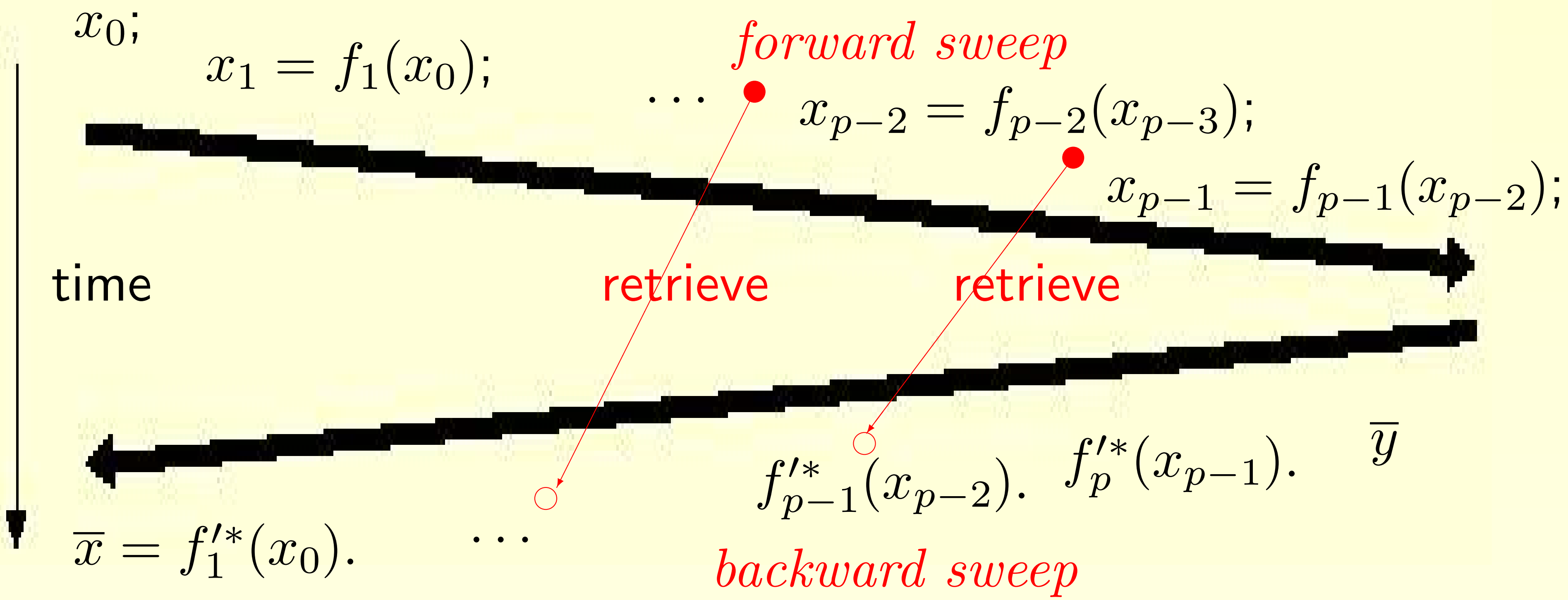
$f_{p-1}'^*(x_{p-2}).$ $f_p'^*(x_{p-1}).$

$\overline{x} = f_1'^*(x_0).$ $\ldots$

backward sweep

Reverse AD is more tricky than Tangent AD

$$\overline{x} = f'^*(x).\overline{y} = f_1'^*(x_0).\ldots.f_{p-1}'^*(x_{p-2}).f_p'^*(x_{p-1}).\overline{y}$$



x_0;

$x_1 = f_1(x_0)$;

forward sweep

$\ldots$

$x_{p-2} = f_{p-2}(x_{p-3})$;

$x_{p-1} = f_{p-1}(x_{p-2})$;

time

retrieve

retrieve

$\overline{y}$

$f_{p-1}'^*(x_{p-2}).$ $f_p'^*(x_{p-1}).$

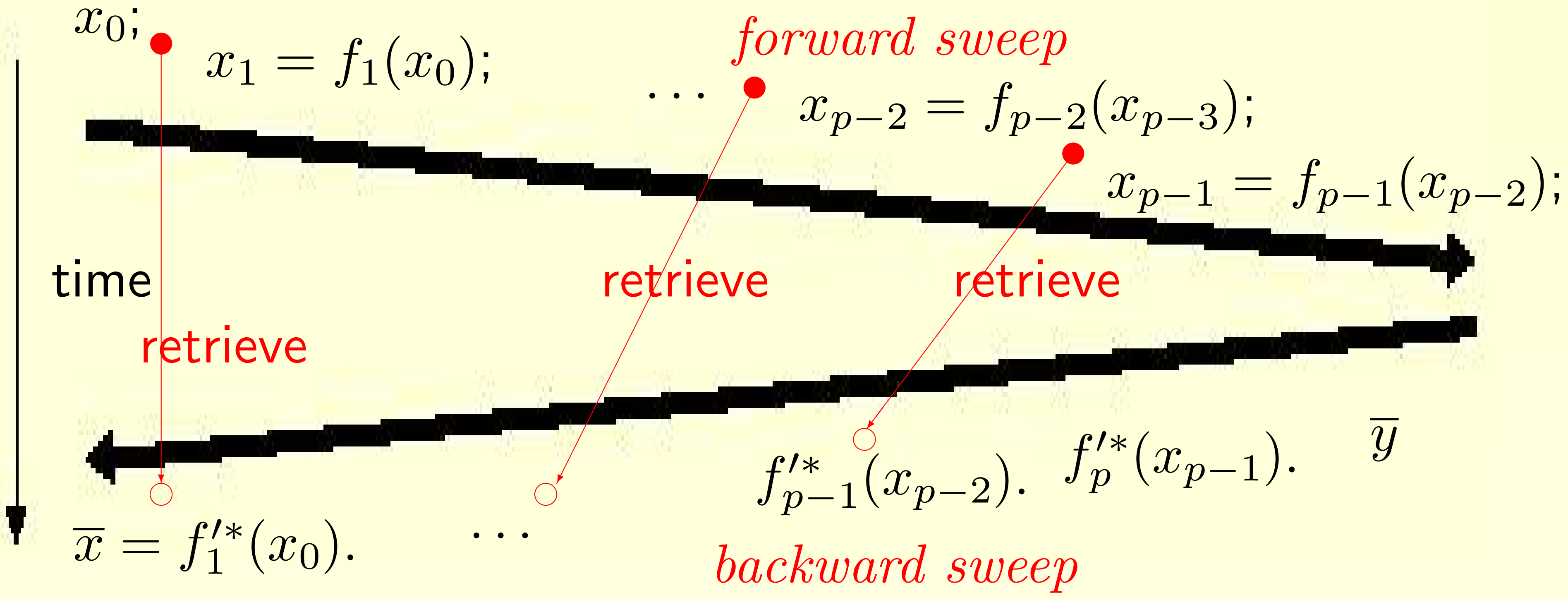
$\overline{x} = f_1'^*(x_0).$

$\ldots$

backward sweep

Reverse AD is more tricky than Tangent AD

$$\overline{x} = f'^*(x).\overline{y} = f_1'^*(x_0).\ldots.f_{p-1}'^*(x_{p-2}).f_p'^*(x_{p-1}).\overline{y}$$



x_0;

$x_1 = f_1(x_0)$;

$\ldots$

forward sweep

$x_{p-2} = f_{p-2}(x_{p-3})$;

$x_{p-1} = f_{p-1}(x_{p-2})$;

time

retrieve

retrieve

retrieve

$\overline{y}$

$f_{p-1}'^*(x_{p-2}).\quad f_p'^*(x_{p-1}).$

$\overline{x} = f_1'^*(x_0).\quad \ldots$

backward sweep

Memory usage (“Tape”) is the bottleneck!

Example:

Program fragment:

$$\begin{array}{rcl} & \textbf{...} & \\ v_2 & = & 2 * v_1 \ + \ 5 \\ v_4 & = & v_2 \ + \ p_1 * v_3 / v_2 \\ & \textbf{...} & \end{array}$$

Example:

Program fragment:

$$\begin{array}{rcl} & \ldots & \\ v_2 & = & 2 * v_1 \ + \ 5 \\ v_4 & = & v_2 \ + \ p_1 * v_3 / v_2 \\ & \ldots & \end{array}$$

Corresponding transposed Partial Jacobians:

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

Reverse mode on the example

...

$$
\begin{aligned}
\bar{v}_2 &= \bar{v}_2 + \bar{v}_4 * (1 - p_1 * v_3/v_2^2) \\
\bar{v}_3 &= \bar{v}_3 + \bar{v}_4 * p_1/v_2 \\
\bar{v}_4 &= 0
\end{aligned}
$$

$$
\begin{aligned}
\bar{v}_1 &= \bar{v}_1 + 2 * \bar{v}_2 \\
\bar{v}_2 &= 0
\end{aligned}
$$

...

Reverse mode on the example

...

$$v_2 = 2 * v_1 + 5$$

$$v_4 = v_2 + p_1 * v_3/v_2$$

...

...

$$\begin{aligned} \bar{v}_2 &= \bar{v}_2 + \bar{v}_4 * (1 - p_1 * v_3/v_2^2) \\ \bar{v}_3 &= \bar{v}_3 + \bar{v}_4 * p_1/v_2 \\ \bar{v}_4 &= 0 \end{aligned}$$

$$\begin{aligned} \bar{v}_1 &= \bar{v}_1 + 2 * \bar{v}_2 \\ \bar{v}_2 &= 0 \end{aligned}$$

...

Reverse mode on the example

...

Push(v_2)

$v_2 \;=\; 2 * v_1 \;+\; 5$

Push(v_4)

$v_4 \;=\; v_2 \;+\; p_1 * v_3/v_2$

...

...

Pop(v_4)

$\bar{v}_2 \;=\; \bar{v}_2 \;+\; \bar{v}_4 * (1 - p_1 * v_3/v_2^2)$

$\bar{v}_3 \;=\; \bar{v}_3 \;+\; \bar{v}_4 * p_1/v_2$

$\bar{v}_4 \;=\; 0$

Pop(v_2)

$\bar{v}_1 \;=\; \bar{v}_1 \;+\; 2 * \bar{v}_2$

$\bar{v}_2 \;=\; 0$

...

Reverse mode on our application

From subroutine `Psi` :

`Psi`: $\gamma, W \mapsto \Psi(\gamma, W)$,

Use reverse AD to build subroutine $\overline{\texttt{Psi}}$:

$\overline{\texttt{Psi}}$: $\gamma, W, \overline{\Psi} \mapsto \frac{\partial \Psi}{\partial W}(\gamma, W))^t \,.\, \overline{\Psi}$

Reverse mode on our application

From subroutine `Psi` :

$$\texttt{Psi}:\ \gamma, W \mapsto \Psi(\gamma, W),$$

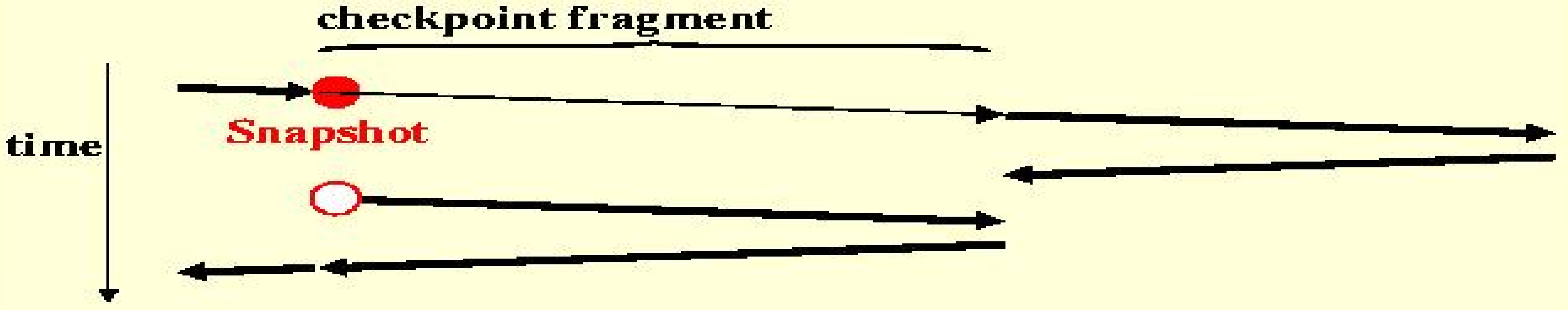
Use reverse AD to build subroutine $\overline{\texttt{Psi}}$:

$$\overline{\texttt{Psi}}:\ \gamma, W, \overline{\Psi} \mapsto \frac{\partial \Psi}{\partial W}(\gamma, W))^t \ . \ \overline{\Psi}$$

But the Tape grows too large on large meshes!

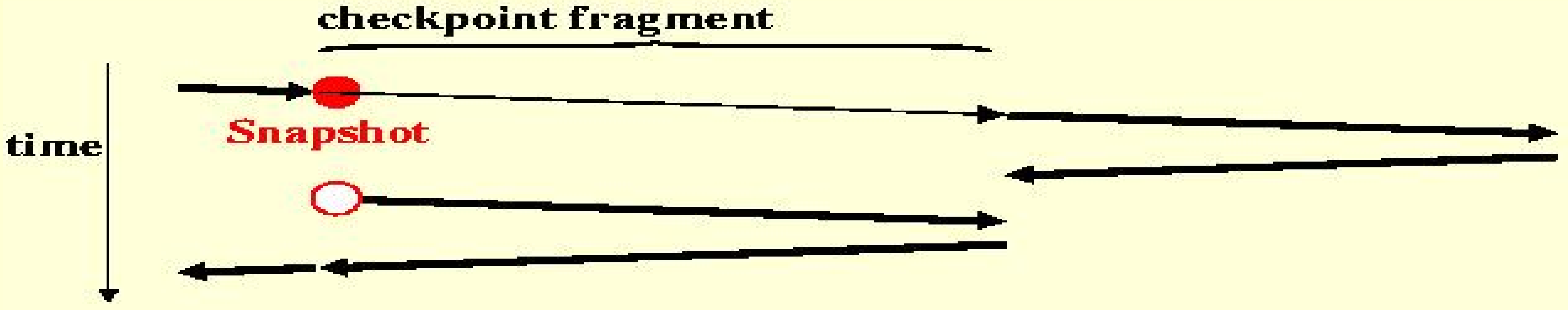
The Checkpointing mechanism

A Storage/Recomputation tradeoff:

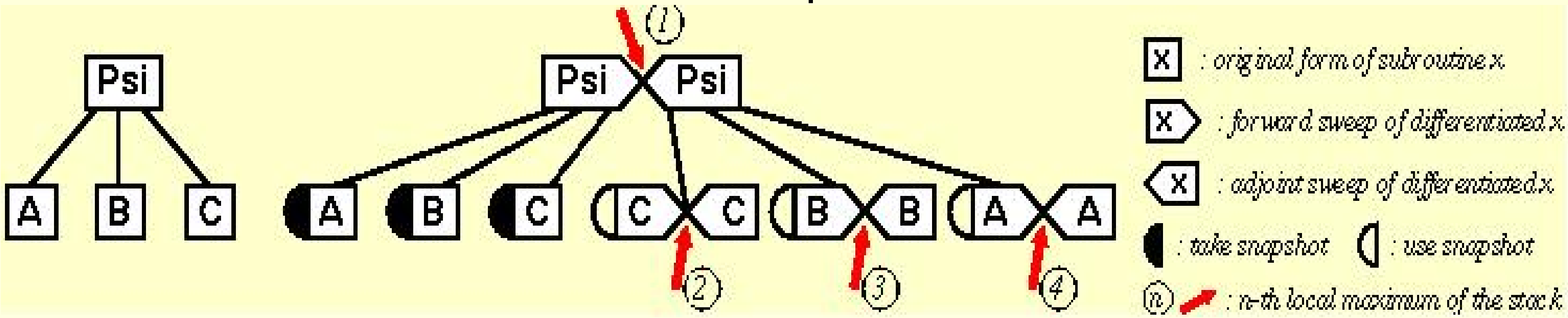


The Checkpointing mechanism

A Storage/Recomputation tradeoff:

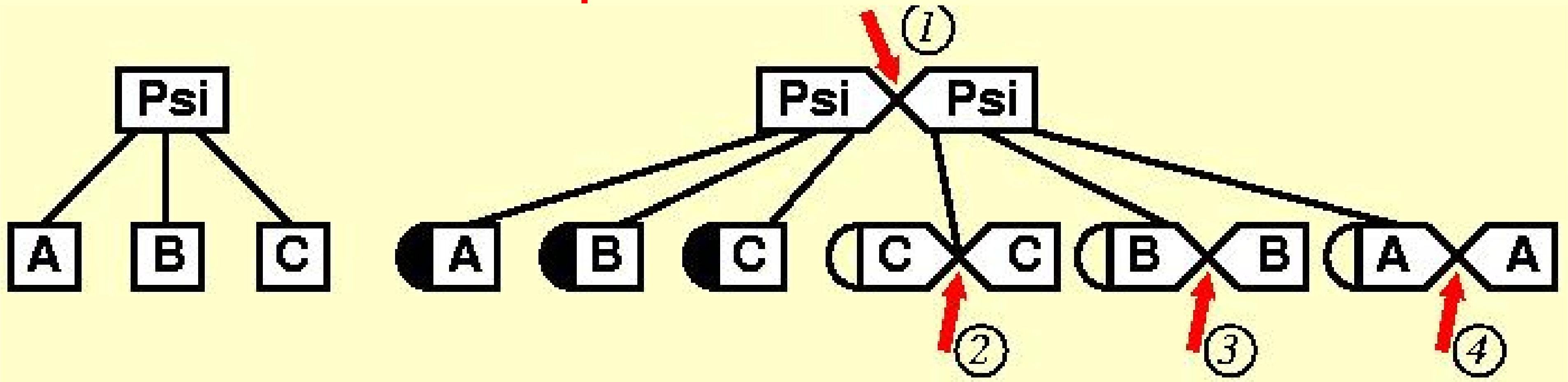


TAPENADE does it on the Call Graph :


Psi
A
B
C
Psi
Psi
A
B
C
C
C
B
B
A
A
(1)
(2)
(3)
(4)
X : original form of subroutine x
X : forward sweep of differentiated x
X : adjoint sweep of differentiated x
: take snapshot
: use snapshot
(n) : n-th local maximum of the stack

Tape size reaches 4 local maxima.

Tape size maxima



Tape local maximum #	**1**	**2**	**3**	**4**
	12.40	12.37	13.60	9.66

773 `R*8`/node: (still) too expensive in memory

⇒ Use Data Flow analysis

Data-Flow “to the rescue” (1)

Data Dependence Graphs of P and $\overline{\mathtt{P}}$ are isomorphic, so...
improve Independent-Iterations loops (“*II*-loops”)

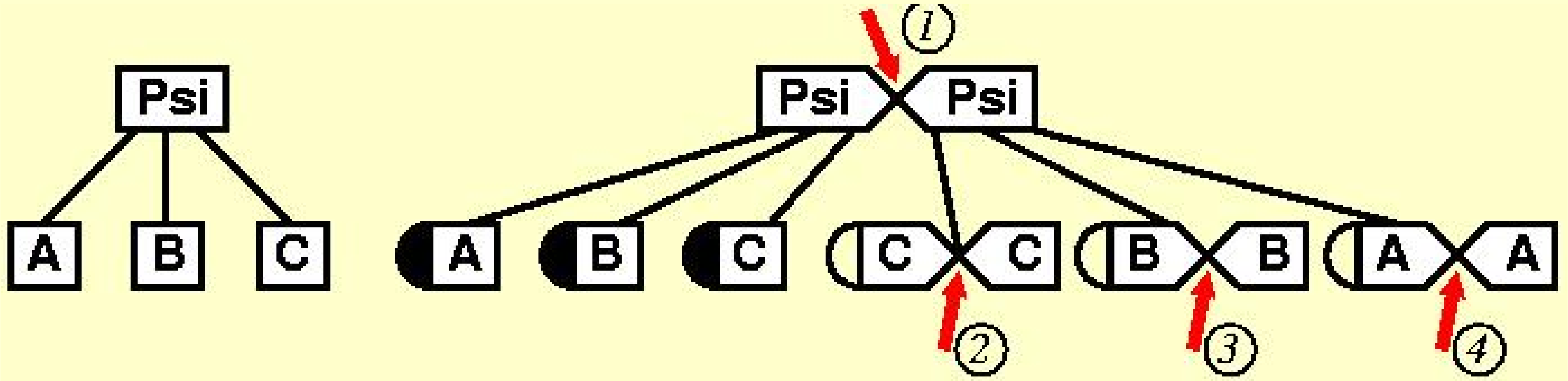
Standard:

```
do // i = 1,N
      $body(i)$
end
do i = N,1
      $\overline{body(i)}$
end
```

$\Longleftrightarrow$

Improved:

```
do i = 1,N
      $body(i)$
      $\overline{body(i)}$
end
```

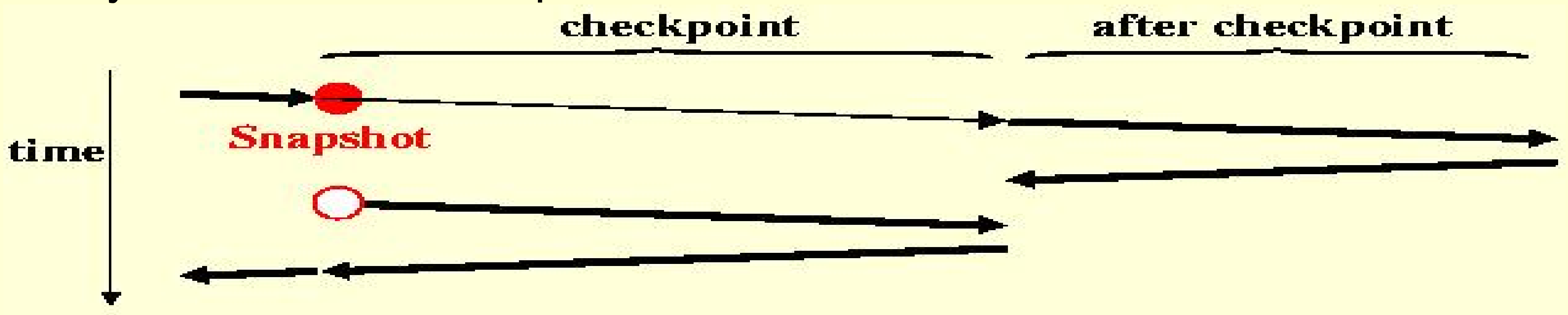


Tape local maximum #	**1**	**2**	**3**	**4**
No modification:	12.40	12.37	13.60	9.66
II-loops improvement:	12.38	7.98	4.10	0.02

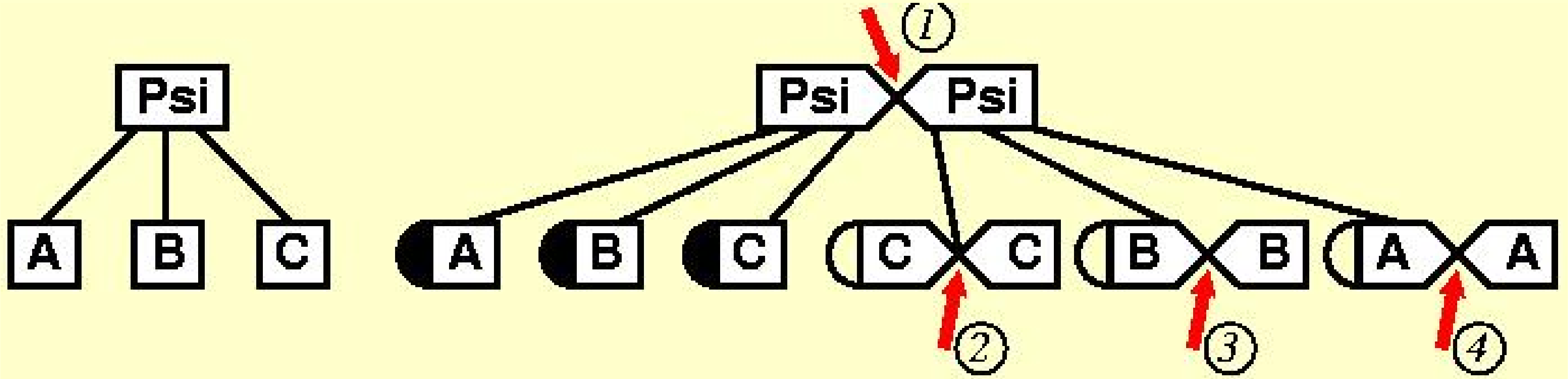
Improvement on (4), but hidden by (1) !

Data-Flow “to the rescue” (2)

Analyze the size of Snapshots:



$\textcolor{red}{\text{Snapshot}} = \textbf{IN}(\textbf{checkpoint}) \bigcap \textbf{OUT}(\textbf{checkpoint and after})$



Tape local maximum #	**1**	**2**	**3**	**4**
No modification:	12.40	12.37	13.60	9.66
Only snapshot reduction:	1.02	0.85	9.70	9.33
Only *II*-loops improvement:	12.38	7.98	4.10	0.02
Both improvements:	1.02	0.61	0.22	0.02

58 `R*8`/node: quite acceptable !

CONCLUSION:

- Part 1: Brute-force reverse AD, including on the iterative solver, is a hazardous strategy ⇒ define a manual strategy *before* AD.

- Part 1: ... but the matrix-free solver proves a delicate step.

- Part 2: Reverse AD can use reasonable memory space, thanks to data flow analyses.

- Advertisment! TAPENADE 2.0: an AD tool on the web `http://tapenade.inria.fr:8080/tapenade`
  or alternatively FTP from our web site
  `http://www.inria.fr/tropics`.