Chapter 6

Automatic Differentiation

At the present stage, the reader known why it can be interesting to consider an adjoint model, what for. In practice, the strategy to write the adjoint model may be different according to the forward computational code already available. One of the three possibilities to derive an adjoint code, consists to derive directly the forward *source code*. This option is called source-to-source transformation; it is possible in C or Fortran (at leat). In the present chapter, the forward source code is supposed to be written in Fortran. In the next section, the advantages and drawbacks of "automatic differentiation" is presented briefly. Next, we present the relationship between the differential calculus (as considered in the previous chapters) and a source code differentiation. The presentation is done in the framework of the software Tapenade, [13]. Also, we present how to validate an adjoint code. Next, the conceptual way to re-use the direct linear solver (eg arising from our favorite linear algebra library) in the adjoint code, since it solves a linear system thus the algorithm should not be differentiated. Finally, are presented some useful tricks related to the standard MPI instructions (how to adjointize the forward MPI instructions ?), and how to save a bit of memory (memory may be the greatest problem while running an adjoint code).

6.1 What adjoint code ?

In practice, there exists three approaches to compute the adjoint state variable, then the gradient of the cost function.

1. We write the continuous adjoint equations, we discretize them using an appropriate numerical scheme (a-priori the same numerical method than the direct model), we discretize the expression of the (continuous) gradient. We obtain the so-called *discretized continuous gradient*

2. We discretize the direct model. We derive these discrete equations in order to obtain the adjoint discrete equations. It is calculations in finite dimension spaces, that is why the present approach is the most famous one in engineering. We obtain the so-called *discrete gradient*.

3. The *computational gradient* is obtained from the differentiation of the forward code directly !

The three approaches are possible. The choice should be done upon the human time development required. The three gradients obtained are not exactly the same since propagation of all different errors is not the same for each approach. Very few numerical analysis results exits on the topic (e.g. in case of a linear PDE discretized using conforming finite element, the first two approaches are the same up to the scheme errors).
For non-linear and very large scale problems, some end-users claim that the tiny differences between the three gradients can justify differences of local convergence behaviors observed...

The algorithmic differentiation approach (last approach) may present two advantages. First, it may ensure a better consistency between the computed cost function, which is the output of the direct code, and its gradient since it is the computed cost function which is differentiated (consistency including all types of errors: schemes, rounding errors, iterative algorithms etc). Second, a large part of this extensive task can be automated using algorithmic differentiation, see e.g. [11], if the direct code has been designed to it !... In case of a direct code written in Fortran, and initially if designed to be differentiated by an algorithmic process, the adjoint code can be almost automatically derived using an automatic differentiation software. One of the most efficient automatic differentiation tool, source-to-source, is Tapenade, see [13, 12].

The computational software DassFlow presented previously has been initially design to be differentiated by Tapenade; that is why one can obtain very fast the adjoint code up-to-date. As a matter of fact, let us point out that as soon as the direct model is modified (extra terms, numerical approximation changed etc), the corresponding adjoint code must be modified in consequence, whatever the approach chosen.

6.2 From mathematical differentiation to source code differentiation

We describe how to define the direct code, then what is the response of the adjoint code automatically generated and finally how to use it. We follow the presentation done in [14], see [15] too.
Let $\mathcal{K}$ be the space of control variables and $\mathcal{Y}$ the space of the forward code response. In the case of DassFlow, we have :

$$\mathbf{k} = (y_0, q_{in}, z_{out}, n, z_b,)^T \quad \text{and} \quad Y = \big(y, j\big)^T$$

Let us point out that we include both the state and the cost function in the response of the forward code.

We can represent the direct code as the operator $\mathcal{M} : \mathcal{K} \longrightarrow \mathcal{Y}$, see figure 6.1.

The tangent model becomes $\frac{\partial \mathcal{M}}{\partial \mathbf{k}}(\mathbf{k}) : \mathcal{K} \longrightarrow \mathcal{Y}$. It takes as input variable a perturbation of the control vector $d\mathbf{k} \in \mathcal{K}$, then it gives the variation $dY \in \mathcal{Y}$ as output variable, see figure 6.2:

$$dY = \frac{\partial \mathcal{M}}{\partial \mathbf{k}}(\mathbf{k}) \cdot d\mathbf{k}$$

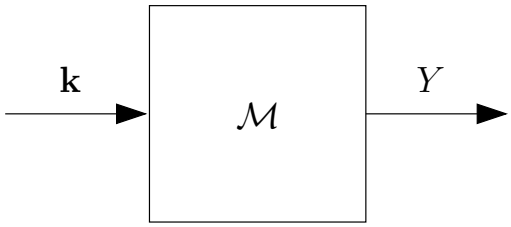


Figure 6.1: Representation of the direct model.

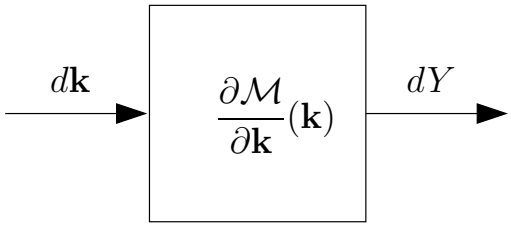


Figure 6.2: Representation of the tangent model.

The adjoint model is defined as the adjoint operator of the tangent model. This can be represented as follows: $\left(\frac{\partial \mathcal{M}}{\partial \mathbf{k}}(\mathbf{k})\right)^* : \mathcal{Y}' \longrightarrow \mathcal{K}'$. It takes $dY^* \in \mathcal{Y}'$ an input variable and provides the adjoint variable $d\mathbf{k}^* \in \mathcal{K}'$ at output, see figure 6.3:

$$d\mathbf{k}^* = \left(\frac{\partial \mathcal{M}}{\partial \mathbf{k}}(\mathbf{k})\right)^* \cdot dY^*$$

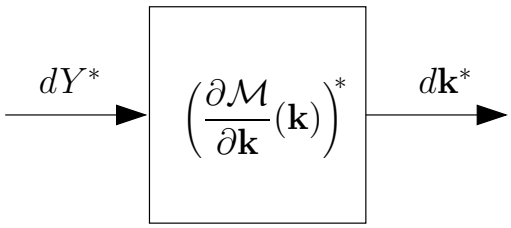


Figure 6.3: Representation of the adjoint model.

Now, let us make the link between the adjoint code and the "computational gradient".

By definition of the adjoint, we have :

$$\left\langle \left(\tfrac{\partial \mathcal{M}}{\partial \mathbf{k}}\right)^* \cdot dY^*,\, d\mathbf{k} \right\rangle_{\mathcal{K}' \times \mathcal{K}} = \left\langle dY^*,\, \left(\tfrac{\partial \mathcal{M}}{\partial \mathbf{k}}\right) \cdot d\mathbf{k} \right\rangle_{\mathcal{Y}' \times \mathcal{Y}} \tag{6.1}$$

or, using the relations presented above:

$$\left\langle d\mathbf{k}^*,\, d\mathbf{k} \right\rangle_{\mathcal{K}' \times \mathcal{K}} = \left\langle dY^*,\, dY \right\rangle_{\mathcal{Y}' \times \mathcal{Y}} . \tag{6.2}$$

If we set $dY^* = (0,1)^T$ and by denoting the perturbation vector $d\mathbf{k} = (\delta y_0, \delta n, \delta z_b, \delta q^{in})^T$,

we obtain:

$$\left\langle \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} dy^* \\ dJ^* \end{pmatrix} \right\rangle_{\mathcal{Y}'\times\mathcal{Y}} = \left\langle \begin{pmatrix} \delta y_0^* \\ \delta n^* \\ \delta z_b^* \\ \delta q^{in*} \\ \delta z_{out}^* \end{pmatrix}, \begin{pmatrix} \delta y_0 \\ \delta n \\ \delta z_b \\ \delta q^{in} \\ \delta z_{out} \end{pmatrix} \right\rangle_{\mathcal{K}'\times\mathcal{K}}$$

Moreover, we have by definition:

$$dj = \frac{\partial J}{\partial y_0}(\mathbf{k}) \cdot \delta y_0 + \frac{\partial J}{\partial n}(\mathbf{k}) \cdot \delta n + \frac{\partial J}{\partial z_b}(\mathbf{k}) \cdot \delta z_b + \frac{\partial J}{\partial q_{in}}(\mathbf{k}) \cdot \delta q_{in} + \frac{\partial J}{\partial z_{out}}(\mathbf{k}) \cdot \delta z_{out}$$

Therefore, the adjoint variable $d\mathbf{k}^*$ (output of the adjoint code with $dY^* = (0,1)^T$) corresponds to the partial derivatives of the cost function j:

$$\frac{\partial J}{\partial y_0}(\mathbf{k}) = y_0^* \qquad \frac{\partial J}{\partial n}(\mathbf{k}) = n^*$$
$$\frac{\partial J}{\partial z_b}(\mathbf{k}) = z_b^* \qquad \frac{\partial J}{\partial q_{in}}(\mathbf{k}) = q_{in}^* \qquad \frac{\partial J}{\partial z_{out}}(\mathbf{k}) = z_{out}^*$$

In summary, in order to compute the "computational gradient" (partial derivatives of the cost function j using differentiation of the forward code), first, one runs the direct code then one runs the adjoint code with $dY^* = (0,1)^T$ as input.

6.3 Automatic differentiation in short

Automatic differentiation softwares Automatic Differentiation (AD) (also called *algorithmic differentiation*) aims at computing the derivative of an program output. one the possible approach is the source-to-source one; it means the AD software (kind of a "super-over-compiler"...) transforms the direct source code into a tangent linear source code and/or an adjoint source code.
Researches in this field are very active; nowadays few reliable AD tools are available, both for Fortran and C languages. Let us cite only OpenAD and Tapenade. There use is still quite technical and tricky. It requires a good knowledge of what is the derivative of the output of the model, next how it is translated in terms of a program instructions.
In Fortran and C languages, the only remaining limitation is the pointer management; nevertheless some easy rules allow to circumvent this limitation.

We present below the basic principles of a AD tool, and we will focus on the Tapenade. Tapenade [13, 12] is an automatic differentiation tool for Fortran and C programs. It is developed by the Tropics team at INRIA Sophia-Antipolis. Tapenade works using source code transformation : it builds the tangent and/or adjoint code automatically from the direct source code. We refer to its rich webpage and the FAQ section.

Concerning AD in general, we refer also to [11, 23].

Some basic rules to respect for the direct code. In order to be able to differentiate a Fortran source code using Tapenade, one must respect some basic rules of code structure. Typically, the code structure must be as follows:

. Preprocessing.
Reading mesh, parameters, data etc AND allocate all arrays.

. Computational kernel.
At the second level of the program tree, one gives the head routine to be differentiated to Tapenade. From this head routine, neither allocate nor pointer may appear.

. Post-processing.
Writing in files, visualization etc.

Towards Data Assimilation

Parameters
- Manning, DEM (time independ.)
- B.C. (time dependent) + I.C.

Model math & num
Computational Code
(DassFlow - Telemac2D)

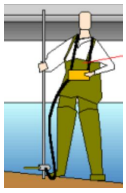
Observations:
measures in situ, remotely-sensed

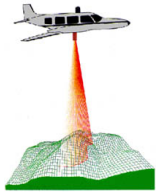
?

?

Simulated state of the system
(h , q)

Variationnal D.A.:
Minimization discrepancy
(cost function)



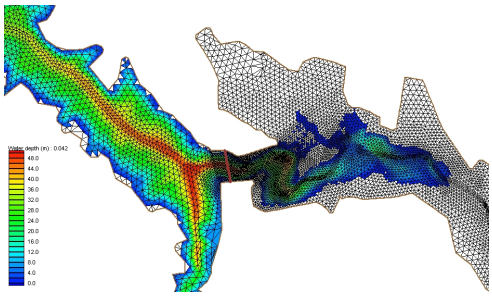


$$\frac{\partial(HU)}{\partial t} + \frac{\partial}{\partial x}\left(\beta_{ss} HUU + (\cos\alpha_x + \cos\alpha_z)^2 \frac{1}{2} gH^2 \right.$$

$$-\Omega HV + \frac{1}{\rho}\left[\tau_{bx} - \tau_{sx} - \frac{\partial(H}{\partial}$$

$$\frac{\partial(HV)}{\partial t} + \frac{\partial}{\partial y}\left(\beta_{ss} HVV + (\cos\alpha_y + \cos\alpha_z)^2 \frac{1}{2} gH^2 \right.$$

$$-\Omega HU + \frac{1}{\rho}\left[\tau_{by} - \tau_{sy} - \frac{\partial(H}{\partial}$$



Cost function:
It measures the discrepancy between the simulated state and the observations

It is a function of the control variable k (Manning, topo., B.C., I.C.):

$$j(k) = \frac{1}{2}\int_0^T \|C\, y(k;t) - y^{obs}(t)\|_*^2 \, dt \; + \; \text{regularization terms}$$

where C: operator of observations

$\|.\|_*$: norm in the observation space.

Optimization problem:

$$\begin{cases} \text{Find } k^\star \text{ such that} \\ j(k^\star) = \min_K j(k) \end{cases}$$

where $j(k) = J(k, y^k)$ and y^k is the solution of the model (state of the system)

The state equation (forward model)

The control variable: $k = (y_0, v(t))^T$

y_0: I.C.; $v(t)$: parameters

State equation (nonlinear): Given k, find $y(t)$ s.t.

$$\begin{cases} \partial_t y(t) + A\big(v(t); y(t)\big) = 0 & \forall t \in]0,T[\\ y(0) = y_0 \end{cases}$$

This gives $y^k(t)$: the state, $j(k) = J(k; y^k)$: the cost function

Since the cost computation is time-consuming:
local minimization algorithm (quasi-Newton e.g. BFGS)
Since many variables of control:
introduction of the adjoint model to compute the gradient

Ref. JL Lions '68

Adjoint State equation Find $\tilde{y}(t)$ s.t.

$$\begin{cases} \partial_t \tilde{y}(t) - \left[\frac{\partial A}{\partial y}(y(t), v(t))\right]^* \tilde{y}(t) = C^* \Lambda_o \big(C\, y(t) - y^{obs}(t)\big) & \forall t \in]0,T[\\ \tilde{y}(T) = 0 \end{cases}$$

Note. a. Reverse time b.Contains the observations

Gradient obtained:

$$\frac{\partial j}{\partial y_0}(\mathbf{k}) = -\tilde{y}(0) + N(y_0 - y_b)$$

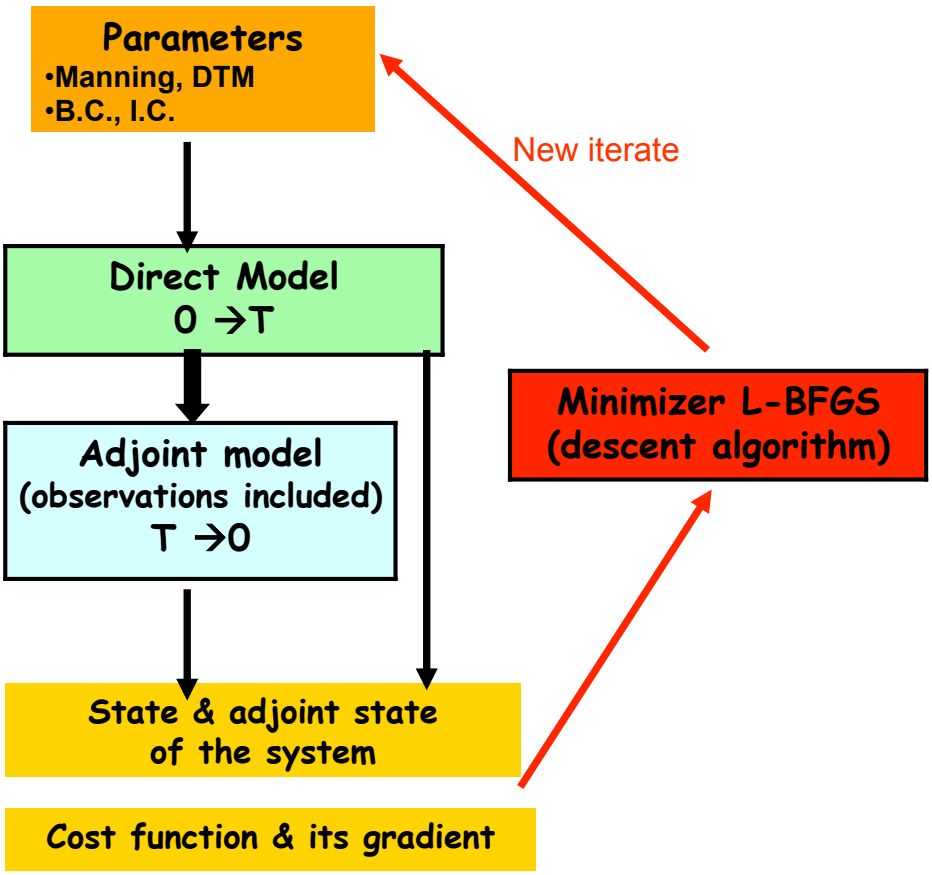
$$\frac{\partial j}{\partial v}(\mathbf{k}) = \left[\frac{\partial A}{\partial v}\Big(y(\mathbf{k}), v\Big)\right]^* \tilde{y}(\mathbf{k})$$

The Optimality System is

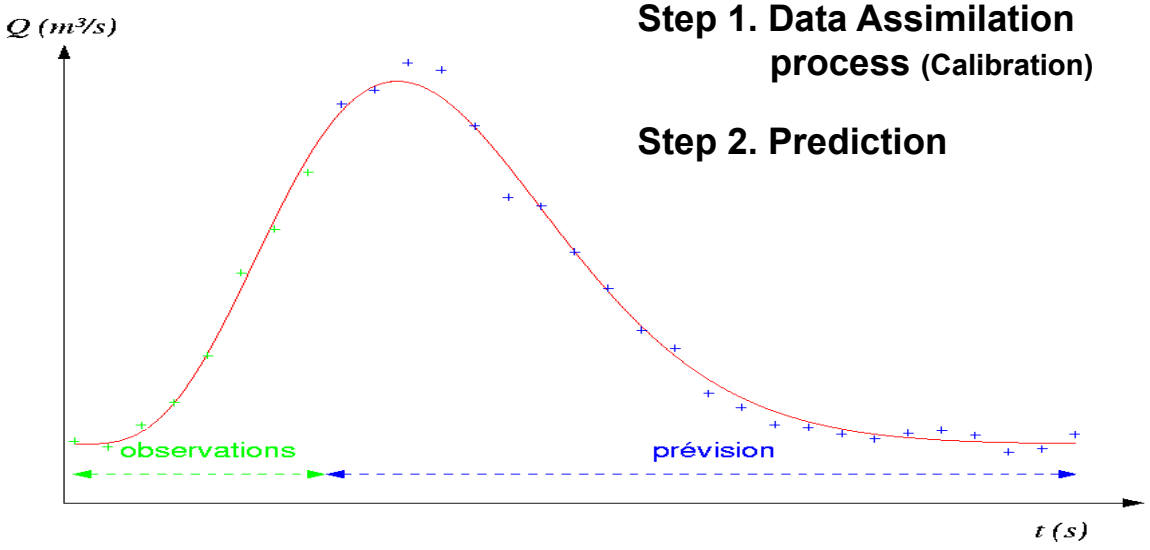
$$\begin{cases} \text{State equation} \\ \text{Ajoint State equation (contains observations)} \\ \nabla j(k) = 0 \end{cases}$$

Ref. e.g.
[Le Dimet – Talagrand '86]
[Talagrand-Courtier'87]

Variationnal Data Assimilation (4D-var): optimal control process



Data Assimilation Principle



Step 1. Data Assimilation process (Calibration)

Step 2. Prediction

Remarks

- The gradient gives local information.
  One run of the forward+adjoint models (without optimization) gives a **local sensitivity information**
- One can control the boundary conditions similarly

What Adjoint Code?
Differentiate / Discretize / Implement : in what order !?

- Continuous gradient:
  1. Differenciate mathematically; 2. discretize; 3. implement

- Discrete gradient:
  1. discretize; 2. differenciate; 3. implement

Make figure

- Code differenciation:
  1. Implement; 2. Differentiate the code (Automatic Differentiation tool e.g. Tapenade)

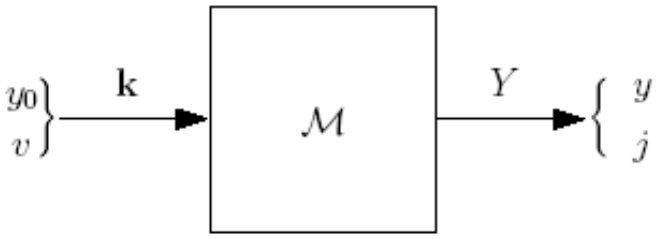
If the forward code well suited, prefer Automatic Differentiation (more re-usable).
NB. A.D. produces the gradient of the computed cost function

From Equations to Automatic Differentiation
Tapenade: source-to-source transformation

Ref.
M. Honnorat
PhD ' 07

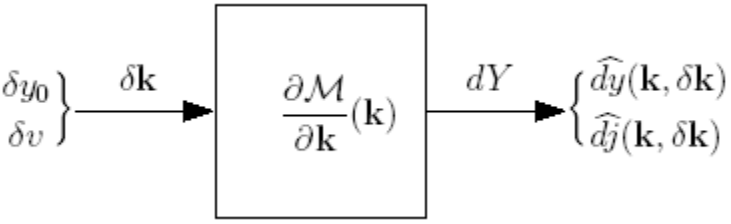
Forward model
(contains the cost function)

$$Y = \mathcal{M}(\mathbf{k}) \ .$$



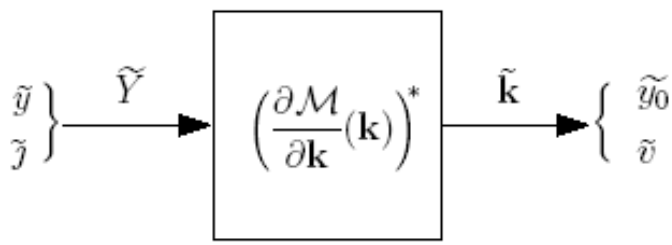
Linearized tangent model

$$dY = \left(\frac{\partial \mathcal{M}}{\partial \mathbf{k}}(\mathbf{k})\right) \cdot \delta\mathbf{k}$$



Adjoint model

$$\widetilde{\mathbf{k}} = \left(\frac{\partial \mathcal{M}}{\partial \mathbf{k}}\right)^* \cdot \widetilde{Y}$$



From Equations to Automatic Differentiation (continued)

Definition of the adjoint model: For all $\delta k \in \mathcal{K}, \tilde{Y} \in \mathcal{Y}$,

$$\left\langle \tilde{Y}, (\tfrac{\partial \mathcal{M}}{\partial \mathbf{k}}) \cdot \delta \mathbf{k} \right\rangle_{\mathcal{Y}} = \left\langle (\tfrac{\partial \mathcal{M}}{\partial \mathbf{k}})^* \cdot \tilde{Y}, \delta \mathbf{k} \right\rangle_{\mathcal{K}}$$

Thus,

$$\langle \tilde{Y}, dY \rangle_{\mathcal{Y}} = \langle \tilde{\mathbf{k}}, \delta \mathbf{k} \rangle_{\mathcal{K}}$$

If we set $\tilde{Y} = (0,1)$ we get $\left\langle \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} \widehat{dy} \\ \widehat{dj} \end{pmatrix} \right\rangle_{\mathcal{Y}} = \left\langle \begin{pmatrix} \tilde{y_0} \\ \tilde{v} \end{pmatrix}, \begin{pmatrix} \delta y_0 \\ \delta v \end{pmatrix} \right\rangle_{\mathcal{X}}$

This means:

$$\widehat{dj}(\mathbf{k}, \delta \mathbf{k}) = \frac{\partial j}{\partial y_0}(\mathbf{k}) \cdot \delta y_0 + \frac{\partial j}{\partial v}(\mathbf{k}) \cdot \delta v = \langle \tilde{y_0}, \delta y_0 \rangle_H + \langle \tilde{v}, \delta v \rangle_{\mathcal{U}}$$

Therefore the response of the adjoint code is the gradient

$$\frac{\partial j}{\partial y_0}(\mathbf{k}) = \tilde{y_0} \qquad \frac{\partial j}{\partial v}(\mathbf{k}) = \tilde{v}$$

Principles of « Automatic Differentiation »
Tapenade: source-to-source transformation

The « A.D. » (Algorithm Differentiation in fact) consists to compute derivatives of a given function M.
The derivatives are exact modulo rounding errors.

Ref. Hascoet et al. Tapenade tool INRIA Tropics

Given the control variable $(\mathbf{k}_\# \subset X_0)$, we have the forward code :

$$Y_\# = \mathcal{M}_\#(\mathbf{k}_\#) = m_p(X_{p-1}) \circ m_{p-1}(X_{p-2}) \circ \cdots \circ m_1(X_0)$$

Then in a direction $\delta \mathbf{k}_\#$

$$\frac{\partial \mathcal{M}_\#}{\partial \mathbf{k}_\#}(\mathbf{k}_\#) \cdot \delta \mathbf{k}_\# = m'_p(X_{p-1}) \times m'_{p-1}(X_{p-2}) \times \cdots \times m'_1(X_0) \cdot \delta \mathbf{k}_\#$$

where m'_p is the jacobian of the elementary operation m_p.

The tangent code value is computed from right to left (matrix-vector products).

This can be inserted into the original source code.

This is the « forward mode » or « tangent mode » of the A.D. tool

Principles of A.D. (continued)

Given the adjoint input variable $\widetilde{Y}_{\#}$, we have the adjoint code :

$$\left(\frac{\partial \mathcal{M}_{\#}}{\partial \mathbf{k}_{\#}}(\mathbf{k}_{\#})\right)^T \times \widetilde{Y}_{\#} = {m'_1}^T(X_0) \times {m'_2}^T(X_1) \times \cdots \times {m'_p}^T(X_{p-1}) \cdot \widetilde{Y}_{\#}$$

This can be computed from right to left (matrix-vector products).
Nevertheless,
• the code is reverse way
• all the variables have to be either stored or all recomputed
→ A balance between storing & recomputing is done (« checkpointing »)

This cannot be inserted
into the original source code !

This is the « adjoint mode »
or « reverse mode »of the A.D. tool

Source-to-source transformation: an extra source code is (automatically) generated

Tapenade On-line Differentiation engine.
Current version: all Fortran 95 excepted pointers & dynamic allocates

An example. The « tangent » mode

Ref. Hascoet et al.
Tapenade tool
INRIA Tropics

Instruction élémentaire du code initial (code direct): $a_i = x\, b_j + \cos(a_i).$

Ce qui peut s'écrire sous la forme:

$$m_k : \mathbb{R}^3 \longrightarrow \mathbb{R}^3$$

$$X_{k-1} = \begin{pmatrix} \texttt{a(i)} \\ \texttt{b(j)} \\ \texttt{x} \end{pmatrix} \longmapsto X_k = \begin{pmatrix} \texttt{a(i)} \\ \texttt{b(j)} \\ \texttt{x} \end{pmatrix}$$

On pose: $\delta X_k = m'_k(X_{k-1}) \cdot \delta X_{k-1}$ $\qquad m'_k(X_{k-1}) = \begin{pmatrix} \texttt{-SIN(a(i))} & \texttt{x} & \texttt{b(j)} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

alors:

$$\begin{pmatrix} \texttt{ad(i)} \\ \texttt{bd(j)} \\ \texttt{xd} \end{pmatrix} = \begin{pmatrix} \texttt{-SIN(a(i))} & \texttt{x} & \texttt{b(j)} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \texttt{ad(i)} \\ \texttt{bd(j)} \\ \texttt{xd} \end{pmatrix}$$

code tangent : `ad(i) = -ad(i)*SIN(a(i)) + x*bd(j) + xd*b(j)`

Tangent code:
•Additional derivative instructions are inserted just before each instruction of the original code.
This leads to a source code quite easy to read.
•Differentiated variables retain the type and dimension of the original variable
•The code can be used either to compute the derivative in ONE direction given,
or to compute for ex. the jacobian of the flux (Newton algorithm)

An example. The « adjoint » mode

$$a_i = x\, b_j + \cos(a_i).$$

On a: $\widetilde{X}_{k-1} = m_k'^{T}(X_k) \cdot \widetilde{X}_k^{*}$

D'où

$$\begin{pmatrix} ab(i) \\ bb(j) \\ xb \end{pmatrix} = \begin{pmatrix} -SIN(a(i)) & 0 & 0 \\ x & 1 & 0 \\ b(j) & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} ab(i) \\ bb(j) \\ xb \end{pmatrix}$$

code adjoint :

```
xb    = xb + b(j)*ab(i)
bb(j) = bb(j) + x*ab(i)
ab(i) = -SIN(a(i))*ab(i)
```

Instructions are computed in reverse order.
Adjoint code: much more complicated to generate and the source code is extremely difficult to read.
CPU time ~2-3 times higher

Principles of D.A. in C++: operator overloading. Tool of ref.: Adol-C.
Nevertheless, very hard in « adjoint» mode…

If and For statements

This includes functions like min, |.| etc

Direct mode

A; if (bool) then B else C; D;
1st branch: A; B; D . This gives: A'; A; B'; B; D'; D
2nd branch: A; C; D. This gives: A'; A; C'; C; D'; D
Hence:
A'; A; if (bool) then {B'; B} else {C' ; C} end if; D'; D;

Similarly A; for i=1:N, do B(i); D;

This gives: A'; A; for i=1:N, do {B'(i); B(i)}; D'; D;

Differentiable or not?
If the var. bool and/or the bounds of the for statement depend on the control var: not differentiable !
In practice, programs are piecewise differentiable; and non-differentiable points (rare !?) should be identified by users …

In adjoint mode:
- branches of IF statements are treated separately, then the results are assembled after the derivation.
- we have to follow the graph of the program in reverse order
  => mind to have an equilibrated original call graph !
- all « intermediate solutions » (for all time step) are a-priori stored
  => very large memory required

=> Snapshot and checkpointing strategy

Snapshot and checkpointing in Tapenade

Store All (SA) strategy. Structure of the adjoint code:

1. **A forward sweep**, identical to the original program augmented with instructions that memorize the intermediate values before they are overwritten, and instructions that memorize the control flow in order to reproduce the control flow in reverse during the **backward sweep**.

2. **Start the backward sweep**, that actually computes the derivatives, and uses the memorized values both to compute the derivatives of each instruction and to choose its flow of control.

Active var: v1, v2, v3, v4

Original code

```
...
v2 = 2*sin(v1) + 5
if (v2>0.0) then
                v4 = v2 + p1*v3/v2
                p1 = p1 + 0.5
endif
call sub1(v2,v4,p1) ...
```

Forward Sweep

```
...
push(v2)
v2 = 2*sin(v1) + 5
if (v2>0.0) then
   push(v4) ; v4 = v2 + p1*v3/v2
   push(p1) ; p1 = p1 + 0.5
   push(true)
else push(false)
endif
push(v2) ; push(v4) ; push(p1)
call sub1(v2,v4,p1) ...
```

The isolated push are memorizations of intermediate values and control.

The group of 3 push before the subroutine call is a **snapshot** because the call will be **checkpointed**.

Checkpointing and snapshot in Tapenade

Differentiated var retain the type and dimension of the original variables

Forward Sweep

```
...
push(v2)
v2 = 2*sin(v1) + 5
if (v2>0.0) then
   push(v4) ; v4 = v2 + p1*v3/v2
   push(p1) ; p1 = p1 + 0.5
   push(true)
else push(false)
endif
push(v2) ; push(v4) ; push(p1)
call sub1(v2,v4,p1) ...
```

Backward Sweep

```
...
pop(p1) ; pop(v4) ; pop(v2)
call sub1_b(v2,v2b,v4,v4b,p1)
pop(control)
if (control) then
   pop(p1) pop(v4)
   v3b = v3b + v4b*p1/v2
   v2b = v2b + v4b*(1-p1*v3/(v2*v2))
   v4b = 0.0
endif
pop(v2)
v1b = v1b + v2b*2*cos(v1)
v2b = 0.0 ...
```

Remarks

- The adjoint of one original instruction is very often a sequence of instructions.
- sub1_b is the result of checkpointing on sub1, and therefore itself consists of a **forward sweep** followed by a **backward sweep**.
- Intrinsic functions are directly differentiated : no call to a sin_b procedure but to cos(v1).

6.4 Exercices

Let us consider a computational Fortran code solving a very simple dynamic system: the scalar linear ODE order one: $y'(t) = f(y(t)u(t))$ with $f(y(t), u(t) = ay(t)+u(t)$, with initial condition. $y(t)$ is the state of the system, while $u(t)$ is the control.

The goal here is to differentiate first by hand the source code (algorithmic differentiation), next using Tapenade software (automatic differentiation).

The direct source code is the following.

```
! Basic program designed to be differentiated using Tapenade
! The head of the tree to be differentiated by Tapenade is
! the first routine of computational part
program dynamic_system
  implicit none
  integer i
  real*8 :: dt, T0, TN, y0
! all allocate must be in the main program
! (out of the computational part which will be diffrentiated)
  real*8, dimension(:), allocatable :: u, y
  integer :: nstep
  T0 = 0.d0
  TN = 1.d0
  !time step (should be defined by the stability condition)
  dt = 0.001d0
  nstep = int((TN-T0)/dt)
  !allocate instructions must be out of the part differentiated by Tapenade
  allocate(y(nstep))
  allocate(u(nstep))
  !initial state and initial control
  !here they are constant (u could be a time dependent vector)
  y0 = 3.4d0
  u  = 2.6d0

  !head routine of the computational part
  !=>the top tree routine of the code thar will be differentiated by Tapenade
  call solve_model( y0, u, y, dt, nstep)

end program dynamic_system

!The direct model is:
!y'(t) = f(y(t)u(t)) + I.C.
!Scheme: Euler explicit
subroutine solve_model( y0, u, y, dt, nstep )
  implicit none
```

```
  integer, intent(in) :: nstep
  real*8,  intent(in) :: dt, y0
  real*8,  dimension(nstep), intent(in)  :: u
  real*8,  dimension(nstep), intent(out) :: y
  real*8 :: f
  integer :: i
  ! initialize the state variable
  y(1) = y0
  ! time loop
  do i = 1, nstep - 1
     y(i+1) = y(i) + dt * f(y(i),u(i))
  end do
  return
end subroutine solve_model

! Right-hand side f
! f(y(t),u(t)) = a y(t) + u(t)
real*8 function f(y,v)
  implicit none
  real*8, intent(in)  :: y, v
  real*8, parameter :: a = 1
  f =  a * y + v
end function f
```

Exercice 6.4.1. *Algorithmic differentiation by hand.*
1) Write the tangent linear codes of the subroutine solve_model and function f.
2) Write the corresponding adjoint codes.

Exercice 6.4.2. *Algorithmic differentiation using Tapenade.*
1) Add in the direct code the computation of a cost function j.
2) Use Tapenade software (web interface on-line) in order to obtain:

a) *the tangent linear codes of the subroutine solve_model and function f.*

b) *the corresponding adjoint codes.*

3) Read carefully and analyze the two routines automatically generated.

6.5 Validation of an adjoint code

We describe below how we check the validity of the adjoint code. Classicaly, we check that it is actually the adjoint of the tangent linear code (scalar product test) and that it computes correctly the partial derivative of the cost function (gradient test).

6.5.1 Scalar product test

The objective of this test is to check if the adjoint code is actually the adjoint of the tangent linear code. In other words, we check the relation (6.1) :

- Given an arbitrary $d\mathbf{k} \in \mathcal{K}$, we compute using the tangent linear code :
$$dY = \left(\frac{\partial \mathcal{M}}{\partial \mathbf{k}}\right) \cdot d\mathbf{k}$$
- Given an arbitrary $dY^* \in \mathcal{Y}$, we compute using the adjoint code :
$$d\mathbf{k}^* = \left(\frac{\partial \mathcal{M}}{\partial \mathbf{k}}\right)^* \cdot dY^*$$
- Then, we compute the following scalar products :
  - $sp_1 = \left\langle dY^*, dY \right\rangle_{\mathcal{Y}}$
  - $sp_2 = \left\langle d\mathbf{k}^*, d\mathbf{k} \right\rangle_{\mathcal{K}}$
- And we check if $sp_1 = sp_2$ or not.

Figure 6.4 (b) shows an example of the scalar product test.

```
####################################
##    TEST DU PRODUIT SCALAIRE    ##
####################################

Appel du code linaire tangent...
Appel du code adjoint...
<Xd,Xb> =   -682.277083033688
<Yd,Yb> =   -682.277082428555
relative error :  -8.869324203484993E-010
```

Figure 6.4: Adjoint code validation: scalar product test

6.5.2 Gradient test

The objective of this test is to check if the adjoint variables dX^* computed by the adjoint code correspond to the partial derivatives of the cost function.
The Taylor expansion of the cost function j at $\mathbf{k}$ for a small perturbation $\alpha\,\delta\mathbf{k}$ (where $\alpha \in \mathbb{R}^+$) writes :

$$j(\mathbf{k} + \alpha\,\delta\mathbf{k}) \;=\; j(\mathbf{k}) + \alpha\,\frac{\partial j}{\partial \mathbf{k}}(\mathbf{k}) \cdot \delta\mathbf{k} + o\left(\alpha \|\delta\mathbf{k}\|\right) . \tag{6.3}$$

Then, we obtain either the uncentered finite difference approximation (order 1) or the centered finite difference approximation order 2:

$$\frac{j(\mathbf{k} + \alpha\,\delta\mathbf{k}) - j(\mathbf{k} - \alpha\,\delta\mathbf{k})}{2\alpha} \;= \frac{\partial j}{\partial \mathbf{k}}(\mathbf{k}).\delta\mathbf{k} + O\left(\alpha^2 \|\delta\mathbf{k}\|^2\right) . \tag{6.4}$$

We set either

$$I_\alpha \;=\; \frac{j(\mathbf{k} + \alpha\,\delta\mathbf{k}) - j(\mathbf{k} - \alpha\,\delta\mathbf{k})}{2\alpha\,\frac{\partial j}{\partial \mathbf{k}}(\mathbf{k}) \cdot \delta\mathbf{k}} . \tag{6.5}$$

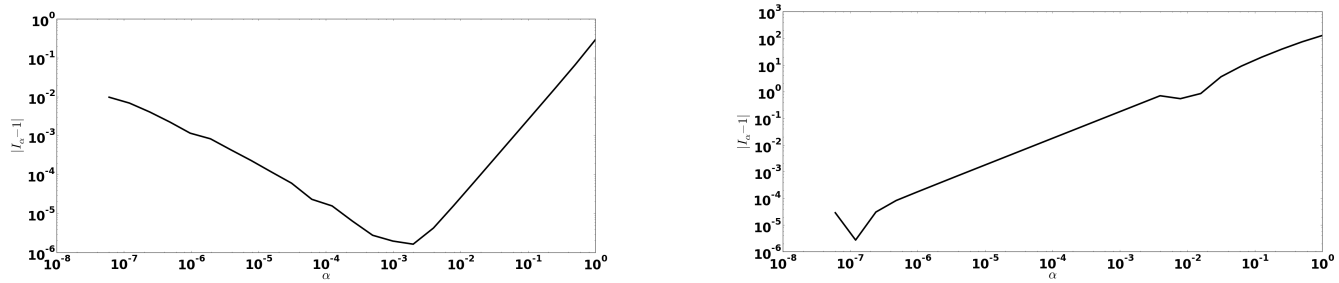
or:

$$I_\alpha \;=\; \frac{j(\mathbf{k}+\alpha\,\delta\mathbf{k})-j(\mathbf{k})}{\alpha\;\frac{\partial j}{\partial \mathbf{k}}(\mathbf{k})\cdot\delta\mathbf{k}}\;. \tag{6.6}$$

According to (6.3), one must have: $\lim_{\alpha\to 0} I_\alpha = 1$.
The gradient test consists to check this property:

- For an arbitrary $\mathbf{k}$, we compute $\frac{\partial j}{\partial \mathbf{k}}(\mathbf{k})$ with the adjoint code.
- With the direct code, we compute $j(\mathbf{k})$.
- For $n = 0, \ldots, N$:
  - We compute $\alpha = 2^{-n}$;
  - With the direct code, we compute $j(\mathbf{k}+\alpha\,\delta\mathbf{k})$;
  - We compute I_α ;
- We check if $\lim_{\alpha\to 0} I_\alpha = 1$ or not.

Figure 6.5 shows two results of the gradient test: at order 2 and at order 1 (observe the difference of accuracy reached).
$|I_\alpha - 1|$ is plotted against α in logarithmic scale. The convergence is good until $\alpha > 10^{-7}$. Then, when α is smaller, the approximation of the partial derivatives is not reliable anymore due to truncation errors (to show it, add a fix term in the Taylor expansion, then it is divided by the perturbation hence increasing).



(a) (b)

Figure 6.5: Adjoint code validation. Gradient test at order 1 (a), at order 2 (b).

6.6 Recycle your linear solver

Let us recall that, roughly, the adjoint of a linear system is the same linear system but transposed. (Notice that the latter feature brings some quite heavy difficulties...). Thus, one should not derive the linear solver instructions one-by-one using an automatic differentiation tool (source-to-source) but instead recycle it ! Here, we describe how to generate the adjoint of a

routine containing a call to a linear solver.

Next, we present a small Fortran program including the call to a linear solver; we apply the method on the codes automatically generated by Tapenade.

6.6.1 How to adjoint-ize your linear solver ?

The direct routine

Let us consider two input parameters c and d (hence *active variables* in Tapenade terminology). Then, we have A matrix and b vector defined as follows:

$$\begin{pmatrix} A \\ b \end{pmatrix} = f(c,d) = \begin{pmatrix} f_1(c,d) \\ f_2(c,d) \end{pmatrix}$$

Let x be the vector solution of the linear system: $Ax = b$. Next, we compute the cost function j.

We present in Fig. 6.6 the direct routines dependencies.

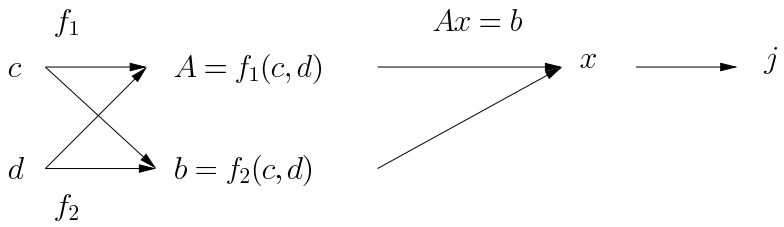


Figure 6.6: Direct routines representation

The linear tangent routine generated

The linear tangent code derived by Tapenade is defined as follows:

- $\dot{c}$ and $\dot{d}$ are the corresponding tangent variables of the parameters, they are input parameters.
- $\dot{A}$ and $\dot{b}$ satisfy:

$$\begin{pmatrix} \dot{A} \\ \dot{b} \end{pmatrix} = df(c,d) \cdot \begin{pmatrix} \dot{c} \\ \dot{d} \end{pmatrix}$$

Next, we have to treat the linear solver. If we differentiate the linear system:

$$Ax = b \tag{6.7}$$

we find: $A\dot{x} + \dot{A}x = \dot{b}$ or, after rewrite:

$$A\dot{x} = \dot{b} - \dot{A}x \tag{6.8}$$

At this stage, we already know A and x by running direct routine. The variables $\dot{A}$, $\dot{b}$ are output variables of the Tapenade tangent linear routines. Next, we call the linear solver in order to obtain: $\dot{x}$. The latter represents the derivative value of x at point (c, d) in the direction $(\dot{c}, \dot{d})$ given.
Thus, we are able to compute $\dot{x}$ without deriving the linear solver instructions. We can call directly the same linear solver (we recycle it).
Next, we obtain $\dot{j}$ the gradient value at point (c, d) in the direction $(\dot{c}, \dot{d})$ given (it is a scalar value like j).
We present in Fig. 6.7 the scheme representing the linear tangent routines.

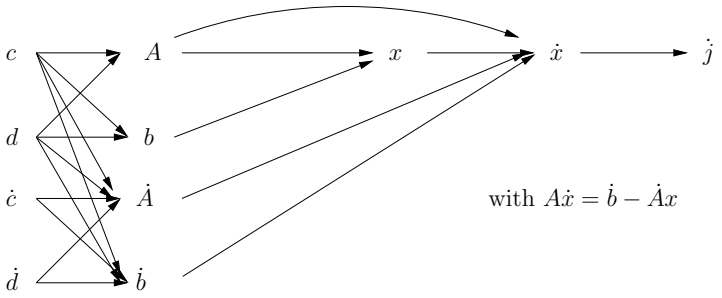


Figure 6.7: Linear tangent routines representation

The adjoint routines generated

Let us recall that the adjoint code is deduced from the tangent linear code, in the reverse order. The output variable of the adjoint routine are $\bar{c}$ and $\bar{d}$. In a Tapenade point of view, they are the adjoint variables of (c, d) (same type); while in a mathematical point of view, they are the *gradient* with respect to c and d.
The input variable of the adjoint cost computation routine is $\bar{\dot{j}}$. Let us recall that $\bar{\dot{j}}$ must be set to 1, see Section 6.2.
The input variable of the adjoint linear system routine is $\bar{x}$.

The adjoint code can be splitted in three steps, see Fig. 6.7:

1) From $\bar{\dot{j}}$ to $\bar{x}$ (let us assume it is a separated routine: cost computation, adjoint)

2) From $\bar{x}$, find $\bar{A}$ and $\bar{b}$

3) From $\bar{A}$ and $\bar{b}$, find $\bar{c}$ and $\bar{d}$.

The first and third steps can be obtained directly by running the Tapenade adjoint code. The adjoint of the third step writes:

$$\begin{pmatrix} \bar{c} \\ \bar{d} \end{pmatrix} = df^*(c, d) \cdot \begin{pmatrix} \bar{A} \\ \bar{b} \end{pmatrix} \tag{6.9}$$

The adjoint of the linear system

The second step involves a call to the linear solver. Thus, let us detail how to compute $\bar{A}$ and $\bar{b}$ with the use of a linear solver as a black box. In other words, we detail below the adjoint of Step 2) only.
Input variable is $\bar{x}$, while output variables are $\bar{A},\ \bar{b}$.
In the linear tangent code, we have : $A\dot{x} = \dot{b} - \dot{A}x$ or equivalently:

$$\begin{array}{lrcl} 2a) & \dot{b'} & = & \dot{b} - \dot{A}x \\ 2b) & A\dot{x} & = & \dot{b'} \end{array}$$

The adjoint is reverse. Thus let us consider first the instruction $2b)$ $A\dot{x} = \dot{b'}$ only. It can be written as follows (see e.g. [12]):

$$\begin{pmatrix} \dot{x} \\ \dot{b'} \end{pmatrix} = \begin{pmatrix} 0 & A^{-1} \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} \dot{x} \\ \dot{b'} \end{pmatrix} \tag{6.10}$$

Two fundamental remarks.
a) Let us point out that by convention, at left-hand sides are the variables at output, while at right-hand sides are the variables at input (of the routine).
b) Let us point out that input variables in the linear tangent routine become output variables in the adjoint routine. Furthermore, by convention, output variables are initialized at 0.

Let us write the adjoint of the instruction $2b)$. Since $\dot{b'}$ is the input variable of the instruction, its adjoint $\bar{b'}$ will be the output one (and set at 0 when entering into the routine). Similarly, since $\dot{x}$ is the output variable, its adjoint $\bar{x}$ will be the input one. The adjoint instruction of (6.10) writes:

$$\begin{pmatrix} \bar{x} \\ \bar{b'} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ A^{-T} & 1 \end{pmatrix} \times \begin{pmatrix} \bar{x} \\ \bar{b'} \end{pmatrix}$$

Hence:

$$\left\{ \begin{array}{lcl} \bar{x} & = & 0 \\ \bar{b'} & = & A^{-T}\bar{x} + \bar{b'} \end{array} \right.$$

Thus the system rewrites:

$$\left\{ \begin{array}{lcl} A^T\bar{b'} & = & \bar{x} \\ \bar{x} & = & 0 \end{array} \right.$$

And one has to solve the linear system: $A^T\bar{b'} = \bar{x}$. It can be done using the linear solver called as a black box (we recycle it !). It gives $\bar{b'}$.

Now, let us consider the adjoint of the instruction 2a):

$$2a)\ \ \dot{b'} = \dot{b} - \dot{A}x \tag{6.11}$$

This linear instruction writes as follows:

$$(\dot{b'}, \dot{b}, \dot{A}) = (\dot{b'}, \dot{b}, \dot{A}) \times \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ -x & 0 & 1 \end{pmatrix}$$

The corresponding adjoint instruction writes:

$$(\bar{b}', \bar{b}, \bar{A}) = (\bar{b}', \bar{b}, \bar{A}) \times \begin{pmatrix} 0 & 1 & -x^T \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Therefore, the adjoint instruction of (6.11) writes:

$$\left\{ \begin{array}{rcl} \bar{b}' & = & 0 \\ \bar{b} & = & \bar{b}' + \bar{b} \\ \bar{A} & = & -\bar{b}' x^T + \bar{A} \end{array} \right.$$

The variables $\bar{A}$, $\bar{b}$ are output variables, hence set at 0 when entering into the adjoint routine.

Therefore, in summary the adjoint of the tangent linear instructions $A\dot{x} = \dot{b} - \dot{A}x$ (ie Step 2)) can be re-write:

$$\left\{ \begin{array}{rcl} A^T \bar{b} & = & \bar{x} \\ \bar{A} & = & -\bar{b} x^T \\ \bar{x} & = & 0 \end{array} \right.$$

The first instruction can be solved using the same linear solver than the direct routine ("recycle your linear solver"). The second instruction writes too: $\bar{A}_{ij} = -\bar{b}_i x_j$.

Remark 6.6.1.
- Let us notice that the matrix $\bar{A}$ is the same type of A with the same sparse profile (even if $-\bar{b}x^T$ is a-priori a full matrix). As a matter of fact, one needs only the adjoint values of coefficients $A_{i,j}$ (and one do not need the others coefficients).

- Let us recall that $(\bar{c}, \bar{d})$ are the components of the gradient with respect to (c, d) respectively; they are obtained by (6.9).

- Since the direct model is $Ax = b$, the state of the system is x, the adjoint state is $\bar{b}$ (the solution of the transposed linear system) and $\bar{x}$ is the right-hand side which contains the observations and misfit terms (it equals to $(x - x^{obs})$ if the observation operator equals identity).

We summarize all steps in Fig. 6.8.

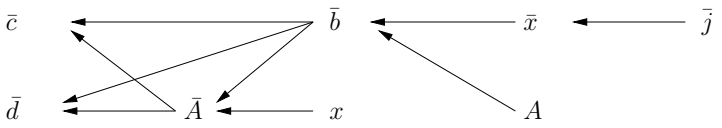


Figure 6.8: Adjoint routine representation

6.6.2 A simple example

We illustrate the previous procedure of the linear solver recycling on a very simple example. In all the following, one can replace the routine *sys_solver* by any linear system solver routine. This example has been written by R. Madec, research engineer, IMT Toulouse.

Direct code

We present the Fortran source code of the direct subroutine written in file *simple_operation.f90*:

```
subroutine simple_operation(c,d,x)
  implicit none
  integer, parameter :: n=2
  double precision, intent(in) :: c
  double precision, intent(in) :: d
  double precision, dimension (n), intent(out):: x
  double precision, dimension (n,n) :: A
  double precision, dimension (n)   :: b

!definition of matrix A
  A(1,1) = 3*c +4*d
  A(1,2) = c +18*d
  A(2,1) = 2.5*d
  A(2,2) = 3.4*c

!definition of vector b
  b(1) = 8*c*d
  b(2) = 2*d

!solver Ax=b
  call sys_solver(A,b,x,n)

end subroutine simple_operation
```

Linear tangent code

We generate the linear tangent code using Tapenade. To do so, we type in a terminal (or in a Makefile):

```
tapenade -forward -head simple_operation \
    -vars "c d" -outvars "x" \
    -html -O $(PWD)/forward simple_operation.f90
```

Then, the file *simple_operation_d.f90* is created in the *forward* directory. This file is the following:

```
!        Generated by TAPENADE     (INRIA, Tropics team)
```

```
!  Tapenade 3.4 (r3375) - 10 Feb 2010 15:08
!
!  Differentiation of simple_operation in forward (tangent) mode:
!   variations   of useful results: x
!   with respect to varying inputs: c d
!   RW status of diff variables: x:out c:in d:in
SUBROUTINE SIMPLE_OPERATION_D(c, cd, d, dd, x, xd)
  IMPLICIT NONE
  INTEGER, PARAMETER :: n=2
  DOUBLE PRECISION, INTENT(IN) :: c
  DOUBLE PRECISION, INTENT(IN) :: cd
  DOUBLE PRECISION, INTENT(IN) :: d
  DOUBLE PRECISION, INTENT(IN) :: dd
  DOUBLE PRECISION, DIMENSION(n), INTENT(OUT) :: x
  DOUBLE PRECISION, DIMENSION(n), INTENT(OUT) :: xd
  DOUBLE PRECISION, DIMENSION(n) :: b
  DOUBLE PRECISION, DIMENSION(n) :: bd
  DOUBLE PRECISION, DIMENSION(n, n) :: a
  DOUBLE PRECISION, DIMENSION(n, n) :: ad
  INTEGER :: i,j

  ad(1, 1) = 3*cd + 4*dd
  a(1, 1) = 3*c + 4*d
  ad(1, 2) = cd + 18*dd
  a(1, 2) = c + 18*d
  ad(2, 1) = 2.5*dd
  a(2, 1) = 2.5*d
  ad(2, 2) = 3.4*cd
  a(2, 2) = 3.4*c
  bd(1) = 8*(cd*d+c*dd)
  b(1) = 8*c*d
  bd(2) = 2*dd
  b(2) = 2*d

  CALL SYS_SOLVER_D(a, ad, b, bd, x, xd, n)

END SUBROUTINE SIMPLE_OPERATION_D
```

Using the method presented in subsection 6.6.1, we replace the unknown linear tangent of the linear system solver:

```
  CALL SYS_SOLVER_D(a, ad, b, bd, x, xd, n)
```

by

```
  CALL SYS_SOLVER(a, b, x, n)
```

```
  !matrix vector product : bd = bd - Ad x
  do j=1,n
     do i=1,n
       bd(i)=bd(i) - ad(i,j)*x(j)
     end do
  end do
  CALL SYS_SOLVER(a, bd, xd, n)
```

Adjoint code

We generate the adjoint code using Tapenade. To do so, we type in a terminal (or in a Makefile):

```
tapenade -backward -head simple_operation \
    -vars "c d" -outvars "x" \
    -html -O $(PWD)/backward simple_operation.f90
```

Then, the file *simple_operation_b.f90* is created in the *backward* directory. This file is the following:

```
!        Generated by TAPENADE     (INRIA, Tropics team)
!  Tapenade 3.4 (r3375) - 10 Feb 2010 15:08
!
!  Differentiation of simple_operation in reverse (adjoint) mode:
!   gradient     of useful results: x
!   with respect to varying inputs: x c d
!   RW status of diff variables: x:in-zero c:out d:out

SUBROUTINE SIMPLE_OPERATION_B(c, cb, d, db, x, xb)
  IMPLICIT NONE
  INTEGER, PARAMETER :: n=2
  DOUBLE PRECISION, INTENT(IN) :: c
  DOUBLE PRECISION :: cb
  DOUBLE PRECISION, INTENT(IN) :: d
  DOUBLE PRECISION :: db
  DOUBLE PRECISION, DIMENSION(n) :: x
  DOUBLE PRECISION, DIMENSION(n) :: xb
  DOUBLE PRECISION, DIMENSION(n, n) :: a, at
  DOUBLE PRECISION, DIMENSION(n, n) :: ab
  DOUBLE PRECISION, DIMENSION(n) :: b
  DOUBLE PRECISION, DIMENSION(n) :: bb
  INTEGER :: i,j
  a(1, 1) = 3*c+ 4*d
  a(1, 2) = c + 18*d
  a(2, 1) = 2.5*d
  a(2, 2) = 3.4*c
  b(1) = 8*c*d
```

```
  b(2) = 2*d

  CALL SYS_SOLVER_B(a, ab, b, bb, x, xb, n)

  db = 2*bb(2)
  bb(2) = 0.D0
  cb = 3.4*ab(2, 2) + 8*d*bb(1)
  ab(2, 2) = 0.D0
  db = db + 2.5*ab(2, 1) + 8*c*bb(1)
  ab(2, 1) = 0.D0
  cb = cb + ab(1, 2)
  db = db + 18*ab(1, 2)
  ab(1, 2) = 0.D0
  cb = cb + 3*ab(1, 1)
  db = db + 4*ab(1, 1)
  xb = 0.D0
END SUBROUTINE SIMPLE_OPERATION_B
```

Using the method presented in subsection 6.6.1 we replace the unknown adjoint of the linear system solver:

```
  CALL SYS_SOLVER_B(a, ab, b, bb, x, xb, n)
```

by the following instructions:

```
!at = transpose(a)
  at(1, 1) = 3*c + 4*d
  at(2, 1) = c + 18*d
  at(1, 2) = 2.5*d
  at(2, 2) = 3.4*c

!new solver
  call sys_solver(a,b,x,n)
  call sys_solver(at,xb,bb,n)
  ab=0.d0
  do j=1,n
     do i=1,n
 if (ab(i,j).ne.0.) ab(i,j)=-bb(i)*x(j)
     end do
  end do

!At = transpose(A)
  do j=1,n
   do i=1,n
    At(i, j) = A(j,i)
   end do
  end do
```

```
!new solver
  call linear_solver(A,b,x,n)
  call linear_solver(At,xb,bb,n)
  do j=1,n
     do i=1,n
  if (Ab(i,j).ne.0.) Ab(i,j)= - bb(i)*x(j)
     end do
  end do
```

We have recycled the linear solver (which can be called as a black box).

6.7 Complement: MPI instructions

We present the procedure to obtain the adjoint code of a Fortran code calling MPI basic routines and using the automatic differentiation tool Tapenade.
This part is skipped since no time enough.

6.8 Complement: optimize your memory. Checkpointing.

When you use Tapenade to create an adjoint code, you have no mean to know the memory consumption that will take Tapenade. The latter is a direct consequence of the '"PUSH" and "POP" instructions introduced by Tapenade in your code. The "PUSH" instruction is keeping in memory the variable until a "POP" of this variable is done. Every adjoint code is divided into two part: the first part, which is just a copy of your direct code with add of "PUSH" instructions filling the memory; the second part written by Tapenade and running a backward code with the adjoint variables, and freeing the memory with POP instructions.
Memory problems can appear if you have "PUSH"es of, for example, your state variable at every time steps on a simulation.

We present below a way to avoid memory faults by using the checkpointing strategy of Tapenade in a "smart" way.

Checkpointing strategy First thing is to anticipate and identify which part of the "PUSH" and "POP" instructions in your code is potentially greedy in memory. To narrowing it, it should appear in the biggest loop (i.e. with the most iterations) of the algorithm and occur where your state variables are.

An example of order of magnitude. In DassFlow software, based on the 2D shallow-water equations, the main "PUSH" and "POP" contributions are related to the state variable in the time step loop.
For a 25 000 cells mesh with 100 000 time steps, if we do nothing, as our state variable counts 3

fields, this would cost just for "pushing" this variable: $3 \times 25000 \times 100000 \times 8 = 6 \times 10^{10}$o= 60 Go of RAM which is a lot !

(NB. The number 8 represents the octet size of a double precision number; it is 4 for a simple precision).
Most of the time, it is only one variable that make your system fall.