INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques dans un tube chauffant.

Frédéric Eyssette, Christèle Faure, Jean Charles Gilbert, Nicole Rostaing-Schmidt

N° 2795

Février 1996

PROGRAMME 2

Rapport de recherche

ISSN 0249-6399

INRIA
SOPHIA ANTIPOLIS

Applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques dans un tube chauffant.

Frédéric Eyssette[*] , Christèle Faure[*] , Jean Charles Gilbert[**] , Nicole Rostaing-Schmidt[*]

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet SAFIR

Rapport de recherche n° 2795 — Février 1996 — 96 pages

Résumé : Dans ce rapport, nous étudions l'applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques dans un tube chauffant. En particulier, il s'agit de voir si le différentiateur ODYSSÉE développé à l'INRIA dans le projet SAFIR peut différentier un code tel que THYC-1D (maquette à une dimension en espace du code THYC), lequel comporte 23 sous-routines de calcul.

Nous avons d'abord généré les dérivées en mode direct et inverse avec la version 1.4 d'ODYSSÉE. Cette génération routine par routine étant très fastidieuse et ne permettant pas d'assurer la cohérence du code dérivé, nous avons introduit un analyseur interprocédural. Celui-ci permet au différentiateur de considérer en une seule fois un ensemble de routines dont le graphe d'appel forme un arbre. D'autre part, le code de THYC-1D contenant des fonctions tabulées que l'on ne peut pas dériver (leur code-source n'est pas disponible), nous avons introduit une base d'information permettant à ODYSSÉE de dériver les appels d'unités non lues. Les dérivées de ces fonctions ont été approchées par différences finies.

Nous avons alors comparé les dérivées calculées par les deux modes de différentiation d'ODYSSÉE, modes direct et inverse, avec celles approchées par des différences finies utilisant un pas calculé de manière à obtenir le plus grand nombre de chiffres significatifs corrects possible. Les comparaisons ont été faites en simple et en double précision.

Ce travail a été en partie financé par Électricité de France (Contrat EDF T13/1J1751/RNE).

[*] INRIA Sophia Antipolis, e-mail: 'Christele.Faure@inria.fr'.
[**] INRIA Rocquencourt, Projet PROMATH, BP 105, F-78153 Le Chesnay Cedex, e-mail: 'Jean-Charles.Gilbert@inria.fr'.

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

De ces tests, il ressort que l'on peut avoir entre 1 et 3 chiffres significatifs corrects en plus par ODYSSÉE que par ces différences finies "optimales", ces dernières donnant entre 7 et 8 chiffres significatifs corrects en double précision. Ce qui a limité les performances d'ODYSSÉE est la nécessité d'inclure dans le code généré des routines approchant la dérivée des fonctions tabulées par différences finies. D'autre part, les performances d'ODYSSÉE en temps sont conformes à la théorie : le calcul d'une dérivée directionnelle se fait en un temps égal à celui de 2.3 appels de fonction et le calcul d'un gradient en un temps équivalent à 4.9 appels de fonction. De plus, l'encombrement-mémoire en mode inverse est resté suffisamment raisonnable pour permettre l'exécution du code dérivé sur une station de travail.

L'expertise de Carole Duval et de Patrick Erhard d'EDF nous a été précieuse pour nous familiariser avec le code THYC-1D, pour interpréter certains résultats obtenus par ODYSSÉE et pour nous éclairer sur les aspects physiques des phénomènes simulés.

Mots-clé : Code Adjoint, Génération de code, Differentiation Automatique, `Odyssée`

(Abstract: pto)

Applicability of Automatic Differentiation on a system of partial differential equations governing thermohydraulic phenomena.

Abstract: The applicability of automatique differentiation on a set of partial differential equations governing thermohydraulic phenomena in heat exchangers is examined. More specifically, the challenge was to differentiate THYC-1D, a 1-D mockup of the 3-D code THYC implementing these equations, with the automatic differentiator ODYSSÉE with as few manual interventions as possible. The code to differentiate contains 23 routines, including linear solvers and black-box functions, whose code was not available.

Key-words: Adjoint Code, Code Generation, Automatic Differentiation, Computational Derivatives, `Odyssée`.

Table des matières

Chapitre 1

Rappels sur la différentiation automatique

La différentiation automatique [1] est une technique permettant d'obtenir les dérivées exactes (aux erreurs d'arrondi près) d'une fonction représentée par un programme informatique. Dans le cas du différentiateur ODYSSÉE [14], que nous utiliserons dans cette étude, on suppose que le programme est écrit en Fortran. Si c'est un programme qui représente la fonction à dériver, c'est un autre programme qui est généré par le différentiateur. Il faudra faire tourner ce dernier pour obtenir la valeur de la dérivée de la fonction en un point donné.

La méthode se distingue donc de la différentiation symbolique telle qu'on la trouve dans des logiciels comme MACSYMA [19], MAPLE [1] ou REDUCE. Elle se distingue aussi de la différentiation par différences finies, avec laquelle les dérivées ne peuvent être calculées avec précision. Un des attraits de la différentiation automatique est de pouvoir générer un programme calculant le gradient d'une fonction à valeurs scalaires, dont le temps d'exécution relatif à celui du programme original est indépendant du nombre de variables par rapport auxquelles on dérive. Il s'agit en fait de génération automatique de codes adjoints (appelés aussi codes linéaires cotangents).

Dans cette section, nous faisons quelques rappels sur la théorie de la différentiation automatique. Pour plus de détails, on pourra consulter [2, 6, 15].

1.1 Modèle d'un code Fortran

Pour obtenir les formules permettant de générer de manière automatique un code calculant les dérivées d'une fonction représentée par un programme, on commence par considérer le cas d'un programme formé d'une suite d'instructions d'affectation. Un tel programme a l'avantage d'avoir une représentation mathématique simple avec laquelle on peut travailler. Les règles que l'on déduit de ce modèle permettent alors de comprendre les règles de transformation de codes à mettre en œuvre sur des programmes plus complexes, voire généraux, tels que ceux écrits en Fortran. L'utilisation du Fortran n'est en rien obligatoire, d'autres langages peuvent même être

---

[1] En anglais, le vocable *Automatic Differentiation* tend à être remplacé par *Computational Differentiation*. L'analogue français reste à être introduit.

plus appropriés. On peut dire que plus le langage est structuré et ne travaille pas directement sur des cases-mémoire sans structure, plus la tâche du différentiateur est aisée.

On note v_1, v_2, ..., v_N les N variables Fortran sur lesquelles travaille le programme modèle. Il n'y a aucun ordre sur ces variables. En particulier, il ne faut pas que ces variables soient évaluées dans l'ordre de leur indice. C'est simplement une manière commode de les désigner. On supposera que les n *variables d'entrée* ($n \leq N$), celles par rapport auxquelles on dérive, encore appelée *variables indépendantes*, sont les variables v_1, ..., v_n. Les m *variables de sortie* ($m \leq N$), celles que l'on veut dériver, sont supposées être les variables v_{N-m+1}, ..., v_N. Des variables peuvent être à la fois d'entrée et de sortie. On dira qu'une variable v_i devient *active* lorsqu'on lui affecte une valeur qui dépend de la valeur donnée aux variables indépendantes. Enfin, si $v \in \mathbb{R}^N$ et A est une partie de $\{1, \ldots, N\}$, on utilisera la notation v_A pour le vecteur de $\mathbb{R}^{|A|}$ dont les composantes sont les v_i avec $i \in A$.

Le programme modèle que nous considérerons est donc supposé être formé d'une suite de K instructions d'affectation exécutées l'une après l'autre, ce que l'on peut écrire :

$$\boxed{v_{\mu_k} := \varphi_k(v_{D_k}), \quad k = 1, \ldots, K.} \tag{1.1}$$

À chaque instruction k, le programme modifie la variable v_{μ_k} au moyen d'une fonction φ_k, en utilisant les variables v_i, $i \in D_k$, où D_k est une partie de $\{1, \ldots, N\}$. Dans ce modèle, on ne se donne aucune restriction sur $\mu_k \in \{1, \ldots, N\}$, qui peut en particulier faire partie de D_k. Si on note x les variables d'entrée et f les variables de sorties, on cherche donc à différentier une fonction

$$f : x \in \mathbb{R}^n \mapsto f(x) \in \mathbb{R}^m,$$

qui est représentée par le programme modèle (1.1) et dont la valeur en un point x peut être obtenue en faisant "tourner" ce programme.

Pour fixer les idées, considérons l'exemple suivant, dans lequel on calcule une variable $f =$ `v5` à partir de la donnée d'un couple de variables $x =$ (`v1`, `v2`) :

```
v3 = v1 + v2**2
v4 = v1**2 * sin(v3)
v4 = v4/v3
v5 = exp(v4)
```

La correspondance entre ce programme et le modèle (1.1) est détaillée au tableau 1.1.

$N = 5$, $n = 2$, $m = 1$

indice k de l'instruction	instruction	indice de la variable modifiée	fonction φ_k	indices de dépendance
1	`v3=v1+v2**2`	$\mu_1 = 3$	$\varphi_1(a,b) = a + b^2$	$D_1 = \{1,2\}$
2	`v4=v1**2*sin(v3)`	$\mu_2 = 4$	$\varphi_2(a,b) = a^2 \sin(b)$	$D_2 = \{1,3\}$
3	`v4=v4/v3`	$\mu_3 = 4$	$\varphi_3(a,b) = b/a$	$D_3 = \{3,4\}$
4	`v5=exp(v4)`	$\mu_4 = 5$	$\varphi_4(a) = \exp(a)$	$D_4 = \{4\}$

TAB. 1.1 – *Correspondance entre l'exemple de programme et le modèle (1.1)*

On désigne par $\mathcal{F}$ l'ensemble des *fonctions intermédiaires*, c'est-à-dire les fonction φ_k utilisées dans le programme modèle (1.1). Il n'y a pas de restriction sur ces fonctions, si ce n'est qu'elles doivent être différentiables. Elle peuvent représenter les opérations élémentaires (`+`, `-`, `*`, `/`), les fonctions intrinsèques du Fortran, des compositions de ces fonctions ou encore des ensembles d'instructions (sous-routines, fonctions). On notera $\mathcal{F}_F = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{sqrt}, \texttt{exp}, \texttt{log}, \texttt{log10}, \texttt{sin}, \texttt{cos}, \texttt{tan}, \texttt{asin}, \texttt{acos}, \texttt{atan}, \texttt{sinh}, \texttt{cosh}, \texttt{tanh}\}$ la classe des fonctions utilisables en Fortran.

1.2 Mode direct de différentiation (code linéaire tangent)

C'est le mode le plus simple et le plus intuitif. Il est bien adapté au calcul des dérivées directionnelles de la fonction f représentée par le programme.

Soit d une direction de $\mathbb{R}^n$. On cherche à calculer la dérivée directionnelle de f en x dans la direction d, ce que l'on note $f'(x) \cdot d$. Pour cela, il est bon de voir chaque variable du code comme une fonction des variables d'entrée $x = (v_1, \ldots, v_n)$. Donc à chaque v_i correspond une dérivée directionnelle $v_i'(x) \cdot d$. Alors, en différentiant l'instruction k du programme (1.1), on obtient

$$v'_{\mu_k}(x) \cdot d := \sum_{i \in D_k} \frac{\partial \varphi_k}{\partial v_i}(v_{D_k}) v_i'(x) \cdot d.$$

On a donc une formule permettant de calculer $v'_{\mu_k}(x) \cdot d$ à partir des $v_i'(x) \cdot d$, $i \in D_k$. Grâce à celle-ci, on peut “propager” le calcul des dérivées directionnelles des variables évaluées dans le code, parallèlement à leur évaluation. C'est l'idée utilisée par le mode direct de différentiation. On notera

$$\dot{v}_i = v_i'(x) \cdot d,$$

qui est donc un scalaire.

Remarquons que $\dot{v}_i = d_i$, pour $1 \leq i \leq n$. Pour calculer $f'(x) \cdot d$, il suffit donc d'initialiser les dérivées directionnelles des variables indépendantes comme suit :

$$\dot{v}_i := d_i, \quad \text{pour } 1 \leq i \leq n,$$

et ensuite de propager les dérivées directionnelles dans le code par les instructions suivantes :

$$\left.\begin{array}{r} \dot{v}_{\mu_k} := \sum_{i \in D_k} \frac{\partial \varphi_k}{\partial v_i}(v_{D_k}) \dot{v}_i \\ v_{\mu_k} := \varphi_k(v_{D_k}) \end{array}\right\}, \quad k = 1, \ldots, K.$$

On calcule $\dot{v}_{\mu_k}$ avant v_{μ_k} au cas où φ_k dépendrait de v_{μ_k}. Il faut en effet évaluer les dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}$ avec la valeur non modifiée de v_{μ_k}. En fin d'exécution, on récupère dans $\dot{v}_{N-m+1}$, …, $\dot{v}_N$, la dérivée $f'(x) \cdot d$:

$$f'(x) \cdot d := (\dot{v}_{N-m+1}, \ldots, \dot{v}_N).$$

Ce mode de différentiation porte le nom de *mode direct* et le code réalisant ces opérations est appelé *code linéaire tangent* de (1.1). Celui-ci peut être résumé ainsi :

```
pour $i = 1, \ldots, n$
    $\dot{v}_i := d_i$ ;
pour $k = 1, \ldots, K$
    $\dot{v}_{\mu_k} := \sum_{i \in D_k} \frac{\partial \varphi_k}{\partial v_i}(v_{D_k}) \dot{v}_i$ ;
    $v_{\mu_k} := \varphi_k(v_{D_k})$ ;
$f'(x) \cdot d := (\dot{v}_{N-m+1}, \ldots, \dot{v}_N)$ .
```
(1.2)

On voit que l'on a associé à chaque variable v_i du code, une variable $\dot{v}_i$ contenant sa dérivée directionnelle. L'espace-mémoire requis est donc exactement le double de celui utilisé par le programme original, si toutes les variables sont actives. En ce qui concerne le temps de calcul $T(f, f')$ nécessaire à l'exécution de (1.2), on peut le comparer au temps $T(f)$ de l'exécution de (1.1). Par exemple si toutes les fonctions intermédiaires φ_k sont prises dans $\mathcal{F}_F$ (famille des fonctions Fortran), on montre, sous des hypothèses raisonnables sur le temps respectif de l'évaluation des fonctions de $\mathcal{F}_F$, que

$$\frac{T(f(x), f'(x) \cdot d)}{T(f(x))} \leq 4,$$

et ceci quel que soit le nombre m de variables dérivées (voir [6]).

À titre d'illustration, voyons comment s'écrit le code linéaire tangent dans le cas de l'exemple de la page 8. Supposons que les deux composantes de la direction d soient données dans les variables `d1` et `d2`. En utilisant le suffixe `tl` pour désigner les variables $\dot{v}_i$ associées aux variables v_i du code et contenant leur dérivée directionnelle (convention de nom utilisée dans ODYSSÉE), le code linéaire tangent s'écrit :

```
v1tl = d1
v2tl = d2
v3tl = v1tl + 2*v2*v2tl
v3 = v1 + v2**2
v4tl = 2*v1*v1tl*sin(v3) + v1**2*cos(v3)*v3tl
v4 = v1**2 * sin(v3)
aux = v4/v3
v4tl = (v4tl - aux*v3tl)/v3
v4 = aux
aux = exp(v4)
v5tl = aux * v4tl
v5 = aux
```

On a utilisé une variable auxiliaire `aux`, de manière à éviter de calculer plusieurs fois certaines quantités.

Ce qu'il faut retenir :

1. Le mode direct de différentiation est le bon mode pour calculer la dérivée d'un grand nombre de variables (ou variables de sortie) par rapport à un petit nombre de variables indépendantes (ou variables d'entrée).

2. C'est donc aussi le bon mode pour calculer une dérivée directionnelle d'une application $f : \mathbb{R}^n \to \mathbb{R}^m$.
3. À chaque variable v_i du code, on associe une variable $\dot{v}_i$ qui contient la dérivée directionnelle de v_i dans une direction d donnée. Sa valeur est nulle (et alors $\dot{v}_i$ peut ne pas exister dans le code linéaire tangent) si v_i n'est pas influencée par les variables par rapport auxquelles on dérive (c'est-à-dire si elle n'est pas active).
4. Le code linéaire tangent peut s'obtenir en "dérivant" le code original ligne par ligne. Si dans ce dernier une instruction s'interprète comme une application $\{v_i : i \in D_k\} \mapsto v_{\mu_k}$, les instructions à ajouter pour obtenir le code linéaire tangent forment une application entre variables $\dot{v}_i$ de la forme $\{\dot{v}_i : i \in D_k\} \mapsto \dot{v}_{\mu_k}$.

1.3 Mode inverse de différentiation (code linéaire cotangent)

Le mode inverse de différentiation est moins intuitif et plus complexe à mettre en œuvre que le mode direct. Il trouve son utilité lorsque l'on veut calculer des dérivées par rapport à un grand nombre de variables. Ce mode est une méthode de génération automatique de codes adjoints, encore appelés codes linéaires cotangents, concept familier en commande optimale.

Il y a plusieurs manières d'introduire le mode inverse. Dans l'état de l'art [6], quatre approches ont été recensées : l'approche du graphe de calcul [16, 10, 11], la méthode des substitutions rétrogrades [17, 12], la méthode de dualité et l'approche de Speelpenning [17]. Chacune de ces approches apporte un éclairage différent sur ce mode de différentiation, toujours surprenant, mais c'est la dernière qui est la plus simple à exposer et à adapter à diverses situations. C'est aussi celle que nous présentons ci-après et que nous utiliserons à nouveau à la section 1.4.1.

L'idée est de voir le code (1.1) comme une composition de K applications. Pour cela, on interprète l'instruction k de (1.1) comme une transformation agissant sur l'ensemble des variables v_1, ..., v_N du code, qui laisse inchangées toutes ces variables sauf v_{μ_k}. On lui associe donc l'application $\Phi_k : \mathbb{R}^N \to \mathbb{R}^N$, définie par

$$(\Phi_k(v))_i = \begin{cases} v_i & \text{si } i \neq \mu_k \\ \varphi_k(v_{D_k}) & \text{si } i = \mu_k. \end{cases}$$

Alors, on peut voir le programme (1.1) comme une composition de K fonctions :

$$(\Phi_K \circ \cdots \circ \Phi_1)(v).$$

Afin de lever certaines ambiguïté, on note $v^0 = v$ et

$$v^k = (\Phi_k \circ \cdots \circ \Phi_1)(v)$$

la valeur des variables du code après exécution des k premières instructions.

Soit alors d dans $\mathbb{R}^m$, espace d'arrivée de f. Avec le mode inverse, on cherche à calculer le gradient de l'application $x \mapsto d^\top f(x)$. Il s'agit donc d'une dérivation cotangente. Si on note 0_p le vecteur nul de $\mathbb{R}^p$, on a

$$d^\top f(x) = \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}^\top (\Phi_K \circ \cdots \circ \Phi_2 \circ \Phi_1)(x, 0_{N-n}),$$

si bien que

$$\nabla(d^\top f)(x) = \begin{pmatrix} I_n & 0_{n\times(N-n)} \end{pmatrix} \Phi_1'(v)^\top \Phi_2'(v^1)^\top \cdots \Phi_K'(v^{K-1})^\top \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix},$$

où I_n est la matrice unité d'ordre n et $0_{p\times q}$ est la matrice nulle de dimension $p \times q$. Dans le produit de matrices du membre de droite, on a un vecteur à droite et une matrice à gauche. Le nombre d'opérations pour l'évaluer sera donc moindre si on effectue les multiplications matricielles de droite à gauche. Pour effectuer ces produits, on introduit des *variables duales* $\overline{v} \in \mathbb{R}^N$, que l'on initialise au vecteur à droite de l'expression ci-dessus

$$\overline{v} := \begin{pmatrix} 0_{N-m} \\ d \end{pmatrix}.$$

Ensuite, ces variables duales sont mises à jour par

$$\overline{v} := \Phi_k(v^{k-1})^\top \overline{v}, \quad \text{pour } k = K, \ldots, 1.$$

On voit que finalement, le gradient $\nabla(d^\top f)(x)$ se trouve dans les n premières composantes du $\overline{v}$ ainsi obtenu.

Pour écrire cette procédure de façon précise, il reste à observer que la jacobienne de $\Phi_k(v^{k-1})$ est la matrice $N \times N$

$$\Phi_k'(v^{k-1}) = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ \frac{\partial\varphi_k}{\partial v_1}(v^{k-1}) & \cdots & \cdots & \frac{\partial\varphi_k}{\partial v_{\mu_k}}(v^{k-1}) & \cdots & \cdots & \frac{\partial\varphi_k}{\partial v_N}(v^{k-1}) \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix},$$

où seule sa μ_k-ième ligne peut ne pas être formée de vecteurs de base de $\mathbb{R}^N$. Alors la mise à jour $\overline{v} := \Phi_k(v^{k-1})^\top \overline{v}$ s'écrit

$$\begin{cases} \overline{v}_i := \overline{v}_i & \text{si } i \notin D_k \cup \{\mu_k\} \\ \overline{v}_i := \overline{v}_i + \dfrac{\partial\varphi_k}{\partial v_i}(v^{k-1})\overline{v}_{\mu_k} & \text{si } i \in D_k \backslash \{\mu_k\} \\ \overline{v}_{\mu_k} := \dfrac{\partial\varphi_k}{\partial v_{\mu_k}}(v^{k-1})\overline{v}_{\mu_k}. & \end{cases} \tag{1.3}$$

Il faut d'abord mettre à jour les $\overline{v}_i$ pour $i \neq \mu_k$ avant $\overline{v}_{\mu_k}$, car il est clair que c'est l'ancienne valeur de $\overline{v}_{\mu_k}$ qu'il faut utiliser dans le calcul des $\overline{v}_i$ ($i \in D_k \backslash \{\mu_k\}$).

On peut donc résumer le code linéaire cotangent calculant $\nabla(d^\top f)(x)$ par le programme suivant

```
pour $i = 1, \ldots, N - m$
  $\overline{v}_i := 0$ ;
pour $i = N - m + 1, \ldots, N$
  $\overline{v}_i := d_i$ ;
pour $k = K, \ldots, 1$
  pour $i \in D_k \backslash \{\mu_k\}$
    $\overline{v}_i := \overline{v}_i + \frac{\partial \varphi_k}{\partial v_i}(v^{k-1})\overline{v}_{\mu_k}$ ;
  $\overline{v}_{\mu_k} := \frac{\partial \varphi_k}{\partial v_{\mu_k}}(v^{k-1})\overline{v}_{\mu_k}$ ;
$\nabla(d^\top f)(x) := (\overline{v}_1, \ldots, \overline{v}_n)$ .
```

(1.4)

On voit que cette procédure s'adapte bien à la situation où on veut dériver une seule variable ($m = 1$) par rapport à beaucoup de variables d'entrée (n grand). Dans ce cas, il suffit d'initialiser toutes les variables duales à 0 sauf $\overline{v}_N$ que l'on initialise à 1.

Illustrons cela en appliquant le mode inverse sur l'exemple de la page 8, lorsque l'on veut dériver la variable `v5` par rapport aux variables `v1` et `v2`. On commence par exécuter le code original en sauvegardant certaines quantités. Le choix de l'information à sauvegarder est délicat. Pour coller à la présentation faite ci-dessus, nous avons choisi de sauvegarder les dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$ dans une pile `p` munie du pointeur `ip`, sans essayer d'optimiser le code de manière à garder une certaine lisibilité.

```
ip = 1
p(ip) = 2*v2
v3 = v1 + v2**2
ip = ip + 1
p(ip) = 2*v1*sin(v3)
ip = ip + 1
p(ip) = v1**2 * cos(v3)
v4 = v1**2 * sin(v3)
ip = ip + 1
p(ip) = 1/v3
ip = ip + 1
p(ip) = -v4/v3**2
v4 = v4/v3
ip = ip + 1
p(ip) = exp(v4)
v5 = exp(v4)
```

Ensuite vient l'exécution du code linéaire cotangent. On a utilisé le suffixe `ad` (convention de nom utilisée dans ODYSSÉE) pour désigner les variables duales $\overline{v}_i$ associées aux variables v_i.

```
v1ad = 0
v2ad = 0
v3ad = 0
```

```
v4ad = 0
v5ad = 1
v4ad = v4ad + p(ip)*v5ad
v5ad = 0
ip = ip - 1
v3ad = v3ad + p(ip)*v4ad
ip = ip - 1
v4ad = p(ip)*v4ad
ip = ip - 1
v3ad = v3ad + p(ip)*v4ad
ip = ip - 1
v1ad = v1ad + p(ip)*v4ad
v4ad = 0
ip = ip - 1
v2ad = v2ad + p(ip)*v3ad
v1ad = v1ad + v3ad
v3ad = 0
```

En fin d'exécution, le gradient de `v5` par rapport à `v1` et `v2` se trouve dans (`v1ad`, `v2ad`).

On peut à présent comprendre la difficulté de mise en œuvre du mode inverse. Il faut d'abord exécuter le code direct (1.1) afin de mémoriser les "quantités" permettant de reconstituer les dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$ au moment où celles-ci sont utilisées dans le code linéaire cotangent (1.4). Remarquons que cette utilisation se fait en ordre inverse (de $k = K, \ldots, 1$) de l'ordre de mémorisation. Il y a donc un problème de gestion de la mémoire qui ne se pose pas avec le mode direct. Une stratégie extrême consiste à mémoriser, à chaque itération k, toute l'information nécessaire à l'évaluation des dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$, $i \in D_k$. Ceci conduit en général à un besoin en place-mémoire à peu près proportionnel au nombre d'instructions où les variables actives interviennent de façon non linéaire, soit en première approximation proportionnel au temps d'exécution du code original (en fait cela dépend beaucoup de ce que l'on mémorise). Cette stratégie ne peut donc fonctionner que pour les petits problèmes. Pour les grands problèmes, il est préférable de mémoriser une partie des variables du code à certains instants de l'exécution de (1.1) (voir par exemple [8] et [9]). Dans un problème d'évolution, par exemple, on pourra ne mémoriser que les variables d'état du problème à chaque pas de temps. Ensuite les dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$ sont recalculées à partir des informations mémorisées. On parvient ainsi à obtenir des codes moins gourmands en place-mémoire, au prix d'une augmentation du temps de calcul.

En ce qui concerne, le temps d'exécution du couple code original et code linéaire cotangent, par rapport au temps d'exécution du premier, on peut montrer que si toutes les fonctions intermédiaires φ_k sont prises dans $\mathcal{F}_F$ (famille des fonctions Fortran) et si on fait des hypothèses raisonnables sur le temps respectif de l'évaluation des fonctions de $\mathcal{F}_F$, on a la majoration

$$\frac{T(f(x), \nabla(d^\top f(x)))}{T(f(x))} \leq 5 \tag{1.5}$$

et ceci quel que soit le nombre n de variables par rapport auxquelles on dérive (voir par exemple [13, 7, 6]). Cette inégalité ne tient pas compte du surcoût lié aux accès-mémoire qui, comme nous l'avons vu, peuvent être importants en mode inverse. En pratique, cette estimation n'est donc pas toujours vérifiée. Pour de grands codes (nous pensons au code d'assimilation variationnelle des données utilisé en météorologie [18]), on parvient toutefois à avoir une borne dans (1.5) voisine de 3. Celle-ci est obtenue par une gestion astucieuse, mais pour l'instant encore en partie manuelle, de la mémoire.

Ce qu'il faut retenir :

1. Le mode inverse de différentiation est le bon mode pour calculer les dérivées partielles d'une seule variable (ou variable de sortie) par rapport à un grand nombre de variables (ou variables d'entrée).
2. C'est donc le bon mode pour calculer le gradient d'une application à valeurs scalaires $f : \mathbb{R}^n \to \mathbb{R}$ ou le gradient de $d^\top f$ si f est à valeurs vectorielles.
3. À chaque variable v_i du code, on associe une variable duale $\overline{v}_i$ qui contient la variation de la variable de sortie (ou de $d^\top f(x)$, pour un vecteur d donné, en cas de sortie vectorielle) par rapport à une perturbation de la variable v_i. Sa valeur est nulle si v_i n'a pas d'influence sur la variable de sortie.
4. Le code linéaire cotangent peut s'obtenir en "dualisant" le code original ligne par ligne. Cette dualisation doit se faire en ordre inverse de l'ordre d'exécution dans le code original. Si dans ce dernier une instruction s'interprète comme une application $\{v_i : i \in D_k\} \mapsto v_{\mu_k}$, les instructions correspondantes dans le code linéaire cotangent forment une application entre variables duales de la forme $(\overline{v}_i : i \in D_k \cup \{\mu_k\}) \mapsto \{\overline{v}_i : i \in D_k \cup \{\mu_k\}\}$.
5. La difficulté principale dans l'écriture de codes linéaires cotangents est la gestion-mémoire permettant de transmettre du code direct au code linéaire cotangent, l'information nécessaire à la reconstitution des dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$.

1.4 Différentiation automatique de quelques cas particuliers

Dans cette section, nous détaillons la différentiation de quelques groupes d'instruction particuliers qui se sont présentés lors du traitement du code THYC-1D par ODYSSÉE.

1.4.1 Cas d'une transformation à valeurs vectorielles

Les formules données en sections 1.2 et 1.3 traitent du cas où chaque instruction du code original modifie une seule variable v_μ, $\mu \in \{1, \ldots, N\}$. Il est utile d'étendre ces formules au cas où les fonctions intermédiaires φ sont à valeurs vectorielles. En effet, une telle fonction peut représenter une sous-routine ou une partie de programme qui, dans certains cas, ne peut pas être différentiée ligne par ligne mais doit être considérée comme un bloc. En particulier, cela se présente si le code-source de certaines parties du programme n'est pas disponible et que l'unique possibilité dont on dispose est d'approcher sa dérivée tangente ou cotangente par différences finies (ce cas est discuté en section 2.2.1).

Aux sections 1.2 et 1.3, nous avons vu qu'à chaque instruction du code original correspondent une ou plusieurs instructions dans les codes linéaire tangent ou linéaire cotangent. Nous montrons ici comment les écrire lorsque l'instruction du code original est à valeurs vectorielles et que l'on connaît sa jacobienne. On suppose donc que l'instruction est de la forme

$$\boxed{v_M := \varphi(v_D),} \tag{1.6}$$

où M et D sont des parties de $\{1,\ldots,N\}$, pouvant être disjointes ou non. On indice par $i \in M$ les composantes de φ.

Code linéaire tangent

Le code linéaire tangent est simple à écrire. Il suffit de donner les formules de transformation des dérivées directionnelles $\dot{v}$, ce que l'on obtient en différentiant les deux membres de (1.6) :

$$\boxed{\dot{v}_M := \varphi'(v_D)\dot{v}_D.}$$

Code linéaire cotangent

Pour obtenir le code linéaire cotangent, on reprend l'idée de la section 1.3 en associant à φ une application $\Phi : \mathbb{R}^N \to \mathbb{R}^N$, qui laisse inchangées toutes les variables du code, sauf celles d'indices dans M. Pour $i \in \{1,\ldots,N\}$, la composante i de $\Phi(v)$ est définie par

$$(\Phi(v))_i = \begin{cases} v_i & \text{si } i \notin M \\ \varphi_i(v_D) & \text{si } i \in M. \end{cases}$$

L'élément (i,j) de la jacobienne de Φ s'écrit

$$(\Phi'(v))_{ij} = \begin{cases} \delta_{ij} & \text{si } i \notin M \\ \frac{\partial \varphi_i}{\partial v_j}(v_D) & \text{si } i \in M. \end{cases}$$

On a utilisé le symbole de Kronecker : δ_{ij} vaut 1 si $i = j$ et 0 si $i \neq j$. Alors le vecteur $\overline{v}$ des variables duales est modifié en lui appliquant la transposée de cette jacobienne (voir section 1.3) : $\overline{v} := \Phi'(v)^\top \overline{v}$. On trouve pour la composante j du nouveau $\overline{v}$:

$$\overline{v}_j := \begin{cases} \overline{v}_j & \text{si } j \notin M \cup D \\ \overline{v}_j + \sum_{i \in M} \frac{\partial \varphi_i(v_D)}{\partial v_j} \overline{v}_i & \text{si } j \in D \backslash M \\ \sum_{i \in M} \frac{\partial \varphi_i(v_D)}{\partial v_j} \overline{v}_i & \text{si } j \in D \cap M \\ 0 & \text{si } j \in M \backslash D. \end{cases}$$

Le signe ":=" rappelle qu'il s'agit d'une affectation, pas d'une égalité au sens mathématique. En particulier, il faut utiliser l'ancien $\overline{v}$ à droite, pas celui qui est calculé par cette formule. On peut donner une forme compacte à ces formules, qui n'a pas cette ambiguïté.

Dans ce but, on introduit la notation suivante. Si A et B sont des parties de $\{1,\ldots,N\}$, on note $v_{A\backslash B}$, le vecteur de $\mathbb{R}^{|A|}$ défini par

$$\left(v_{A\backslash B}\right)_i = \begin{cases} v_i & \text{si } i \in A\backslash B \\ 0 & \text{si } i \in A \cap B. \end{cases}$$

Alors, la mise à jour de $\overline{v}$ se fera en appliquant *dans l'ordre* les formules suivantes :

$$\boxed{\begin{array}{l} \overline{v}_D := \overline{v}_{D\backslash M} + \varphi'(v_D)^\top \overline{v}_M \\ \overline{v}_{M\backslash D} := 0. \end{array}} \tag{1.7}$$

Si $M = \{\mu\}$, on retrouve les formules (1.3).

1.4.2 Cas d'un solveur linéaire

Supposons qu'une partie d'un programme consiste à résoudre le système linéaire en $y \in \mathbb{R}^n$ suivant :

$$\boxed{Ay = b,} \tag{1.8}$$

où la matrice carrée inversible A et le vecteur $b \in \mathbb{R}^n$ peuvent dépendre des variables indépendantes x. Les codes linéaire tangent et linéaire cotangent de (1.8) peuvent s'obtenir en appliquant les formules de la section 1.4.1.

Code linéaire tangent

Le code original définit une application $(A, b) \mapsto y$, qui à la donnée de la matrice A et du vecteur b calcule la solution y du système linéaire (1.8). Le code linéaire tangent définira donc une application $(\dot{A}, \dot{b}) \mapsto \dot{y}$ entre les dérivées directionnelles (suivant une certaine direction $d \in \mathbb{R}^n$) $\dot{A}$, $\dot{b}$ et $\dot{y}$ de A, b et y respectivement. Il est facile à écrire. Il suffit de dériver l'équation (1.8). On obtient

$$\boxed{A\dot{y} = \dot{b} - \dot{A}y.} \tag{1.9}$$

Dans ce système, $\dot{b}$ et $\dot{A}$ sont connus (ils ont déjà été calculés précédemment). Il s'agit donc de résoudre un système linéaire en $\dot{y}$ avec la même matrice que dans le code original.

Code linéaire cotangent

Venons-en au code linéaire cotangent de (1.8). Ce système peut être considéré comme une fonction intermédiaire à valeurs vectorielles $\varphi : (A, b) \mapsto y = A^{-1}b$. Donc son code linéaire cotangent réalisera une application $(\overline{A}, \overline{b}, \overline{y}) \mapsto (\overline{A}, \overline{b})$, où $\overline{A} \in \mathbb{R}^{n \times n}$, $\overline{b} \in \mathbb{R}^n$ et $\overline{y} \in \mathbb{R}^n$ sont les variables duales associées aux variables A, b et y. Nous allons montrer que le code linéaire cotangent de (1.8) est un code réalisant les opérations suivantes :

$$\boxed{\begin{array}{l} A^\top z = \overline{y} \\ \overline{b} := \overline{b} + z \\ \overline{A} := \overline{A} - zy^\top \\ \overline{y} := 0. \end{array}} \tag{1.10}$$

On voit que l'on a besoin de mémoriser la matrice A et la solution y de (1.8). On voit aussi qu'il suffit de résoudre un système linéaire avec la matrice transposée de A et que l'on apporte à $\overline{A}$ une correction de rang un.

Pour montrer cela, on applique les formules (1.7) obtenues dans le cas général (1.6). Ici, $\varphi(A, b) = A^{-1}b$. À la dépendance de φ en b correspond l'instruction adjointe suivante

$$\overline{b} := \overline{b} + (A^{-1})^\top \overline{y}.$$

Ceci donne les deux premières formules de (1.10). Regardons à présent quelles sont les instructions adjointes correspondant à la dépendance de φ en A. On note e_i le i-ième vecteur de la base canonique de $\mathbb{R}^n$ et $I_{ij} = e_i e_j^\top$ la matrice carrée ayant un 1 en position (i, j) et des 0 ailleurs. Alors, suivant les règles (1.7), tout élément (i, j) de $\overline{A}$ est mis à jour par

$$\begin{aligned} \overline{A}_{ij} &= \overline{A}_{ij} + \sum_{k=1}^{n} \frac{\partial (A^{-1}b)_k}{\partial A_{ij}} \overline{y}_k \\ &= \overline{A}_{ij} - (A^{-1} I_{ij} A^{-1} b)^\top \overline{y} \\ &= \overline{A}_{ij} - (A^{-1} e_i y_j)^\top \overline{y} \\ &= \overline{A}_{ij} - y_j e_i^\top A^{-\top} \overline{y} \\ &= \overline{A}_{ij} - z_i y_j . \end{aligned} \tag{1.11}$$

Ceci donne la troisième affectation de (1.10). La dernière affectation de (1.10) vient de la dernière de (1.7).

Cas des matrices creuses

Supposons à présent que A soit creuse, c'est-à-dire qu'elle ait beaucoup d'éléments que l'on sait être nul et que l'on impose à zéro. Ce cas ne pose pas de difficultés pour le code linéaire tangent, car (1.9) consiste à résoudre un système linéaire avec la même matrice creuse A. Pour le code linéaire cotangent, si les formules (1.10) sont appliquées telles quelles, $\overline{A}$ devient dense, ce qui n'est pas souhaitable. Ce n'est pas non plus très naturel, car si on sait que A_{ij} est nul, il n'y a probablement pas de variable dans le programme pour mémoriser cet élément. Il ne devrait donc pas y avoir de variable duale $\overline{A}_{ij}$.

En fait, dans le cas où la matrice A est creuse, on peut prendre les formules de mise à jour des variables duales suivantes :

$$\left\{ \begin{array}{l} A^\top z = \overline{y} \\ \overline{b} := \overline{b} + z \\ \overline{A}_{ij} := \overline{A}_{ij} - z_i y_j, \quad \forall (i,j) \in \mathcal{N} \\ \overline{y} := 0, \end{array} \right. \tag{1.12}$$

où $\mathcal{N}$ est l'ensemble des indices (i, j) des éléments de A pouvant être non nuls. Ces formules diffèrent de (1.10) uniquement par le fait que seuls les éléments pertinents de $\overline{A}$ sont mis à jour, ce qui est assez naturel. Pour les obtenir, on observera que le calcul (1.11) est toujours valable.

Chapitre 2

Le code THYC-1D

2.1 Brève description

Le code THYC-1D (thermohydraulique des cœurs de réacteurs) est une maquette à une dimension en espace du code THYC. Ce dernier permet de simuler l'évolution d'un écoulement diphasique (eau et vapeur, par exemple) en dimension trois, en configuration de faisceaux de tubes suivants : crayons chauffants, échangeurs, ou condenseurs.

Le modèle physique de THYC-1D est formé des équations de conservation des masses du mélange et de la vapeur, de celles des quantités de mouvement du mélange et relative entre phases, de l'énergie du mélange et de lois de fermeture. Les inconnues du modèle sont la vitesse massique, la pression, l'entropie du mélange, le titre statique et la vitesse relative.

La discrétisation en espace se fait sur 25 points alignés. La discrétisation en temps utilise un schéma à pas variables. À chaque pas de temps, il s'agit de résoudre un système linéaire de 5 équations vectorielles, ce qui se fait par une méthode du type Gauss-Seidel par bloc avec une seule correction. La résolution de chaque bloc se fait par des itérations de Jacobi ou de Gauss-Seidel. Dans nos tests, les équations sont intégrées du temps $t = 0$ au temps $t = 5$.

D'un point de vue informatique, le code THYC-1D est constitué d'un programme principal `princp` et de 25 sous-routines. Parmi celles-ci on trouve :

- 2 routines d'entrée-sortie (`ouvrir` et `impression`),
- 2 solveurs linéaires (`jacobi` et `tdma`),
- 9 autres routines de calcul (`table`, `solvp`, `solvur1`, `solvtit`, `solvs1`, `solvq1`, `solvtit1`, `actua`, `xnorme`) et
- 12 fonctions tabulées dont le code-source n'est pas disponible (`tbalps`, `tbhhps`, `tbhhsa`, `tbcpps`, `tba2ps`, `tbsssa`, `tbetps`, `tbssph`, `tbrops`, `tbttps`, `tbttsa`, `tbbbps`).

2.2 Particularités

Le code THYC-1D présente quelques particularités qui ont demandé un traitement adapté, nécessitant une intervention manuelle et ne permettant donc pas une génération complètement automatique du code calculant les dérivées. Les trouble-fêtes sont :

- des fonctions tabulées, dont le code-source n'est pas disponible,

– des solveurs linéaires utilisant un processus itératif à convergence non finie.

Dans cette section, nous allons discuter les raisons qui nous ont orientés vers cette voie "semi-automatique". Dans le premier cas, il n'y avait apparemment pas d'autre possibilité, dans le second c'est l'efficacité du code généré qui a primé. Nous verrons que les interventions manuelles sont réduites au minimum. On verra à la section 3.2.1 qu'ODYSSÉE est conçu pour ce traitement semi-automatique.

2.2.1 Les fonctions en boîte noire

Lorsque le code-source n'est pas disponible, comme pour les fonctions tabulées de THYC-1D, il n'y a apparemment pas d'autres possibilités que de fournir au différentiateur les routines correspondantes qu'il doit utiliser en mode direct ou inverse. Dans ce cas, ce sont les formules obtenues en section 1.4.1 qui sont utiles.

Écrivons la routine à différentier sous la forme d'une affectation vectorielle

$$v_M := \varphi(v_D).$$

En mode direct, il s'agit d'écrire un code qui calcule à la fois $v_M := \varphi(v_D)$ et $\dot{v}_M := \varphi'(v_D)\dot{v}_D$. Pour le calcul de $\varphi'(v_D)\dot{v}_D$, le plus simple et le moins coûteux est d'en faire une évaluation par différences finies (voir annexe A.3 ou A.4). Notons que l'on aura plus de précision en calculant les $\varphi_i'(v_D)\dot{v}_D$ $(i \in M)$ séparément et en adaptant le pas à chaque composante (annexe A.4), mais le coût de ce calcul sera d'autant plus grand que $|M|$ est grand.

En mode inverse, il s'agit d'appliquer les formules (1.7) et donc d'évaluer le produit $\varphi'(v_D)^\top \overline{v}_M$. Il n'y a pas à notre connaissance de méthode permettant d'approcher ce produit en ne calculant qu'un seul quotient différentiel. On peut cependant en avoir une estimation composante par composante. Si e_j est le j-ième vecteur de la base canonique de $\mathbb{R}^{|D|}$, la j-ième composante du produit s'écrit

$$e_j^\top \varphi'(v_D)^\top \overline{v}_M = \overline{v}_M^\top \varphi'(v_D) e_j.$$

Il suffit donc d'évaluer les $\varphi'(v_D)e_j$ $(j \in D)$ par différences finies. On voit qu'il s'agit en fait de calculer la jacobienne $\varphi'(v_D)$, colonne par colonne.

2.2.2 Les fonctions de bibliothèque

Discutons à présent du traitement des sous-routines qui sont ou pourraient être issues de bibliothèques de programmes comme celles spécialisées en algèbre linéaire, en optimisation, en automatique, *etc.* Dans le cas de THYC-1D, il s'agit uniquement de solveurs linéaires écrits, en fait, par les auteurs du code. Le premier solveur (sous-routine `jacobi`) utilise la méthode de Jacobi pour résoudre un système linéaire tridiagonal. Le second solveur (sous-routine `tdma`) utilise la méthode de Gauss-Seidel également pour résoudre un système linéaire tridiagonal.

Dans le cas où le code-source est disponible, on peut envisager au moins deux possibilités de traitement de ces sous-routines. Soit ODYSSÉE les considère comme des sous-routines normales et les traite de manière habituelle, c'est-à-dire ligne par ligne, en générant un code dérivé (linéaire tangent ou linéaire cotangent). Soit l'utilisateur fournit lui-même le code dérivé et ODYSSÉE ne se charge que de la dérivation des appels. Cette alternative n'est pas propre aux sous-routines de bibliothèque. Comme nous allons le voir, pour des raisons d'efficacité il est parfois préférable d'utiliser la deuxième possibilité pour les routines effectuant des transformations de variables représentables par des opérateurs mathématiques simples.

Le cas de solveurs linéaires utilisant un processus itératif à convergence (comme la méthode de Jacobi par exemple) est instructif. Dans ce cas, le nombre d'itérations à effectuer pour obtenir la solution n'est pas connu et dépend de la précision demandée. Le traitement en aveugle par ODYSSÉE d'un tel code est délicat, car dans ce cas, le nombre d'itérations dans le code généré sera identique à celui du code original. Or rien ne dit que l'on aura convergence des dérivées pour le même nombre d'itérations (voir [5]). Il faudrait donc contrôler ce nombre dans le code dérivé. Presque inévitablement, ceci demande une intervention de l'utilisateur, au moyen de directives par exemple, car on voit mal comment on pourrait distinguer un processus itératif demandant convergence d'une boucle anodine. Mais s'il faut demander une intervention de l'utilisateur, autant que ce soit au niveau plus structuré de la sous-routine car cette intervention peut alors se faire plus proprement. On est dans un cas typique où il vaut mieux utiliser la structure du code à dériver plutôt que de dériver ligne par ligne.

Dans le cas de solveurs linéaires, la tâche de l'utilisateur est aisée. Il suffit d'écrire les codes réalisant (1.9) pour le mode direct et (1.10) ou (1.12) pour le mode inverse. Ces codes peuvent utiliser les solveurs linéaires eux-mêmes pour résoudre les systèmes linéaires que l'on trouve dans ces formules.

Ce traitement n'est pas propre aux solveurs linéaires. Par exemple, Talagrand [18] mentionne le cas d'une transformation de Fourier, qui est une transformation orthogonale. Alors, le code adjoint, qui utilise la transposée de cette transformation, donc son inverse, pourra s'écrire en utilisant la transformation de Fourier inverse. Dans ce cas aussi il est préférable de fournir au différentiateur le code adjoint plutôt que de le laisser générer un code qui a peu de chance d'être plus efficace.

Nous pensons que ce traitement devrait être fait pour beaucoup de fonctions de bibliothèque.

Chapitre 3

Le différentiateur ODYSSÉE

ODYSSÉE [14] est un logiciel de différentiation automatique mettant en œuvre à la fois la dérivation en mode direct et en mode inverse. Il est développé à l'INRIA (Sophia Antipolis) dans le projet SAFIR en Camllight sous une forme modulaire.

Le système ODYSSÉE prend pour entrée un code écrit en Fortran-77 qui calcule une fonction différentiable par morceaux et fournit en sortie un nouveau code Fortran-77 qui calcule une dérivée tangente (une dérivée directionnelle) ou cotangente (un gradient) de cette fonction. Le système fonctionne par transformation du code-source. Le code généré a la même structure que le code initial.

3.1 De la version 1.4 à la version 1.5

Les premiers tests effectués dans le cadre de ce travail ont été faits avec la version 1.4 d'ODYSSÉE, ceux présentés dans ce rapport sont réalisés avec la version 1.5.

La version 1.4 d'ODYSSÉE permettait de dériver des unités de compilation (`subroutine`, `function`, ...) indépendantes, c'est-à-dire sans appel à d'autres procédures. Une telle version ne pouvait être utilisée sur THYC-1D qu'en remplaçant les appels aux unités par des affectations possédant les mêmes variables d'entrée/sortie (arguments et `commons`). Ces modifications du code initial sur chaque appel auraient donc été très fastidieuses.

Pour éviter cela, nous avons dans un premier temps ajouté à la version 1.4 d'ODYSSÉE une base d'informations (notée `BI`) dans laquelle peuvent être mémorisées les entrées/sorties[1] des routines appelées, les entrées actives ainsi que les dépendances entre les sorties et les entrées. Connaissant ces informations, le système peut en déduire les variables actives dans le code d'une unité. Cette `BI` peut être remplie par chargement d'un fichier de déclarations fourni par l'utilisateur. Nous avons donc défini la `BI` pour THYC-1D en en étudiant le code. De plus, il nous a fallu définir les entrées actives de chaque routine de THYC-1D, à partir des variables actives de la routine de tête `princp`. C'est ainsi que le code de THYC-1D a pu être dérivé une première fois, routine après routine. Mais cette solution n'était pas très satisfaisante car hormis le fait que l'analyse du code fut très

---

[1] Nous abrégeons "variable d'entrée" d'une unité par "entrée", "variable de sortie" par "sortie" et traduisons `input`/`output` par "lecture/écriture".

fastidieuse, il n'y avait de cette manière aucune cohérence assurée entre les routines générées par le système.

Pour assurer cette cohérence, deux fonctionnalités ont été ajoutées à ODYSSÉE :

1. l'analyse automatique des dépendances,
2. la propagation automatique des variables actives.

C'est l'apport de ces deux nouvelles fonctionnalités qui a conduit à la version 1.5 d'ODYSSÉE, version que nous allons décrire et que nous avons utilisée dans les tests présentés en section 5. L'utilisation du logiciel est décrite dans le manuel (voir annexe F).

3.2 Fonctionnement

Les seules routines dérivées par ODYSSÉE sont celles définies par l'instruction `subroutine`. Ni les `functions`, ni leurs appels ne sont dérivés. Il faut donc modifier le code à dériver pour remplacer les appels et les définitions de fonctions par des appels et des définitions de `subroutines`. Les unités de compilation de type `program` ne sont pas non plus dérivées par ODYSSÉE, de même que les unités de type `blockdata` pour lesquelles cette opération n'a pas de signification.

D'autre part, dans la version 1.5 d'ODYSSÉE, nous avons choisi d'appliquer les règles de transformation uniquement aux instructions de calcul et pas aux instructions de lecture/écriture (celles-ci seront prises en compte dans les versions futures). Par conséquent, il faut séparer dans le programme principal `main` la partie qui lit/écrit de celle qui fait des calculs et transformer cette dernière en `subroutine`.

ODYSSÉE permet la dérivation d'un ensemble d'unités de compilation dont le graphe d'appel forme un arbre. La racine de cet arbre est appelée *routine de tête*. L'ensemble de ces routines est dérivé par rapport à des variables d'entrée de la routine de tête, désignées par l'utilisateur.

Si l'ensemble des unités à traiter ne possède pas cette structure d'arbre, deux cas de figure sont possibles.

1. *L'utilisateur ne veut pas dériver l'ensemble des unités comme un tout.* Il faut alors dériver chaque sous-arbre séparément en spécifiant à ODYSSÉE les entrées actives des routines de tête de chaque sous-arbre. De plus, l'utilisateur doit se charger lui-même de l'initialisation correcte des variables associées (dérivées directionnelles $\dot{v}_i$ ou variables duales $\overline{v}_i$) aux variables d'entrée actives de chaque routine de tête.

2. *L'utilisateur veut dériver l'ensemble des unités comme un tout.* Il faut alors regrouper ces unités en écrivant une routine de tête qui les appelle et dériver l'ensemble en spécifiant à ODYSSÉE les entrées actives de cette routine de tête. L'initialisation des variables associées aux variables d'entrée actives de chaque routine autre que la routine de tête est alors prise en charge par ODYSSÉE.

3.2.1 L'analyse de code

L'analyse du code consiste dans un premier temps à calculer un graphe de dépendances entre les symboles, adapté à la dérivation. Ses caractéristiques sont les suivantes.

- *Seules les dépendances de valeur sont prises en compte.*

Deux types de dépendances entre symboles peuvent se présenter, les dépendances de valeur et les dépendances de contrôle. Par exemple, dans l'instruction

```
if (x.lt.0.) then
  z = y
end if
```

`z` dépend en contrôle de `x` et en valeur de `y`. Les dépendances de contrôle ne sont pas prises en compte par ODYSSÉE car, en reprenant l'exemple ci-dessus, on voit que le symbole `z` doit devenir actif si `y` est actif, mais ne doit pas le devenir si `x` est actif. Les dépendances de contrôle sont donc inutiles pour la propagation des variables actives.

- *Seule la clôture du graphe par rapport aux entrées de la routine est conservée.*

  Nous avons choisi de ne conserver que la clôture du graphe par rapport aux entrées d'une unité. On ne conserve donc pas les dépendances entre variables intermédiaires. En effet, le mode de dérivation choisi par ODYSSÉE consiste en la dérivation d'une instruction par rapport aux entrées de la routine qui la contient, de plus cette information est intrinsèque à la routine : elle ne dépend pas de la façon dont la routine est appelée, ni de la façon dont elle est dérivée. De plus ce graphe est plus petit à stocker que le graphe complet. De cette manière, l'analyse du code peut être utilisée dans la même session ODYSSÉE pour dériver un code donné par rapport à plusieurs jeux d'entrées actives, et pourrait être écrite dans un fichier et relue lors d'une dérivation ultérieure.

Les informations déduites de cette analyse sont pour chaque unité lue : la définition des entrées/sorties et les dépendances des sorties par rapport aux entrées. Celles-ci sont enregistrées automatiquement dans la base d'informations (`BI`).

Cette `BI` est complètement remplie par le système concernant les unités lues (dont on possède le code Fortran), mais peut aussi être remplie par l'utilisateur en particulier concernant les unités uniquement appelées. Lorsque l'utilisateur ne déclare aucune information concernant une unité appelée, ODYSSÉE considère que tous les arguments de la routine sont à la fois entrée/sortie, et que chaque sortie dépend de toutes les entrées.

Une fois cette base d'informations remplie, le système propage automatiquement les variables actives en partant de la routine de tête (la première routine appelée pour le calcul de la fonction) vers les feuilles. Ceci lui permet de déterminer pour chaque routine quelles sont ses entrées actives par rapport auxquelles elle sera dérivée.

3.2.2 Le prétraitement des unités

Les unités lues ont parfois besoin de subir certaines transformations avec de pouvoir être dérivées par ODYSSÉE :

- Le processus "`expand`" permet de macro-expanser les "`function statement`". Ceci permet de ne plus s'en préoccuper dans les étapes suivantes.
- Le processus "`rewrite`" transforme les appels de functions intrinsèques `max`, `abs`, ..., non différentiables en `if` ... `then` ... `else` équivalents.

- Le processus "`split`" découpe les expressions en sous-expressions binaires (voir description dans [4]) soit totalement ce qui en théorie améliore la complexité de l'exécution, soit de façon minimale.

Le prétraitement d'une unité est effectué par ODYSSÉE juste avant la dérivation de l'unité.

3.2.3 La génération du code dérivé

La génération des dérivées en modes direct et inverse par ODYSSÉE suit les règles suivantes :

- *À chaque unité du code initial est associée une unique unité dérivée, qui doit donc être maximale par rapport aux entrées actives.*

  Dans ODYSSÉE nous avons choisi de calculer une dérivée maximale, c'est-à-dire que l'on calcule la dérivée d'une unité en prenant pour entrées actives la réunion de toutes les entrées étant une fois actives lors d'un appel à cette unité. Lors de la dérivation d'un appel, le système vérifie que le masque de la dérivée correspond à l'appel et génère l'appel maximal. Si certaines entrées de la routine sont supposées actives (donc dans le masque) alors qu'elles ne le sont pas effectivement, un message est mis en commentaire indiquant les dérivées d'entrée à mettre à zéro avant l'appel.
- *Chaque unité est dérivée par rapport à ses propres entrées.*
- *À chaque symbole actif d'une unité est associé un symbole dérivé possédant le même statut.*

  Le symbole dérivé désigne la dérivée directionnelle en mode direct ou la variable duale en mode inverse. Son statut (symbole global, argument ou local) est le même que celui du symbole auquel il est associé.
- *La même transformation est effectuée sur chaque unité.*

  La même transformation est effectuée sur chaque unité, la routine de tête est donc traitée comme les autres. Comme ODYSSÉE ne dérive pas l'unité appelant la routine de tête, il faut, en mode direct, initialiser soi-même les variables dérivées (dérivées directionnelles), globales ou arguments, associées aux variables actives de la routine de tête (voir section 3.3.1). De même en mode inverse, il faut initialiser soi-même les variables duales, globales ou arguments, de la routine adjointe de la routine de tête (voir section 3.3.2).

3.3 Fonctionnalités

3.3.1 Le linéaire tangent (mode direct)

Le code linéaire tangent d'une unité est l'unité qui calcule la fonction représentée par le code initial ainsi que son application linéaire tangente. Avec les notations de la section 1.2, si les variables d'entrée actives de l'unité initiale sont notées v_D et les variables de sortie actives sont notées v_M et si cette unité calcule $v_M := \varphi(v_D)$, le code linéaire tangent calculera $v_M := \varphi(v_D)$ et calculera $\dot{v}_M$ en v_D dans la direction $\dot{v}_D$, par $\dot{v}_M := \varphi'(v_D) \cdot \dot{v}_D$. La variable $\dot{v}_M$ ainsi calculée contient la valeur de la dérivée (directionnelle) de v_M au point v_D dans la direction $\dot{v}_D$.

Prenons comme exemple une routine avec des arguments seulement, afin de faire apparaître clairement la transformation. Le même travail est effectué lorsque le programme initial comporte des variables d'entrée globales. Soit l'unité

```
sample (a,b,c,d,e)
```

dont les arguments `a` et `b` sont des entrées actives, l'argument `c` est inactif et les arguments `d` et `e` sont des sorties actives. Supposons que `d` dépende de `a` et `b` et que `e` dépende de `b`. Alors, l'en-tête de la routine générée par ODYSSÉE en mode direct est :

```
sampletl (a,b,c,d,e,atl,btl,dtl,etl)
```

On observe qu'ODYSSÉE utilise le suffixe `tl` en mode direct à la fois pour désigner la routine dérivée `sampletl` et les variables contenant les variations `atl`, `btl`, `dtl`, `etl` (ou dérivées directionnelles) données aux variables actives `a`, `b`, `d` et `e`. S'il s'agit de la procédure de tête, les entrées `atl` et `btl` de cette routine doivent être initialisées à la direction suivant laquelle on veut dériver, sinon leurs valeurs se propagent d'une routine à l'autre.

En fin de calcul, les sorties `dtl` et `etl` vaudront :

$$
\begin{aligned}
\mathtt{dtl} &= \frac{\partial \mathtt{d}}{\partial \mathtt{a}}\,\mathtt{atl} + \frac{\partial \mathtt{d}}{\partial \mathtt{b}}\,\mathtt{btl}\\
\mathtt{etl} &= \frac{\partial \mathtt{e}}{\partial \mathtt{b}}\,\mathtt{btl}.
\end{aligned}
$$

Dans le corps d'une unité générée, chaque instruction du code initial est précédée de l'instruction dérivée qui lui correspond (voir (1.2)). La structure de contrôle est conservée.

3.3.2 Le linéaire cotangent (mode inverse)

On rappelle que le code linéaire cotangent d'une unité est l'unité qui calcule son application linéaire cotangente (voir section 1.3). Si on désigne encore par v_D les variables d'entrée actives et par v_M les variables de sorties actives, le code linéaire cotangent calcule ce qui est donné par les formules (1.7), c'est-à-dire qu'il met à jour les variables duales $\overline{v}_D$ associées aux entrées initiales v_D (argument ou globale) à partir des variables d'entrées initiales v_D et des variables duales $\overline{v}_D$ et $\overline{v}_M$.

Reprenons l'exemple précédent de la routine `sample`. L'en-tête de la routine générée en mode inverse est :

```
samplead (a,b,c,d,e,aad,bad,dad,ead)
```

On observe qu'en mode inverse ODYSSÉE utilise le suffixe `ad` pour désigner à la fois la routine générée `samplead` et les variables duales `aad`, `bad`, `dad`, `ead` associées aux variables actives `a`, `b`, `d` et `e`.

Si `sample` est la procédure de tête (et donc `samplead` est la première routine à être appelée dans le code linéaire cotangent), les entrées `dad` et `ead` de `samplead` doivent être initialisées à une direction $d \in \mathbb{R}^2$ si on veut calculer le gradient de $d^\top \binom{\mathtt{d}}{\mathtt{e}}$. Les autres variables duales sont initialisées à 0. Par exemple si on veut calculer le gradient de `e`, on initialise `ead` à 1 et les autres variables duales à 0. Si on veut calculer le gradient de `d`, c'est `dad` qu'il faut initialiser à 1 et les autres variables duales à 0. Sinon les valeurs duales en entrée de `samplead` sont données par la routine qui l'appelle. Les valeurs duales se propagent ainsi d'une routine à l'autre à partir des valeurs données aux variables duales avant l'exécution de la première routine du code linéaire cotangent.

En fin d'exécution de `samplead`, on aura (voir formules (1.7)) :

$$\begin{aligned}
\mathtt{aad} &= \mathtt{aad} + \frac{\partial \mathtt{d}}{\partial \mathtt{a}}\,\mathtt{dad} \\
\mathtt{bad} &= \mathtt{bad} + \frac{\partial \mathtt{d}}{\partial \mathtt{b}}\,\mathtt{dad} + \frac{\partial \mathtt{e}}{\partial \mathtt{b}}\,\mathtt{ead} \\
\mathtt{dad} &= 0 \\
\mathtt{ead} &= 0.
\end{aligned}$$

Comme nous l'avons vu en section 1.3, en mode inverse, il est nécessaire de disposer de toute la *trajectoire*, c'est-à-dire de mémoriser tout au long de l'exécution du code original des quantités (à préciser) permettant de calculer les dérivées partielles $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$ intervenant dans (1.4). La stratégie mise en œuvre dans la version 1.5 d'ODYSSÉE consiste à sauvegarder une variable (argument, globale ou intermédiaire) avant chaque modification de celle-ci. Sur le code engendré, cela revient à augmenter le code original d'instructions de sauvegarde. À une instruction du code original augmenté comportant des variables actives correspond sa forme linéaire transposée dans le code linéaire cotangent et à une instruction de sauvegarde correspond l'instruction réciproque de remise de la valeur.

Sauvegarder les variables modifiées n'est peut être pas une stratégie optimale mais semble préférable à la sauvegarde des dérivées $\frac{\partial \varphi_k}{\partial v_i}(v^{k-1})$ intervenant dans (1.4). En effet, cela ne demande la sauvegarde que d'un scalaire par instruction. Ces dérivées sont alors calculées dans la phase régressive du code. Avec cette stratégie on peut diminuer le stockage en regroupant plusieurs instructions en une seule.

Le code linéaire cotangent d'une unité généré par ODYSSÉE se compose en fait de trois parties :

1. initialisation à zéro des variables duales locales,
2. exécution du code original augmenté des sauvegardes,
3. calcul des variables duales en mode inverse.

Ceci a pour conséquence qu'une unité de profondeur p est exécutée p fois. Avec cette structure, toutes les sauvegardes peuvent être faites dans des variables locales.

Si une variable scalaire est modifiée à chaque tour d'une boucle, toutes ses valeurs vont être sauvegardées dans un tableau local qui aura pour taille le nombre de tours de boucle. Si ce nombre est passé en paramètre ou est une variable globale du programme, le compilateur va refuser cette déclaration (tableau local déclaré avec une dimension inconnue à priori). Si la variable `n` contient le nombre de tours de boucle, ODYSSÉE donne la taille `Odyn` aux sauvegardes de manière à ce que l'utilisateur puisse savoir que la valeur à donner à `Odyn` est `n`. Il ne déclare pas le `parameter` correspondant à `Odyn` car une valeur par défaut générerait des codes inutilement gros puisque l'utilisateur connaît la taille de ces symboles. Si la taille ne peut pas être calculée statiquement (cas d'un processus itératif qui termine selon un critère dépendant de l'exécution), ODYSSÉE lui donne la taille `Odysseemax` qui est déclarée en `parameter` et dont la valeur (10000 par défaut) peut être modifiée par l'utilisateur.

3.4 Limitations

3.4.1 Dérivées de sortie

La dérivée d'une unité par rapport à des entrées calcule les dérivées de toutes les sorties dépendant de ces entrées. Il n'est pas possible pour l'instant de calculer la dérivée de certaines sorties seulement, ceci est surtout dommageable en mode inverse.

3.4.2 Gestion des équivalences

Le système vérifie que les `commons` sont déclarés de la même façon dans toutes les unités du programme. Les équivalences gérées par redécoupage de commons d'une routine à l'autre ne sont pas acceptés.

Les équivalences entres symboles ne sont pas gérées pour le calcul de dépendance, ce qui pose problème si les deux formes équivalencées sont actives. ODYSSÉE ne fait aucune vérification. C'est donc à l'utilisateur de vérifier (pour l'instant).

3.4.3 Nommage des dérivées

Le nommage des dérivées n'est pas sûr, c'est-à-dire que le système ne vérifie pas qu'un symbole n'est pas utilisé avant de l'associer à une dérivée.

En linéaire tangent si `a` et `atl` sont deux symboles du code initial, il risque d'y avoir conflit sur `atl` qui est le nom de la dérivée de `a`.

Pour palier à ce problème, il faut vérifier qu'aucun symbole du code initial ne finisse par `tl` en linéaire tangent et `ad` en linéaire cotangent.

3.4.4 En mode inverse

En mode inverse les sauts (`goto`, ...) dans le code ne sont pas gérés. ODYSSÉE provoque une erreur s'il en rencontre. C'est à l'utilisateur (pour l'instant) de transformer le code en utilisant des `dowhile` ou des `if-then-else`. Cette transformation que nous automatiserons dans ODYSSÉE n'est pas toujours intéressante à effectuer car elle introduit de nombreuses variables booléennes.

3.4.5 Autres

Les codes des fichiers d'`include` sont "in-linés"(reproduits) dans le code. Les instructions de lecture/écriture sont commentées.

Chapitre 4

Mise en œuvre d'ODYSSÉE sur THYC-1D

4.1 Les variables à dériver

Les variables qui vont être utilisées dans le calcul des dérivées sont les suivantes :

- `ur` (vitesse relative de la vapeur par rapport au liquide),
- `cd` (coefficient de frottement interfacial),
- `qsi` (coefficient de masse virtuelle),
- `tau` (temps de relaxation),
- `puisvol` (puissance volumique apportée) et
- `urentre` (vitesse relative vapeur-liquide en entrée).

La vitesse relative `ur` varie en temps et en espace (25 points de discrétisation). Les autres variables sont des scalaires.

En fait, on s'intéressera essentiellement aux dérivées de `ur` par rapport aux entrées `cd`, `qsi`, `tau`, `puisvol` et `urentre`. Si on s'intéresse aux 25 composantes de `ur` à un instant donné, il est intéressant de générer le code linéaire tangent (mode direct). Si par contre on s'intéresse à `ur` en une position et un temps donnés, il est préférable de générer le code linéaire cotangent (mode inverse). Dans ce dernier cas, il s'est agit de dériver le maximum en temps et en espace de `ur`. Plus précisément, nous avons repéré le maximum de la vitesse relative, qui s'est produit au pas de temps 306 et à la composante `idx` $= 23$ de `ur`. C'est donc ensuite `ur(23)` au 306-ième pas de temps, noté `ur(306,23)`, qui a été dérivé.

4.2 Modification du code THYC-1D

Les limitations actuelles d'ODYSSÉE (voir section 3.4) nous ont contraint à modifier le code de THYC-1D.

Tout d'abord comme ODYSSÉE ne dérive que des `subroutines`, nous avons formé avec une partie du programme principal `princp`, la sous-routine `princp_sub` qui ne fait que des calculs. C'est la routine de tête. Cette routine possède un argument `npas` qui est la seule variable locale de `princp`. De plus, nous avons modifié la routine `table` pour qu'elle appelle, non pas les fonctions tabulées, mais des routines de même nom ayant les mêmes entrées/sorties. Comme ces routines n'existent pas, cette transformation rend le code à dériver non exécutable. Nous appellerons `princp1` le programme contenant les routines de `princp`, mais avec la routine `princp_sub` à la place du `princp` initial et la routine `table` modifiée.

Pour le mode inverse, il nous a fallu remplacer dans l'unité `princp_sub` le `do ... goto` par un `dowhile` puisqu'ODYSSÉE ne dérive pas les `goto`. La transformation effectuée est celle qui sera automatisée dans une version future d'ODYSSÉE. De plus, nous avons introduit une variable globale `uridx` qui correspond à la composante `ur(idx)` au pas de temps `npas`. Pour les tests, le pas `npas` est fixé à `306` et `idx` a pour valeur `23`. Nous appellerons `princp2` le programme contenant les routines de `princp1`, excepté la routine `princp_sub` qui est la routine modifiée.

4.3 Dérivées de THYC-1D

Une dérivée de THYC-1D se compose des routines écrites à la main, des routines générées par ODYSSÉE et du programme appelant, lui aussi écrit à la main. Les différentes étapes permettant d'arriver aux codes exécutables sont donc les suivantes :

- écriture manuelle des routines dérivées que l'on ne peut pas ou ne veut pas dériver par ODYSSÉE (voir section 4.3.1),
- dérivation par ODYSSÉE des autres routines (voir section 4.3.2),
- écriture des programmes principaux `princptl` et `princpad` appelant les routines de tête dérivées obtenues par ODYSSÉE à l'étape précédente (voir sections 4.3.3 et 4.3.4).

4.3.1 Dérivées manuelles

Dans cette section nous présentons des exemples de routines écrites à la main : les dérivées en mode direct et en mode inverse d'une fonction tabulée et d'un solveur linéaire.

Pour les fonctions tabulées, les codes linéaire tangent et linéaire cotangent utilisent les différences finies. Le pas de discrétisation dépend de l'epsilon-machine ϵ_M (dans le `common /ceps/myeps`) qui est calculé par la fonction `meps` appelée au début du calcul des dérivées (voir figures 4.1 et 4.2). L'annexe B donne les routines écrites pour la fonction tabulée `tba2ps`. En mode direct, deux types de différences finies ont été testées. Dans `tba2pstl`, on utilise des différences décentrées à droite avec un seul pas calculé par la formule (A.5). Dans `tba2ps2tl`, on utilise des différences centrées. Le pas donné par la formule (A.5) est comparé à celui donné par (A.6). Si le premier diffère du second dans un rapport inférieur à 10, c'est lui qui est choisi, sinon c'est le second pas (en principe meilleur) qui est choisi. Ce test permet d'éviter parfois une évaluation de fonction supplémentaire. En mode inverse, nous avons utilisé les formules (1.7). La jacobienne de la transformation qui y est utilisée a été calculée par différences finies décentrées à droite avec un pas donné par (A.4).

Les codes linéaire tangent et linéaire cotangent des solveurs linéaires ont été écrits à la main également, pour des raisons d'efficacité comme cela est expliqué en section 2.2.2. Les formules (1.9) et (1.12) ont été utilisées pour obtenir le linéaire tangent et le linéaire cotangent respectivement. Les codes écrits pour le solveur `tdma` sont donnés en annexe C. On remarquera que les systèmes linéaires à résoudre dans ces codes le sont par la sous-routine `tdma` elle-même.

4.3.2 Dérivées générées par ODYSSÉE

Pour étudier la sensibilité de `uridx` aux variations de `cd`, `qsi`, `tau`, `puisvol` et `urentre`, nous avons choisi de dériver `princp2` en mode direct (linéaire tangent) et en mode inverse (linéaire cotangent).

Nous avons généré avec ODYSSÉE les dérivées des unités de `princp2` excepté `jacobi`, `tdma` et les routines tabulées pour lesquelles une base d'informations de nom `princp_sub_BI.od` a été écrite à la main. On peut remarquer qu'ODYSSÉE dérive pour l'instant toutes les unités qu'il a lues, il nous a donc fallu décrire `jacobi, tdma` dans la `BI` pour ne pas en générer les dérivées.

Le fichier `princp_sub_BI.od` contient pour chaque unité décrite, les noms de ses arguments déclarés par la commande `set_info_dummys`, les noms de ses globales par `set_info_globals`. Parmi ces noms, les entrées et les sorties sont déclarées par `set_info_in_out`.

Voici par exemple les déclarations de `princp_sub_BI.od`, écrites dans le language d'ODYSSÉE, correspondant à la fonction tabulée `tbalps` :

```
set_info_dummys "tbalps" ["p1";"p2";"p3";"p4";"p5";"tbalps"];;
set_info_in_out "tbalps" ["p2";"p3"] [] ["tbalps"] [];;
```

et celles correspondant au solveur linéaire `tdma` :

```
set_info_dummys "tdma" ["c";"a";"d";"b";"x";"imax"];;
set_info_in_out "tdma" ["c";"a";"d";"b";"x"] []
                       ["c";"a";"d";"b";"x"] [];;
```

Le déroulement de la dérivation d'un ensemble d'unités à l'interface d'ODYSSÉE se fait en 5 étapes :

1. chargement des fichiers,
2. choix du programme à dériver (par désignation de la routine de tête),
3. choix du mode de dérivation,
4. choix des variables par rapport auxquelles dériver,
5. sauvegarde des unités dérivées.

À titre d'illustration, on trouvera en annexe D les codes linéaire tangent `solvtit1tl` et linéaire cotangent `solvtit1ad` générés par ODYSSÉE à partir de `solvtit1`.

Le code ainsi généré n'est pas directement exécutable. Il nous a fallu faire les modifications suivantes pour pouvoir l'exécuter. Lors de la dérivation en mode inverse, ODYSSÉE a introduit deux paramètres : `Odyn` auquel ODYSSÉE n'a pas donné de valeur et `Odysseemax` qui correspond au nombre de tours effectués dans la boucle principale et qui par défaut a été mis à 10000. Par définition `Odyn` doit avoir la même valeur que le nombre `n` de tours de boucles en espace (voir section 3.3.2). Nous avons donc défini `Odyn` par

```
parameter (Odyn = 25)
```

Comme 10000 nous a paru énorme pour `Odysseemax`, nous avons calculé le nombre de tours de boucles effectifs de `princp2` qui vaut 500, pour le jeu de données utilisé. Nous avons donc défini `Odysseemax` par

```
parameter (Odysseemax = 500)
```

De plus, pour que la dérivée soit exécutable, il a fallu remplacer les appels fictifs aux routines tabulées en appels aux fonctions tabulées dans `tablead`.

4.3.3 Le programme `princptl`

Le programme `princptl` est le programme principal du code linéaire tangent. Il est écrit à la main. Il garde de `princp` les instructions de lecture-écriture initiales et les instructions d'initialisations. Il prépare ensuite l'appel à `princp_subtl` le code dérivé en mode direct de la routine de tête `princp_sub`, pour une dérivée par rapport à `cd`, `qsi`, `tau`, `puisvol` et `urentre`.

Plus précisément, le programme principal `princptl` se compose des parties suivantes :

1. les déclarations initiales,
2. les déclarations concernant les symboles des dérivées,
3. les lectures des entrées dans le fichier `data`,
4. l'initialisation de `npas`, `idx` et `myeps` (`myeps` contient la valeur de l'epsilon-machine ϵ_M calculé par la fonction `meps` que nous avons écrite),
5. le calcul de la dérivée de `uridx` par rapport à `cd` en appelant `princp_subtl`; au retour, la dérivée se trouve dans `uridxtl` :

```
c
c Derivation par rapport a CD
c
      cdtl = 1.0
      qsitl = 0.
      tautl = 0.
      puisvoltl = 0.
      urentretl = 0.
      call princp_subtl (npas)
      print *, "uridxtl = ", uridxtl
```

6. le calcul de la dérivée de `uridx` par rapport à `qsi` en appelant à nouveau `princp_subtl`; au retour, la dérivée se trouve dans `uridxtl` :

```
C -- Itialisation

      call init_zero ()

C -- Fin Reinitialisation
c
c Derivation par rapport a QSI
c
      cdtl = 0.
      qsitl = 1.0
      tautl = 0.
      puisvoltl = 0.
```

```
princptl +- meps
         +- ouvrir
         +- init_zero
         +- princp_subtl +- (table)
                         +- solvptl +- tdmatl +- (tdma)
                         +- solvq1tl +- jacobitl +- (jacobi)
                         +- solvtit1tl +- jacobitl +- (jacobi)
                         +- solvtittl +- jacobitl +- (jacobi)
                         +- solvs1tl +- jacobitl +- (jacobi)
                         +- solvur1tl +- jacobitl +- (jacobi)
                         +- actuatl +- tabletl +- tbttpstl +- (tbttps)
                                               +- tbttsatl +- (tbttsa)
                                               +- tbsssatl +- (tbsssa)
                                               +- tbropstl +- (tbrops)
                                               +- tba2pstl +- (tba2ps)
                                               +- tbalpstl +- (tbalps)
                                               +- tbcppstl +- (tbcpps)
                                               +- tbetpstl +- (tbetps)
                                               +- tbssphtl +- (tbssph)
                                               +- tbbbpstl +- (tbbbps)
                                               +- tbhhpstl +- (tbhhps)
                                               +- tbhhsatl +- (tbhhsa)
```

FIG. 4.1 – *Graphe d'appel des routines du programme tangent de* THYC-1D

```
urentretl = 0.
call princp_subtl (npas)
print *, "uridxtl = ", uridxtl
```

7. et ainsi de suite pour toutes les entrées actives de `princp`.

Si la routine `princp_sub` de `princp2` est appelée deux fois avec les mêmes entrées, elle ne calcule pas les mêmes sorties, du fait que certaines variables globales ont été modifiées en cours d'exécution. Entre deux appels successifs de `princp_subtl`, il est donc nécessaire de réinitialiser les variables globales à zéro, ce qui se fait dans la routine `init_zero`.

On remarque que pour calculer la dérivée de `uridx` par rapport aux cinq entrées actives de `princp`, il faut appeler cinq fois `princp_subtl`.

On peut voir dans la figure 4.1, le graphe d'appel des routines de `princptl` généré par ODYSSÉE. Nous avons choisi de limiter le graphe aux unités générées par ODYSSÉE et n'avons donc pas lu les routines du code initial (`table`, `tdma`, ...). Les parenthèses autour des noms de routines indiquent que la routine n'a pas été chargée par ODYSSÉE.

4.3.4 Le programme `princpad`

Le programme `princpad` est le programme principal du code linéaire cotangent. Il est écrit à la main. Comme `princptl`, il garde de `princp` les instructions de lecture-écriture initiales et les instructions d'initialisations. Il prépare ensuite l'appel à `princp_subad` le code dérivé en mode inverse de la routine de tête `princp_sub`, pour une dérivée par rapport à `cd`, `qsi`, `tau`, `puisvol` et `urentre`.

Plus précisément, le programme principal `princpad` se compose des parties suivantes :

1. les déclarations initiales,
2. les déclarations concernant les symboles des dérivées,
3. les lectures des entrées dans le fichier `data`,
4. l'initialisation de `npas`, `idx` et `myeps`,
5. le calcul simultané de toutes les dérivées : la dérivée de `uridx` par rapport à `cd` se trouve dans `cdad`, celle par rapport à `qsi` dans `qsiad`, *etc*,

```
cdad = 0.
qsiad = 0.
tauad = 0.
puisvolad = 0.
urentread = 0.
uridxad = 1.0
call princp_subad (npas)
print *, "cdad = ", cdad
print *, "qsiad = ", qsiad
print *, "tauad = ", tauad
print *, "puisvolad = ", puisvolad
print *, "urentread = ", urentread
```

On remarque que pour calculer la dérivée de `uridx` par rapport aux cinq entrées actives de `princp`, il faut ici n'appeler qu'une seule fois `princp_subad`.

On peut voir dans la figure 4.2, le graphe d'appel des routines de `princpad` généré par ODYSSÉE.

De même que dans le graphe d'appel des routines de `princptl`, nous avons choisi de limiter le graphe aux unités générées par ODYSSÉE.

```
princpad +- meps
         +- ouvrir
         +- princp_subad +- (table)
                         +- solvp
                         +- solvpad +- (tdma)
                                    +- tdmaad +- (tdma)
                         +- solvq1
                         +- solvq1ad +- (jacobi)
                                     +- jacobiad +- (jacobi)
                         +- solvtit1
                         +- solvtit1ad +- jacobi
                                       +- jacobiad +- jacobi
                         +- solvtit
                         +- solvtitad +- (jacobi)
                                      +- jacobiad +- (jacobi)
                         +- solvs1
                         +- solvs1ad +- (jacobi)
                                     +- jacobiad +- (jacobi)
                         +- solvur1
                         +- solvur1ad +- (jacobi)
                                      +- jacobiad +- (jacobi)
                         +- (actua)
                         +- actuaad +- (table)
                                    +- tablead +- (tbalps)
                                               +- tbalpsad +- (tbalps)
                                               +- (tba2ps)
                                               +- tba2psad +- (tba2ps)
                                               +- (tbhhps)
                                               +- tbhhpsad +- (tbhhps)
                                               +- (tbhhsa)
                                               +- tbhhsaad +- (tbhhsa)
                                               +- (tbcpps)
                                               +- tbcppsad +- (tbcpps)
                                               +- (tbssph)
                                               +- tbssphad +- (tbssph)
                                               +- (tbetps)
                                               +- tbetpsad +- (tbetps)
                                               +- (tbbbps)
                                               +- tbbbpsad +- (tbbbps)
                                               +- (tbsssa)
                                               +- tbsssaad +- (tbsssa)
                                               +- (tbttsa)
                                               +- tbttsaad +- (tbttsa)
                                               +- (tbttps)
                                               +- tbttpsad +- (tbttps)
                                               +- (tbrops)
                                               +- tbropsad +- (tbrops)
```

FIG. 4.2 – *Graphe d'appel des routines du programme cotangent de* THYC-1D

Chapitre 5

Résultats et analyse

Les tests que nous allons décrire ont été réalisés sur une SPARCstation 20 de SUN en Fortran-77 double précision (sauf mention contraire). On cherche par ces essais à valider la valeur des dérivées calculées par ODYSSÉE, à estimer le temps nécessaire au calcul de ces dérivées et à apprécier l'encombrement-mémoire.

Nous avons choisi d'étudier les dérivées de la vitesse relative `ur` au pas de temps 306, auquel `ur` est maximal (par sa composante 23). On peut alors calculer soit les dérivées des 25 composantes de `ur` en ce pas de temps (correspondant aux 25 points de discrétisation en espace), soit les dérivées de `ur(306,23)`, composante maximale de `ur`. Les dérivées sont calculées par rapport aux variables d'entrée `cd`, `qsi`, `tau`, `puisvol` et `urentre` (voir section 4.1 pour la signification de ces variables).

5.1 Comparaison entre linéaire tangent et linéaire cotangent

Les dérivées en mode direct ont été obtenues par appel du code linéaire tangent pour chaque entrée active (donc 5 fois), alors qu'en mode inverse il a fallu appeler le code linéaire cotangent pour chaque composante de `ur` (donc 25 fois).

Nous avons testé plusieurs codes linéaires tangents correspondant à des codages manuels différents des routines dérivées associées aux fonctions tabulées (voir section 4.3.1). Les différences portent sur la manière de calculer les différences finies. Le premier essai a été fait avec un code linéaire tangent utilisant des différences finies décentrées avec un seul essai de pas. On observait alors une dégradation appréciable de la concordance entre les résultats des codes linéaire tangent et linéaire cotangent, à partir de la vingtième composante. En utilisant un calcul par différences finies plus précis (schéma centré avec deux essais de pas, voir section 4.3.1), la concordance est devenue à peu près uniforme en espace et a été globalement améliorée (gain d'environ 1.5, 2, 3 et 1 chiffres significatifs communs aux deux codes pour les dérivées par rapport à `cd`, `qsi`, `tau` et `puisvol`, respectivement).

Les résultats comparatifs que nous présentons ci-après ont été obtenus en utilisant une troisième méthode : on calcule la jacobienne de la transformation réalisée par les fonctions tabulées, plutôt qu'une dérivée directionnelle. Pour la calculer, on a utilisé des différences finies décentrées avec un seul essai de pas. La raison de ce choix vient de ce que c'est la jacobienne qui est évaluée en mode inverse (c'est presque toujours nécessaire dans notre cas, parce que les fonctions tabulées sont le plus souvent à valeurs scalaires, voir section 2.2.1) et qu'il nous a paru intéressant de comparer les modes direct et inverse lorsque les fonctions intermédiaires étaient différenciées de la même manière. Avec ce choix, la concordance entre les codes linéaire tangent et linéaire cotangent s'améliore encore légèrement (parfois 0.5 chiffre significatif commun en plus, par rapport au second choix).

Pour certaines composantes de `ur`, nous avons généré deux codes linéaires cotangents. Le premier correspond à la dérivation de Thyc-1D par rapport aux cinq entrées simultanément, le second correspond à la dérivation de Thyc-1D par rapport à `cd` uniquement. Les deux dérivées obtenues étaient identiques. Nous présenterons les résultats obtenus avec le premier code.

Les figures 5.1, 5.3, 5.5, 5.7 et 5.9 donnent la valeur des dérivées des 25 composantes de `ur` par rapport à `cd`, `qsi`, `tau`, `puisvol` et `urentre` respectivement, obtenues par le code linéaire tangent. Nous n'avons pas tracé les dérivées calculées en mode inverse car, sur ces graphiques, ce tracé est indiscernable de celui du linéaire tangent.

La comparaison des valeurs obtenues par les deux modes de différentiation, direct et inverse, peut se voir sur les figures 5.2, 5.4, 5.6, 5.8 et 5.10. L'échelle des ordonnées est logarithmique en base 10 et la courbe en trait plein donne l'écart relatif entre les valeurs des dérivées pour chaque composante. La concordance est à peu près uniforme en espace et en général très bonne puisque les dérivées obtenues par les deux modes ont entre 9 et 11.5 chiffres significatifs communs lorsqu'on dérive par rapport à `cd`, `qsi`, `tau` et `puisvol`. Le tableau 5.1 donne la valeur des dérivées de `ur(306,23)`, obtenues par les deux modes de différentiation.

	linéaire tangent	linéaire cotangent
`cd`	$-2.6877707476674D+00$	$-2.6877707476626D+00$
`qsi`	$-1.1283833581385D-02$	$-1.1283833581488D-02$
`tau`	$2.3953473434534D-01$	$2.3953473304208D-01$
`puisvol`	$8.4189731568450D-09$	$8.4189731614290D-09$
`urentre`	$-1.1112453959230D-14$	$2.0851403488688D-16$

Tab. 5.1 – *Dérivées de* `ur(306,23)` *par* Odyssée

La dérivée par rapport à `urentre` est particulière. Il s'agit en effet de dériver la vitesse relative obtenue au régime stationnaire atteint en fin de simulation par rapport à sa valeur de condition limite. Le régime stationnaire est apparemment indépendant de cette valeur puisque la dérivée obtenue est pratiquement nulle (de l'ordre de la précision-machine). La figure 5.10 compare donc entre elles des valeurs presque nulles. Les résultats chaotiques que l'on peut observer sont donc peu significatifs.

5.2 Comparaison avec les différences finies

Les résultats obtenus avec les codes générés par Odyssée ont été confrontés à ceux obtenus par différences finies centrées. Nous avons procédé de la manière suivante.

Dans un premier temps, nous avons calculé par différences finies les dérivées de la variable `uridx` (composante 23 de `ur` au pas de temps 306) par rapport aux variables habituelles (`cd`, `qsi`, `tau`,

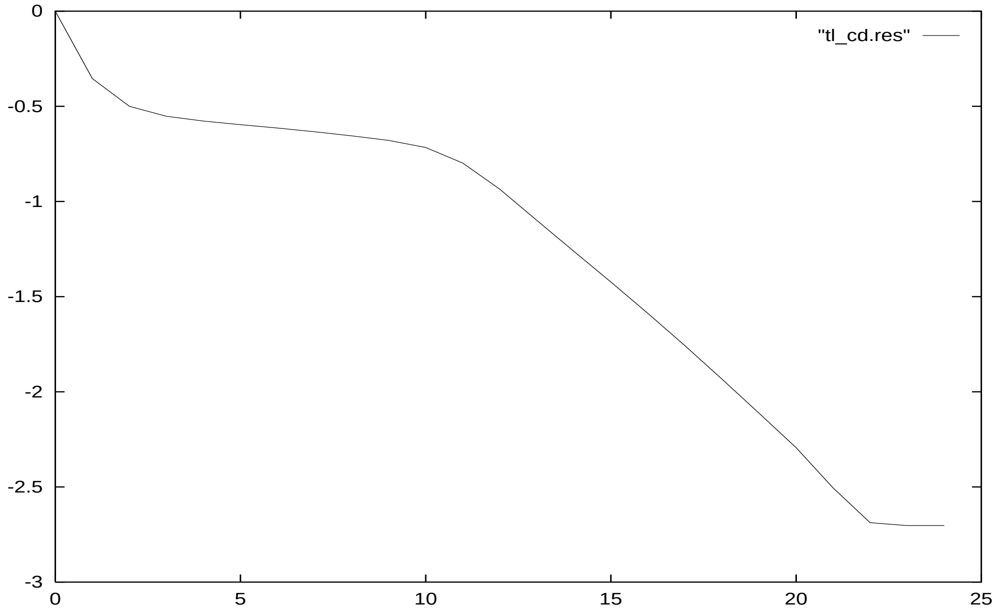

FIG. 5.1 – *Dérivée en linéaire tangent par rapport à* `cd`

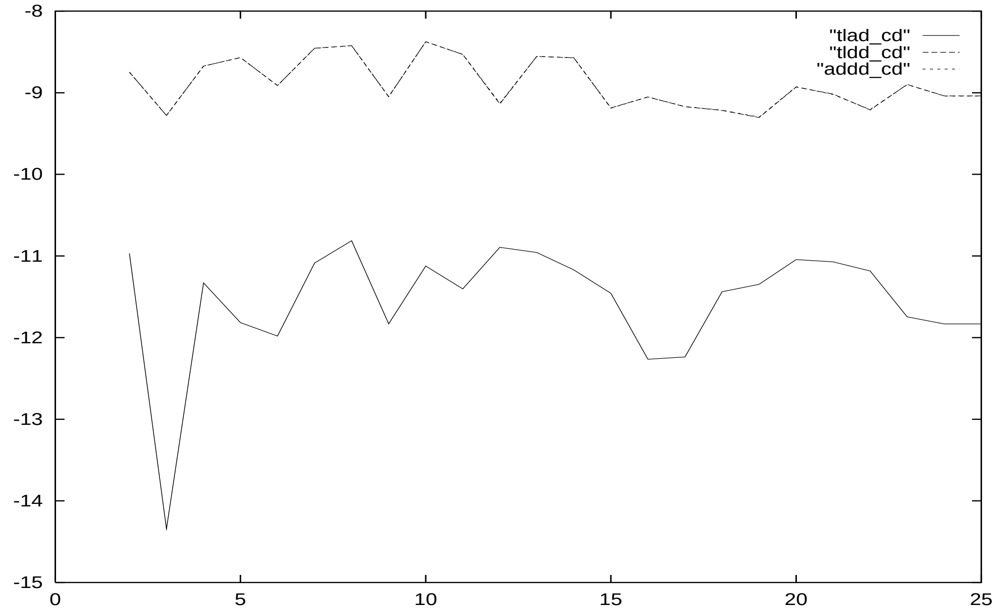

FIG. 5.2 – *Différences entre les dérivées par rapport à* `cd`

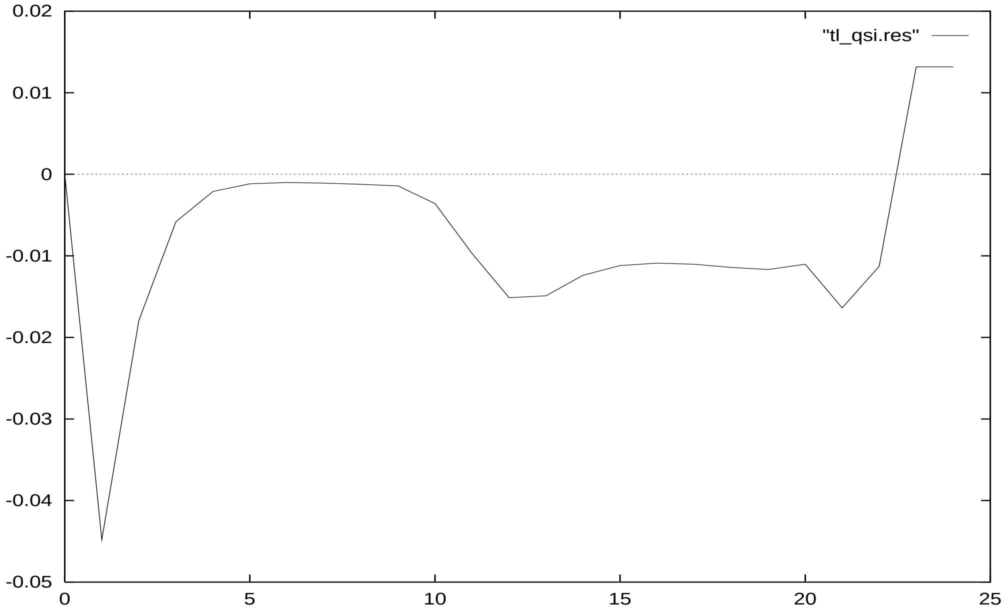

Fig. 5.3 – *Dérivée en linéaire tangent par rapport à* `qsi`

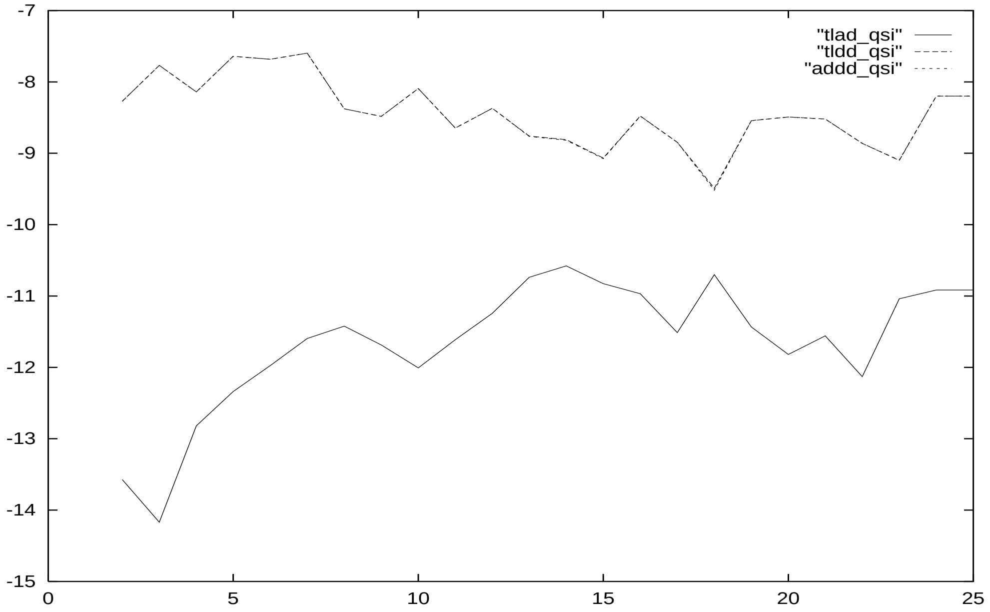

Fig. 5.4 – *Différences entre les dérivées par rapport à* `qsi`

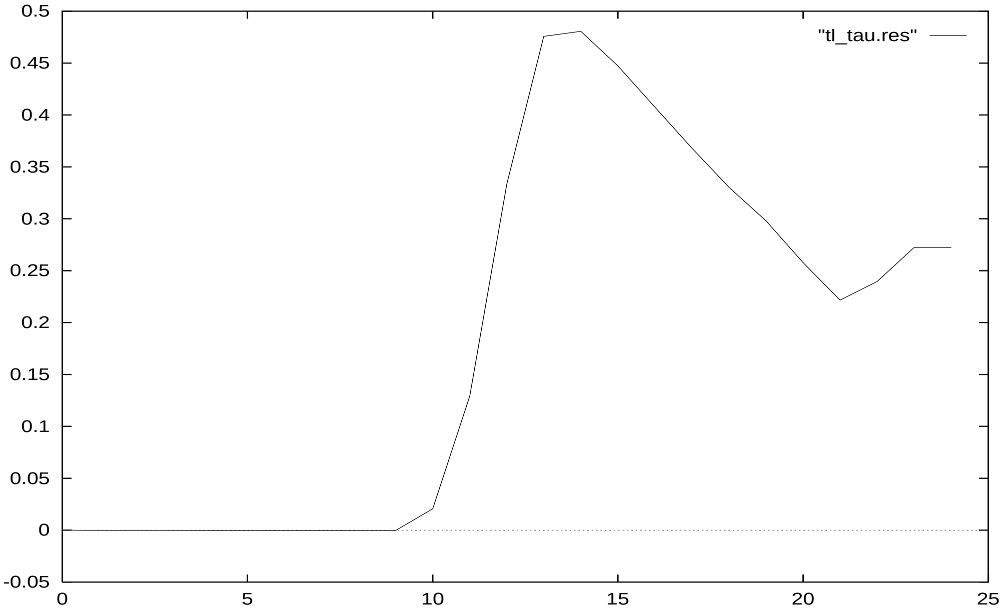

Fig. 5.5 – *Dérivée en linéaire tangent par rapport à* `tau`

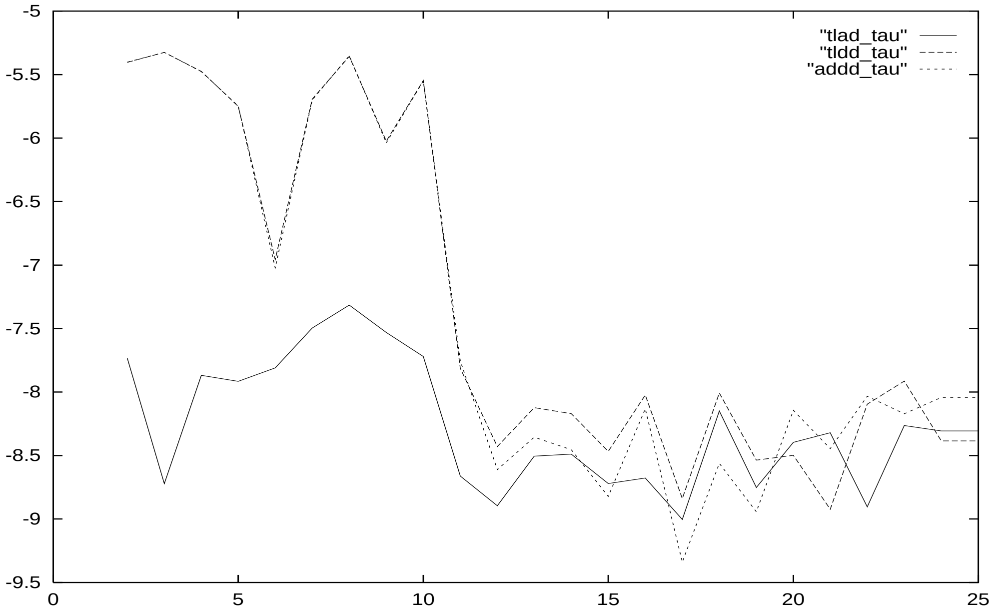

Fig. 5.6 – *Différences entre les dérivées par rapport à* `tau`

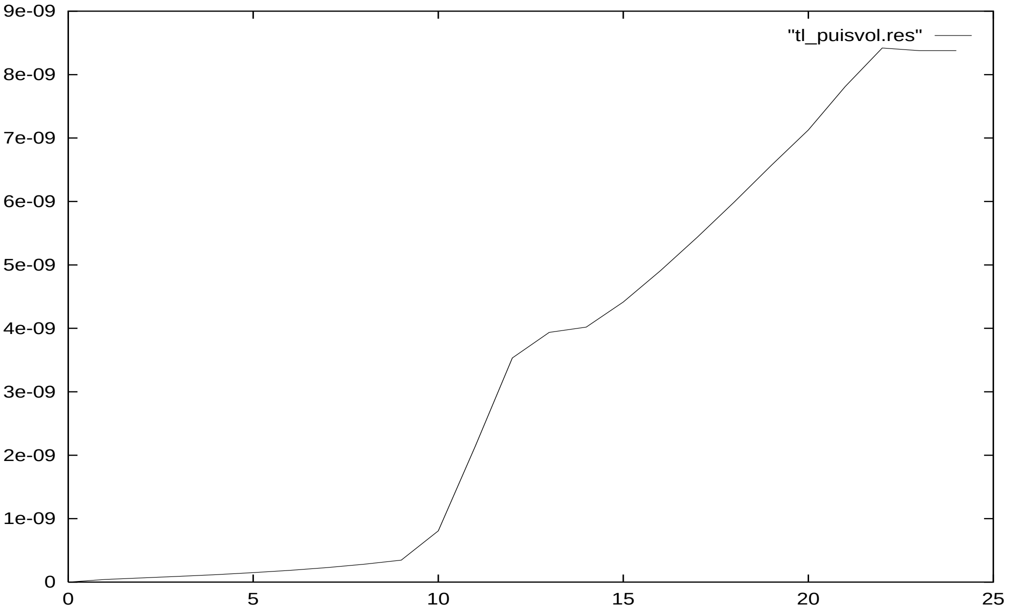

FIG. 5.7 – *Dérivée en linéaire tangent par rapport à* `puisvol`

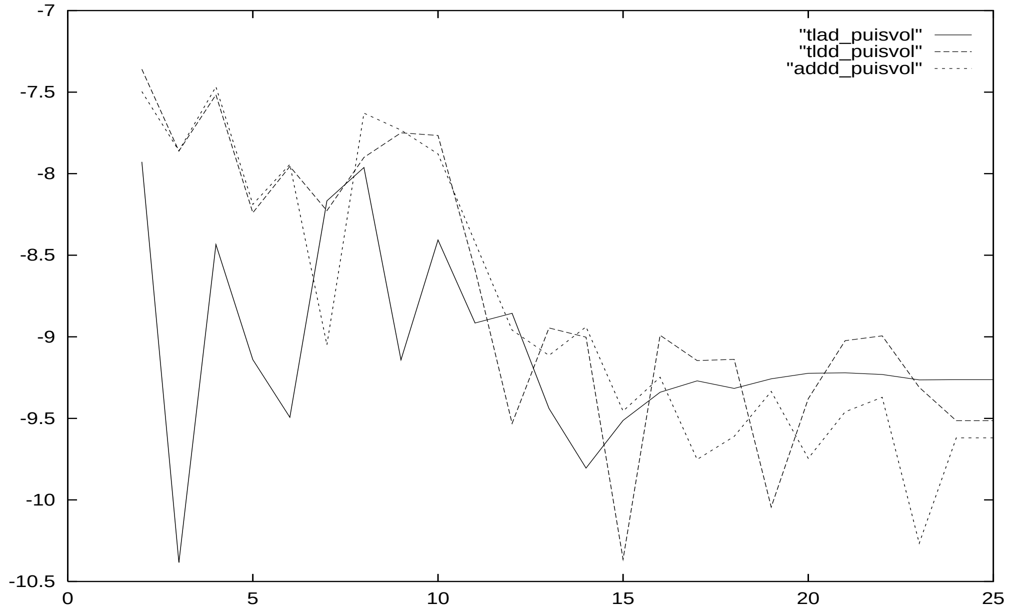

FIG. 5.8 – *Différences entre les dérivées par rapport à* `puisvol`

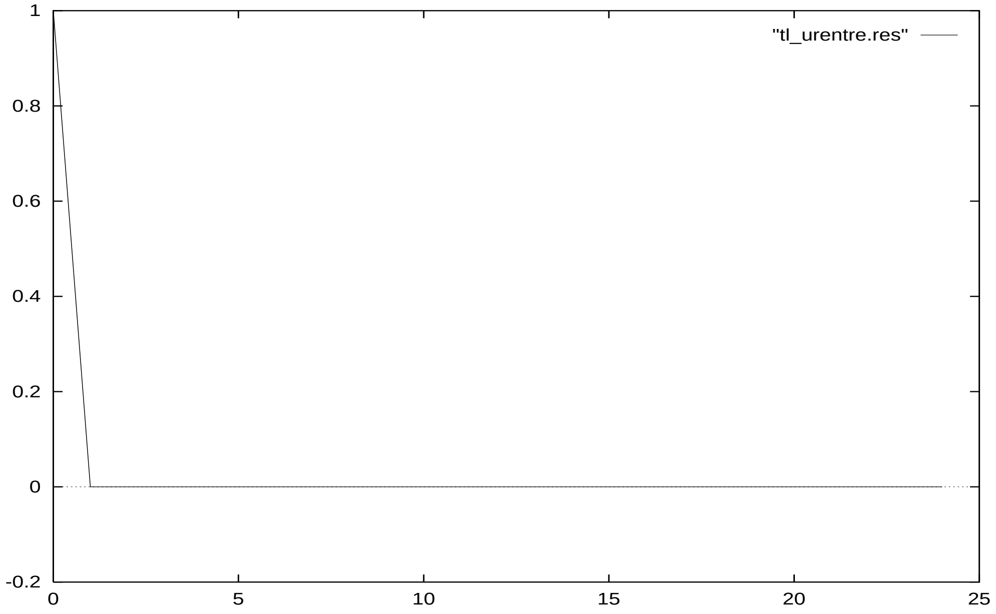

FIG. 5.9 – *Dérivée en linéaire tangent par rapport à* `urentre`

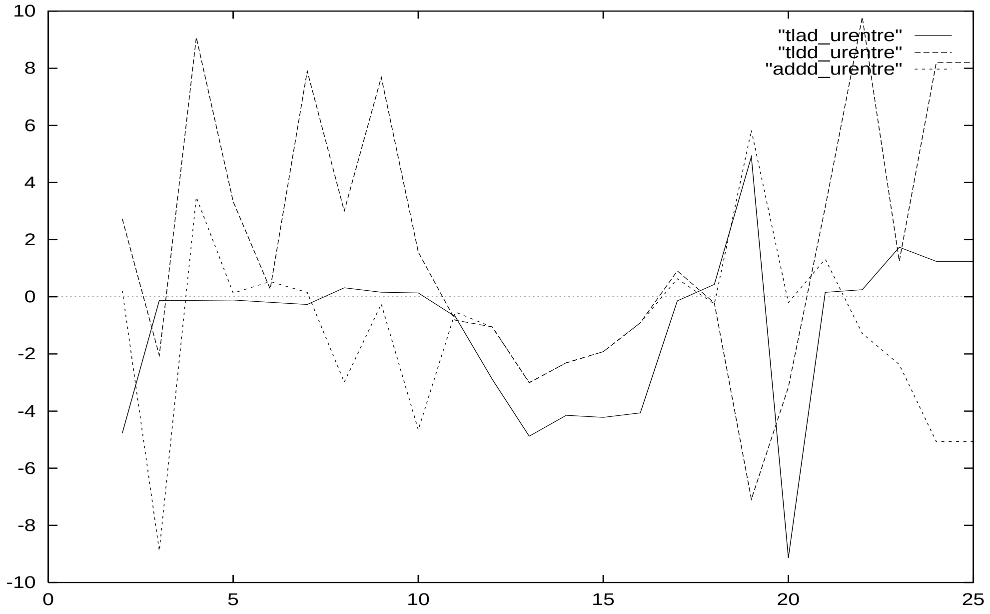

FIG. 5.10 – *Différences entre les dérivées par rapport à* `urentre`

`puisvol` et `urentre`), en faisant varier le pas de manière à obtenir le plus de chiffres significatifs corrects. Pour cela, on s'est basé sur les résultats donnés dans les tableaux qui suivent. On y trouve en première colonne le pas t, en seconde $\mathtt{ur}(x-t)$, en troisième $\mathtt{ur}(x+t)$ et en dernière colonne les différences finies centrées $(\mathtt{ur}(x+t)-\mathtt{ur}(x-t))/(2t)$ (x désigne ici la variable par rapport à laquelle on dérive). Comme décrit en annexe A.5, il s'agit de repérer dans la dernière colonne les chiffres les plus significatifs qui sont conservés d'un pas à l'autre.

cd = 7.E-01

h	$f(x+h)$	$f(x-h)$	$\frac{f(x+h)-f(x-h)}{h}$
$5.E-11$	$2.370420869842508E+00$	$2.370420869573734E+00$	$-2.687743361207140E+00$
$5.E-10$	$2.370420871052076E+00$	$2.370420868364278E+00$	$-2.687798428269161E+00$
$5.E-09$	$2.370420883147005E+00$	$2.370420856269294E+00$	$-2.687771161191677E+00$
$5.E-08$	$2.370421004096704E+00$	$2.370420735319601E+00$	$-2.687771023524022E+00$
$5.E-07$	$2.370422213594315E+00$	$2.370419525823556E+00$	$-2.687770758846852E+00$
$5.E-06$	$2.370434308640004E+00$	$2.370407430932561E+00$	$-2.687770744280727E+00$
$5.E-05$	$2.370555266062323E+00$	$2.370286488986641E+00$	$-2.687770756817365E+00$
$5.E-04$	$2.371765537186346E+00$	$2.369077765505365E+00$	$-2.687771680981221E+00$
$5.E-03$	$2.383938356748383E+00$	$2.357059715861638E+00$	$-2.687864088674541E+00$
$5.E-02$	$2.513122925593328E+00$	$2.243408892271010E+00$	$-2.697140333223182E+00$

qsi = 1.E+00

$5.E-09$	$2.370420869764565E+00$	$2.370420869651737E+00$	$-1.128279691897660E-02$
$5.E-08$	$2.370420870272332E+00$	$2.370420869143967E+00$	$-1.128364957025951E-02$
$5.E-07$	$2.370420875350079E+00$	$2.370420864066220E+00$	$-1.128385918036656E-02$
$5.E-06$	$2.370420926127311E+00$	$2.370420813288965E+00$	$-1.128383457782434E-02$
$5.E-05$	$2.370421433899133E+00$	$2.370420305515793E+00$	$-1.128383339654704E-02$
$5.E-04$	$2.370426511556488E+00$	$2.370415227722915E+00$	$-1.128383357240637E-02$
$5.E-03$	$2.370477282026962E+00$	$2.370364443701427E+00$	$-1.128383255357690E-02$
$5.E-02$	$2.370984371775951E+00$	$2.369855998707340E+00$	$-1.128373068611577E-02$
$5.E-01$	$2.375988400118305E+00$	$2.364715012894220E+00$	$-1.127338722408533E-02$

tau = 1.E-01

$5.E-11$	$2.370420869696147E+00$	$2.370420869720090E+00$	$2.394262565985628E-01$
$5.E-10$	$2.370420869588361E+00$	$2.370420869827943E+00$	$2.395825760004300E-01$
$5.E-09$	$2.370420868510474E+00$	$2.370420870905823E+00$	$2.395348808192921E-01$
$5.E-08$	$2.370420857731419E+00$	$2.370420881684882E+00$	$2.395346321293346E-01$
$5.E-07$	$2.370420749940862E+00$	$2.370420989475625E+00$	$2.395347622474731E-01$
$5.E-06$	$2.370419672044952E+00$	$2.370422067392266E+00$	$2.395347314276819E-01$
$5.E-05$	$2.370408894019773E+00$	$2.370432847490506E+00$	$2.395347073314014E-01$
$5.E-04$	$2.370301208367185E+00$	$2.370540740456197E+00$	$2.395320890116182E-01$
$5.E-03$	$2.369235027204600E+00$	$2.371600105689946E+00$	$2.365078485345951E-01$
$5.E-02$	$2.361254891567542E+00$	$2.382502692203918E+00$	$2.124780063637610E-01$

puisvol = 5.93589E+08

$5.E-02$	$2.370420869287190E+00$	$2.370420870129101E+00$	$8.419118735503162E-09$
$5.E-01$	$2.370420865498692E+00$	$2.370420873917602E+00$	$8.418910013574532E-09$
$5.E+00$	$2.370420827613255E+00$	$2.370420911802979E+00$	$8.418972363699594E-09$
$5.E+01$	$2.370420448759516E+00$	$2.370421290656854E+00$	$8.418973385104777E-09$
$5.E+02$	$2.370416660222961E+00$	$2.370425079196105E+00$	$8.418973143964337E-09$
$5.E+03$	$2.370378774981966E+00$	$2.370462964713576E+00$	$8.418973160972954E-09$
$5.E+04$	$2.369999935014275E+00$	$2.370841832330115E+00$	$8.418973158406118E-09$
$5.E+05$	$2.366212779758054E+00$	$2.374631752461517E+00$	$8.418972703462924E-09$
$5.E+06$	$2.328465868945608E+00$	$2.412341135123118E+00$	$8.387526617750929E-09$
$5.E+07$	$1.964509895667832E+00$	$2.802187126391661E+00$	$8.376772307238287E-09$

urentre = 0.E+00

$5.E-05$	$2.370420869708151E+00$	$2.370420869708145E+00$	$-6.217248937900877E-11$
$5.E-04$	$2.370420869708145E+00$	$2.370420869708144E+00$	$-8.881784197001252E-13$
$5.E-03$	$2.370420869708152E+00$	$2.370420869708147E+00$	$-4.884981308350689E-13$
$5.E-02$	$2.370420869708182E+00$	$2.370420869707872E+00$	$-3.099742684753437E-12$
$5.E-01$	$2.370420869707973E+00$	$2.370420869708355E+00$	$3.819167204710538E-13$
$5.E+00$	$2.370420869709116E+00$	$2.370420869708819E+00$	$-2.970956813896919E-14$
$5.E+01$	$2.370420869710200E+00$	$2.370420863221630E+00$	$-6.488570125640080E-11$

Suite à l'examen de ces tableaux, il est raisonnable de penser que les chiffres significatifs corrects de la dérivée de `ur(306,23)` par rapport à `cd`, `qsi`, `tau`, `puisvol` et `urentre` sont ceux donnés dans la troisième colonne du tableau 5.2. Dans la seconde colonne de celui-ci, on trouve le pas t_{opt} donnant les meilleures dérivées par différences finies. On peut comparer ces dérivées à celles obtenues par ODYSSÉE. Dans la quatrième colonne du tableau 5.2 sont donnés les chiffres communs aux deux dérivées d'ODYSSÉE (voir tableau 5.1). On peut alors en déduire le nombre maximal de chiffres significatifs que l'on peut espérer gagner avec ODYSSÉE. Il est donné en dernière colonne du tableau 5.2. Nous verrons, avec les tests passés en simple précision, que le gain réel est moins important que ce 'gain maximal'.

	différences finies		ODYSSÉE	
	t_{opt}	dérivée	dérivée	gain maximal
`cd`	$5.E-06$	$-2.6877707D+00$	$-2.68777074766D+00$	$+4$
`qsi`	$5.E-04$	$-1.1283833D-02$	$-1.1283833581\ D-02$	$+3$
`tau`	$5.E-06$	$2.395347\ D-01$	$2.3953473\ D-01$	$+1$
`puisvol`	$5.E+03$	$8.4189731D-09$	$8.4189731\ D-09$	$+0$
`urentre`	$5.E-01$			

TAB. 5.2 – *Comparaison des dérivées en double précision*

Dans les figures 5.2, 5.4, 5.6, 5.8 et 5.10, nous avons comparé les valeurs des dérivées obtenues par différences finies pour le pas "optimal" t_{opt} calculé ci-dessus avec celles obtenues par les codes générés par ODYSSÉE. Les courbes donnent les écarts relatifs : celle en pointillés larges compare le linéaire tangent aux différences finies et celle en pointillés courts compare le linéaire cotangent aux différences finies. Dans ces figures, la courbe la plus basse repère le couple des méthodes les plus concordantes. On observe que ce couple est en général formé des dérivées d'ODYSSÉE. Si cela n'implique pas nécessairement que le différentiateur calcule les dérivées les plus précises, on peut toutefois penser que cela donne du crédit à ses dérivées.

La même batterie de tests à été passée en simple précision. Comme en double précision, intéressons-nous aux dérivées de `uridx` (composante 23 de `ur` au 306-ième pas de temps). Les résultats obtenus avec ODYSSÉE sont donnés au tableau 5.3. Comme en double précision, nous avons choisi le pas optimal pour les différences finies et comparé les résultats obtenus par ODYSSÉE à ceux obtenus par différences finies (on trouvera le détail des résultats en annexe E). On obtient alors le tableau 5.4 que l'on comparera au tableau 5.2. Si on fait confiance aux premiers chiffres significatifs obtenus en double précision, on voit qu'avec la méthode utilisée pour déterminer les

	linéaire tangent	linéaire cotangent
`cd`	$-2.68763E+00$	$-2.68763E+00$
`qsi`	$-1.12781E-02$	$-1.12782E-02$
`tau`	$2.39515E-01$	$2.39520E-01$
`puisvol`	$8.41967E-09$	$8.41967E-09$
`urentre`	$-3.94494E-08$	$-3.92737E-08$

Tab. 5.3 – *Dérivées de* `ur(306,23)` *par* Odyssée *(simple précision)*

	différences finies		Odyssée	
	t_{opt}	dérivée	dérivée	gain réel
`cd`	$5.E-03$	$-2.68E+00$	$-2.68763E+00$	+1
`qsi`	$5.E-04$	$-1.19E-02$	$-1.1278\ E-02$	+1
`tau`	$5.E-03$	$2.\ \ E-01$	$2.395\ \ E-01$	+3
`puisvol`	$5.E+06$	$8.3\ E-09$	$8.41967E-09$	+2
`urentre`				

Tab. 5.4 – *Comparaison des dérivées en double précision*

dérivées par différences finies (voir annexe A.5), seul le dernier chiffre significatif peut ne pas être correct. On voit également que les chiffres significatifs communs aux dérivées obtenues en modes direct et inverse ne sont pas tous corrects : on peut avoir jusqu'à 2 chiffres erronés. Cependant, la dernière colonne du tableau montre qu'Odyssée trouve entre 1 et 3 chiffres corrects en plus que le maximum que l'on peut avoir par différences finies. Rappelons que les dérivées générées par Odyssée appellent des unités calculant des différences finies. Cette conclusion est donc spécifique au code Thyc-1D, dans le sens où il est vraisemblable que les résultats d'Odyssée auraient pu être meilleurs, si le code-source des fonctions tabulées avait été disponible.

5.3 Efficacité en temps et place-mémoire

Afin de comparer les performances des codes générés (linéaire tangent et cotangent), nous avons calculé les dérivées par rapport à 1 (`cd`), 2 (`cd`, `qsi`), 3 (`cd`, `qsi`, `tau`), ..., jusqu'à 5 variables d'entrée. Les valeurs sont données dans le tableau 5.5. La ligne avec le chiffre 0 correspond au code original. La colonne CPU donne le temps CPU en secondes, `text` donne la longueur en octets des codes générés par Odyssée (après compilation) et `bss` donne la taille en octets des données non initialisées.

On peut sans doute extrapoler les valeurs du temps CPU données dans ce tableau pour obtenir les temps de calcul pour plus de cinq variables d'entrée. Si n est le nombre de variables d'entrée,

	linéaire tangent			linéaire cotangent		
	CPU	`text`	`bss`	CPU	`text`	`bss`
0	5.72	270336	1056080	5.93	270336	1056080
1	14.08	417792	266680	27.61	573440	2540712
2	26.91	417792	266712	27.30	573440	2540912
3	38.25	417792	267592	27.20	581632	2540544
4	55.21	417792	267616	27.31	581632	2540608
5	64.16	425984	267832	27.32	581632	2541256

TAB. 5.5 – *Temps et place-mémoire*

on obtient en moyenne pour le linéaire tangent

$$\frac{T(f,f')}{T(f)} \approx 2.3\,n,$$

où $T(f,f')$ est le temps d'exécution du linéaire tangent et $T(f)$ est celui du code original. Pour le linéaire cotangent le temps d'exécution $T(f,\nabla f)$ relatif à celui du code original est indépendant de n et vaut

$$\frac{T(f,\nabla f)}{T(f)} \approx 4.6.$$

Ces valeurs sont en conformité avec les valeurs théoriques données en section 1. On observe que pour plus de 3 variables d'entrée, c'est le code linéaire cotangent qui devient le plus rapide.

En ce qui concerne la taille du code (`text`), on constate que le linéaire tangent demande environ 35% de mémoire en plus que le code original, alors que le linéaire cotangent en demande plus du double.

Enfin, le nombre de variables non initialisées (`bss`) double approximativement en linéaire tangent (ce qui est conforme aux estimations théoriques) et est multiplié par 25 en linéaire cotangent. On rappelle qu'une seule stratégie de gestion-mémoire a été essayée pour le linéaire cotangent. Ce chiffre est donc surtout donné comme référence à battre pour les études comparatives futures.

Chapitre 6

Conclusion

La génération automatique de codes linéaire tangent et linéaire cotangent à partir d'un code industriel est une tâche difficile. On peut considérer qu'ODYSSÉE a bien surmonté la difficulté puisque le différentiateur a réussi à génerer le code calculant les dérivées que l'on s'était fixées au départ de l'étude. Une contribution importante de cette étude, qui a permis d'arriver à ce résultat, est l'introduction dans ODYSSÉE d'une analyse interprocédurale permettant de détecter les dépendances entre variables actives à travers l'ensemble des sous-routines du code.

Nous pensons que le passage au code THYC (3D) est à présent envisageable. Deux éléments nous confortent dans cette opinion. D'une part, ODYSSÉE a déjà pu traiter de très grands codes, tel que celui utilisé en météorologie pour l'assimilation variationnelle des données [18, 15]. L'utilisation d'ODYSSÉE sur celui-ci se faisait routine par routine avec des conventions d'écriture qui permettaient de rendre consistant l'ensemble du code généré. D'autre part, le succès d'ODYSSÉE sur THYC-1D a permis de montrer que l'on pouvait passer à un traitement du code dans son ensemble, en se libérant des conventions d'écriture précédemment nécessaires. En conséquence, nous pensons que cette étude a montré que les bases sur lesquelles ODYSSÉE est fondé sont suffisamment souples et riches pour permettre au différentiateur de s'attaquer à un code industriel de la taille et de la complexité de THYC.

On peut tirer d'autres enseignements de cette étude. En double précision, on peut probablement avoir entre 1 et 3 chiffres significatifs corrects en plus par ODYSSÉE que par des différences finies utilisant la méthode multi-pas coûteuse, cette dernière donnant entre 7 et 8 chiffres significatifs corrects en double précision. Ce qui a sans aucun doute limité les performances d'ODYSSÉE est la nécessité d'inclure dans le code généré des routines approchant la jacobienne de fonctions dont le code-source n'était pas disponible. D'autre part, les performances d'ODYSSÉE en temps sont conformes à la théorie : le calcul d'une dérivée directionnelle se fait en un temps égal à celui de 2.3 appels de fonction et le calcul d'un gradient en un temps équivalent à 4.9 appels de fonction. L'encombrement-mémoire en mode inverse est resté suffisamment raisonnable pour permettre le traitement du code dérivé de THYC-1D sur une station de travail.

Annexe A

Evaluation de dérivées par différences finies

Dans cette annexe, nous précisons quelques concepts de base relatifs au calcul des dérivées par différences finies.

On rappelle que la *précision-machine* d'un calculateur donné est définie comme le plus petit nombre réel $\epsilon_M > 0$ tel que sur ce calculateur $1 + \epsilon_M \neq 1$. Cela dépend évidemment du nombre de bits utilisés pour représenter les nombres. La table A.1 résume les précisions-machine approchées (on s'est restreint aux puissances de 10) sur une SPARCstation 20 de SUN en fonction du type Fortran des variables. On note alors

type de variable	précision-machine ϵ_M
`real*4` (simple précision)	10^{-7}
`real*8` (double précision)	10^{-15}
`real*16` (quadruple précision)	10^{-34}

TAB. A.1 – *Précisions-machine ϵ_M sur une SPARCstation 20 de SUN*

$$n_M = -\log_{10} \epsilon_M,$$

le nombre maximal de chiffres significatifs corrects.

A.2 Principes

Supposons que l'on connaisse $x \in \mathbb{R}^n$ et $f(x) \in \mathbb{R}^m$ et que l'on cherche à calculer par différences finies, une approximation de la dérivée directionnelle $f'(x; d) \in \mathbb{R}^m$, pour $d \in \mathbb{R}^n$ donné. Rappelons que celle-ci est définie par la limite

$$f'(x; d) = \lim_{t \to 0} \frac{f(x + td) - f(x)}{t}. \tag{A.1}$$

On parle d'approximation par *différences décentrées* lorsque $f'(x;d)$ est approché par le quotient différentiel

$$\frac{f(x+td)-f(x)}{t}, \tag{A.2}$$

pour un *pas* t "bien choisi". Suivant que t est positif ou négatif, on parle de différences finies décentrées *à droite* ou *à gauche*. On dit qu'il s'agit de différences finies *centrées* si on utilise le quotient différentiel

$$\frac{f(x+td)-f(x-td)}{2t}. \tag{A.3}$$

Pour des fonctions régulières, la formule (A.2) est correcte à un $O(t)$ près et la formule (A.3) à un $O(t^2)$ près. Cette dernière formule est donc préférable mais demande deux évaluations de fonctions supplémentaires plutôt qu'une.

La difficulté du calcul des dérivées approchées par différences finies réside dans le choix du pas t. Pour t trop grand, le résultat est imprécis et pour t trop petit les erreurs d'arrondi l'emportent dans le calcul. De ce point de vue, la formule (A.3) est meilleure, car elle autorise le choix d'un t plus grand, s'éloignant ainsi de la zone des pas où les erreurs d'arrondi deviennent prédominantes. Nous reviendrons sur ce sujet en section A.5

A.3 Utilisation d'un seul pas

Que l'on choisisse les formules (A.2) ou (A.3), si on ne s'autorise l'essai que d'un seul pas t, il est en général recommandé de s'y prendre comme suit (voir par exemple [3]). Puisque l'on ne peut pas corriger la valeur de t en regardant comment varie $f(x+td)$, on choisit de déterminer le pas en contrôlant la variation de x dans la direction d.

Si $d=0$, la dérivée directionnelle est nulle et il n'y a pas lieu de choisir un pas.

Supposons à présent que $d \neq 0$ et $x \neq 0$. On note $\|\cdot\|$ la norme ℓ_2 (euclidienne). On choisit alors t de telle sorte que l'accroissement relatif $(td)/\|x\|$ donné à x soit en norme de l'ordre de la racine carrée de ϵ_M, ce qui donne

$$t = \sqrt{\epsilon_M}\frac{\|x\|}{\|d\|} \qquad (d \neq 0, x \neq 0). \tag{A.4}$$

Ceci revient à peu près à modifier le $(n_M/2)$-ième chiffre significatif de x qui en a au plus n_M corrects.

Dans certains cas, il est nécessaire d'améliorer cette formule. Il arrive en effet que certaines composantes de d soit nulles ou très petites par rapport à d'autres composantes. Dans ce cas, il n'est pas souhaitable de faire intervenir les composantes correspondantes de x dans le calcul du pas. Par exemple, si d n'a que sa composante j non nulle, on cherche à calculer la dérivée partielle de f par rapport à la j-ième variable et il n'y a pas lieu de faire intervenir d'autres composantes de x que x_j dans le calcul de t. Compte tenu de cette observation, on comprend qu'il sera souvent préférable de choisir le pas suivant

$$t = \frac{\sqrt{\epsilon_M}}{\|d\|}\left(\sum_{i:|d_i|\geq \epsilon_M\|d\|_\infty} x_i^2\right)^{\frac{1}{2}} \qquad (d \neq 0, x \neq 0), \tag{A.5}$$

où $\|d\|_\infty = \max_{1\le i\le n} |d_i|$.

Lorsque $d \neq 0$ et $x = 0$, il y a peu d'information pour déterminer t. Nous avons alors pris $t = 1$.

A.4 Utilisation de deux pas

Supposons à présent que l'on s'autorise un deuxième essai avec un pas t_2, un premier essai ayant été fait avec un pas t_1 (par exemple avec la méthode proposée en section A.3). On dispose alors d'une estimation de la dérivée directionnelle, à savoir

$$f_1' = \frac{f(x+t_1d)-f(x)}{t_1}.$$

La situation est la même lorsque l'on dispose déjà d'une estimation de la dérivée et que l'on veut en vérifier la qualité par différences finies.

On peut alors chercher le pas t_2 tel que la variation relative de f pour ce pas soit à peu près $\sqrt{\epsilon_M}$. En approchant $f(x+td)$ par $f(x)+tf_1'$, on trouve

$$t_2 = \sqrt{\epsilon_M}\frac{\|f(x)\|}{\|f_1'\|}. \tag{A.6}$$

Cette estimation du pas est donc destinée à modifier à peu près le $(n_M/2)$-ième chiffre significatif de $f(x)$ qui en a au plus n_M corrects. On se base donc ici sur la variation de f plutôt que sur celle de x pour déterminer le pas.

A.5 Utilisation d'un nombre quelconque de pas

Supposons à présent que l'on cherche à obtenir le plus grand nombre possible de chiffres significatifs corrects par différences finies en s'autorisant un nombre quelconque d'essais de pas.

La formule (A.1) nous dit qu'en “arithmétique exacte”, plus t est petit, plus grand est le nombre de chiffres significatifs corrects dans les quotients différentiels (A.2) ou (A.3). Nous avons représenté ce fait par la courbe en trait continu à la figure A.1. Notons que ce nombre ne croît pas nécessairement de façon monotone comme sur cette figure. D'autre part, du fait de la précision finie de la machine, si t est plus petit qu'un certain seuil (t_b à la figure A.1), les quotients différentiels (A.2) ou (A.3) ne donnent aucun chiffre significatif correct. Lorsque t augmente ce nombre peut augmenter. D'autre part, encore du fait de la précision finie de la machine, ce nombre est limité par n_M. On peut donc tracer une courbe, celle en trait discontinu de la figure A.1, joignant les points $(0, t_b)$ et $(n_M, 0)$, à droite de laquelle les chiffres sont bruités. En combinant ces deux points de vue, on obtient la zone grise de la figure A.1 dans laquelle les chiffres sont stables et égaux à ceux de la dérivée directionnelle.

Comme la zone grise de la figure A.1 est délimitée par deux courbes, son étendue va décroître dans les cas suivants. D'une part, si la fonction $t \mapsto f(x+td)$ ne se stabilise pas rapidement autour de sa partie linéaire, le point t_a sera plus négatif et la courbe en trait continu de la figure A.1 sera plus basse. D'autre part, si la précision-machine ϵ_M diminue le seuil t_b augmente ainsi que la limite droite de la courbe en trait discontinu de la figure A.1. Celle-ci se déplace donc vers le haut et vers la gauche, réduisant ainsi la zone grise. Enfin, en prenant la formule (A.2) au lieu de (A.3), on

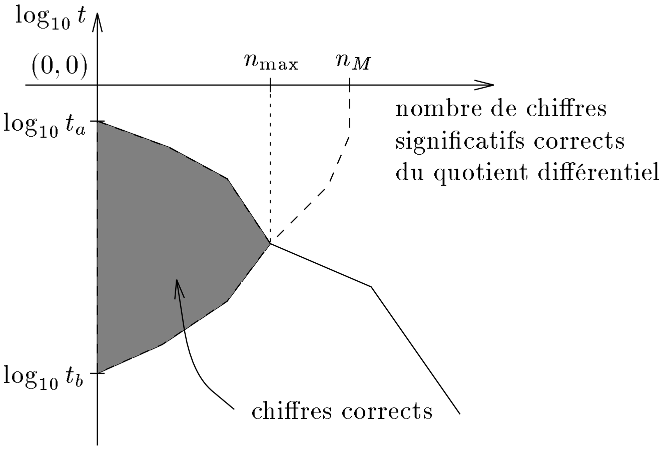

FIG. A.1 – *Précision des différences finies.*

réduit également la zone grise, surtout parce que la courbe en trait continu descend. Dans certains cas, cette zone peut ne pas exister.

Ces observations permettent de définir une méthode pour estimer le nombre maximal $n_{\max}$ de chiffres significatifs corrects que l'on peut obtenir par différences finies et la valeur de ces chiffres. Il suffit d'évaluer les quotients différentiels (A.2) ou (A.3) pour des valeurs décroissantes de t (par un facteur de 10^{-1} par exemple) et de repérer la zone où les chiffres les plus significatifs se stabilisent. Cette zone correspond à la zone grise de la figure A.1. On obtient alors le nombre $n_{\max}$ comme sur cette figure.

Annexe B

Codes linéaire tangent et cotangent de la sous-routine `tba2ps`

B.1 Code linéaire tangent (différences finies décentrées avec un seul pas)

```
      subroutine tba2pstl (i,p,sv,r1,r2,x,pp,svp,xp)
c
      integer i
      real p,sv,r1,r2,x
      real pp,xp

      real myeps
      common /ceps/ myeps
      real tba2ps

      real a
c
c --- calcul de la fonction
c
      x=tba2ps(i,p,sv,r1,r2)
c
c --- cas d'une DD nulle
c
      if (pp.eq.0. .and. svp.eq.0.) then
          xp=0.
          return
      endif
c
c --- calcul de la DD
c
      a=p*p+sv*sv
      if (a.eq.0.) a=1.
      a= abs(sqrt(myeps*a/(pp*pp+svp*svp)))
c
      xp=(tba2ps(i,p+a*pp,sv+a*svp,r1,r2)-x)/a
c
      return
      end
```

B.2 Code linéaire tangent (différences finies centrées avec deux pas)

```
      subroutine tba2ps2tl (i,p,s,r1,r2,x,pp,sp,xp)
c
      integer i
      real p,s,r1,r2,x
      real pp,sp,xp
c
c----
c
c     Code tangent de tba2ps (par differences finies).
c     Hypothese: entrees: p,s
c                sortie: x
c
c     Differences centrees, avec eventuellement 2 essais de pas
c
c----
c
c --- variables globales
c
      real myeps
      common /ceps/ myeps
      real tba2ps
c
c --- variables locales
c
      real t,dx,dmax,xplus,xmoins
c
c --- tester myeps
c
      if (myeps.eq.0.) then
          write (6,900)
          stop
      endif
  900 format (">>> ERREUR-tba2ps2tl: myeps=0, appeler meps() d'abord"
c
c --- calcul de la fonction
c
      x=tba2ps(i,p,s,r1,r2)
c
c --- cas d'une DF nulle
c
      if (pp.eq.0. .and. sp.eq.0.) then
          xp=0.
          return
      endif
c
c --- calcul des DF
c
c     --- calcul du premier pas
c
      dmax=max(abs(pp),abs(sp))*myeps
      t=0.
      if (abs(pp).ge.dmax) t=p*p
      if (abs(sp).ge.dmax) t=t+s*s
      if (t.eq.0.) t=1.
      t=abs(sqrt(myeps*t/(pp*pp+sp*sp)))
c
c     --- evaluation de la fonction au point perturbe +
c
```

```
      xplus=tba2ps(i,p+t*pp,s+t*sp,r1,r2)
c
c     --- calcul du second pas
c
      dx=abs(xplus-x)
      if (dx.ne.0) then
c
c         --- xp utilise comme variable temporaire
c
          xp=sqrt(myeps)*abs(x)/dx
          if (xp.lt.0.1 .or. xp.gt.10.) then
c
c         --- le bon pas differe de t par un facteur plus grand que 10
c             donc on modifie t et on recalcule xplus
c
              t=t*xp
              xplus=tba2ps(i,p+t*pp,s+t*sp,r1,r2)
          endif
      end
c
c     --- evaluation de la fonction au point perturbe -
c
      xmoins=tba2ps(i,p-t*pp,s-t*sp,r1,r2)
c
c     --- calcul des DF
c
      xp=(xplus-xmoins)/(2.*t)
c
      return
      end
```

B.3 Code linéaire cotangent

```
      subroutine tba2psad (i,p,s,r1,r2,xx,pa,sa,xa)
      integer i
      real p,s,r1,r2,x
      real pa,sa,xa

      real myeps
      common /ceps/ myeps
      real tba2ps

      real t
c
c --- calcul de la fonction
c
      x=tba2ps(i,p,s,r1,r2)
c
c --- calcul de la jacobienne
c
      t=p
      if (t.eq.0.) t=1.
      t=abs(sqrt(myeps)*t)
      x_p=(tba2ps(i,p+t,s,r1,r2)-x)/t
c
      t=s
      if (t.eq.0.) t=1.
      t=abs(sqrt(myeps)*t)
      x_s=(tba2ps(i,p,s+t,r1,r2)-x)/t
c
c --- calcul des valeurs adjointes
```

```
c
      pa=pa+x_p*xa
      sa=sa+x_s*xa
      xa=0
c
      return
      end
```

Annexe C

Codes linéaire tangent et cotangent de la sous-routine `tdma`

C.1 Code linéaire tangent

```
      subroutine tdmatl (c,a,d,b,x,imax,cp,ap,dp,bp,xp)
c
      integer imax
      real a(*),b(*),c(*),d(*),x(*)
      real ap(*),bp(*),cp(*),dp(*),xp(*)

      integer ndim
      parameter (ndim=100)

      integer i
      real aa(ndim),ba(ndim),ca(ndim),aux(ndim)
c
c --- solution x du SL
c
      call tdma (ca,aa,d,ba,x,imax)
c
c --- second membre aux() de l'equation derivee
c
      aux(1)=dp(1)-bp(1)*x(1)+cp(1)*x(2)
      if (imax.gt.2) then
          do i=2,imax-1
              aux(i)=dp(i)+ap(i)*x(i-1)-bp(i)*x(i)+cp(i)*x(i+1)
          enddo
      endif
      if (imax.gt.1)
     &    aux(imax)=dp(imax)+ap(imax)*x(imax-1)-bp(imax)*x(imax)
c
c --- recopie de a, b, c dans aa, ba, ca
c
      do i=1,imax
          aa(i)=a(i)
          ba(i)=b(i)
          ca(i)=c(i)
      enddo
c
c --- solution derivee xp
c
      call tdma (ca,aa,aux,ba,xp,imax)
```

```
c
      return
      end
```

C.2 Code linéaire cotangent

```
      subroutine tdmaad (c,a,d,b,x,imax,ca,aa,da,ba,xa)
c
      integer imax
      real a(*),b(*),c(*),x(*),d(*)
      real aa(*),ba(*),ca(*),da(*),xa(*)

      integer ndim
      parameter (ndim=100)
c
      integer i
      real v(ndim)
c
c --- solution v du SL adjoint
c
c     --- decaler a et c
c
      do i=1,imax-1
          a(i)=a(i+1)
      enddo
      a(imax)=0.
      do i=imax,2,-1
          c(i)=c(i-1)
      enddo
      c(1)=0.
c
c     --- resoudre le systeme lineaire adjoint
c
      call tdma (a,c,xa,b,v,imax)
c
c     --- restituer a et c
c
      do i=imax,2,-1
          a(i)=a(i-1)
      enddo
      a(1)=0.
      do i=1,imax-1
          c(i)=c(i+1)
      enddo
      c(imax)=0.
c
c --- mise a jour des variables adjointes
c
      do i=2,imax
          aa(i)=aa(i)+v(i)*x(i-1)
      enddo
      do i=1,imax
          ba(i)=ba(i)-v(i)*x(i)
          da(i)=da(i)+v(i)
          xa(i)=0.
      enddo
      do i=1,imax-1
          ca(i)=ca(i)+v(i)*x(i+1)
      enddo
c
      return
```

```
end
```

Annexe D

Codes linéaire tangent et cotangent de la sous-routine `solvtit1`

D.1 Code linéaire tangent

```
COD Unite : solvtit1tl
COD Derivee de l'unite : solvtit1
COD IN  globals: tau uv ro titn sigma a tit1 b c qn d
COD OUT globals: dtit1 d c b tit1 a
COD Dependances entre IN et OUT
COD dtit1 <--  tau uv qn ro b titn sigma d a c
COD d <--  titn tau uv ro sigma qn d
COD c <--  tau uv ro c
COD b <--  tau uv qn ro b
COD tit1 <--  titn tau uv qn ro b sigma d a c tit1
COD a <--  tau uv ro a


      SUBROUTINE solvtit1tl ()

      PARAMETER (ndim = 100)
      IMPLICIT REAL (a-h, o-z)
      REAL sr01s, sr02s, sr03s, sr04s
      COMMON /PRESS1/p(ndim), pn(ndim), p1(ndim), p2(ndim)
      COMMON /DEBIT1/q(ndim), qn(ndim), q1(ndim), q2(ndim)
      COMMON /VITRE1/ur(ndim), urn(ndim), ur1(ndim), ur2(ndim)
      COMMON /TITRE1/tit(ndim), titn(ndim), tit1(ndim), tit2(ndim)
      COMMON /PRESS2/dp1(ndim), dp2(ndim), dp3(ndim)
      COMMON /DEBIT2/dq1(ndim), dq2(ndim), dq3(ndim)
      COMMON /VITRE2/dur1(ndim), dur2(ndim), dur3(ndim), dtit(ndim),
     : dtit1(ndim), dtit2(ndim)
      COMMON /ENTRO1/s(ndim), sn(ndim)
      COMMON /ENTRO2/ds(ndim)
      COMMON /VARPR1/cson(ndim), vp(ndim), ga(ndim)
      COMMON /VARPR2/ro(ndim), rol(ndim), rov(ndim), unsrovl(ndim)
      COMMON /VARPR3/diago(ndim), rotit(ndim), titit(ndim)
      COMMON /VARPR4/h(ndim), chl(ndim), t(ndim), tith(ndim)
      COMMON /VARPR5/alliq(ndim), cpliq(ndim)
      COMMON /VARPR6/rdp(ndim), rdc(ndim), rds(ndim)
      COMMON /VARPR7/slsat(ndim), svsat(ndim)
      COMMON /VARPR8/dslsdp(ndim), dsvsdp(ndim)
      COMMON /VARPR9/sigma(ndim)
      COMMON /VARDE1/u(ndim), v(ndim), uv(ndim), ul(ndim)
```

```
      COMMON /VARDE2/alpha1(ndim), alpha2(ndim)
      COMMON /VARDE3/beta1(ndim), beta2(ndim)
      COMMON /VARDE4/tit_u(ndim), ro_u(ndim), unsrovl_u(ndim)
      COMMON /VARDE5/rotit_u(ndim), romvl_u(ndim), sigma_u(ndim)
      COMMON /AMAILL/x(ndim)
      COMMON /RESOUD/a(ndim), b(ndim), c(ndim), d(ndim)
      COMMON /GEO1/n
      COMMON /GEO2/dx, dt, dtdx, xlong
      COMMON /PARA1/alamv, alaml, cd, temp, r, grav
      COMMON /DATE/temps, ttot
      COMMON /CONDI/psortie, qentree, uentree, urentre, titentr,
     : sentree
      COMMON /PUIS/puisvol, qvol(ndim)
      COMMON /PARA2/tau, qsi
      COMMON /DEBIT3/qt(ndim)
      COMMON /FICH01/nficur, nficp, nficq, nficrov, nfictit
      COMMON /FICH02/nficro, nficu, nficuns, nficson
      COMMON /FICH03/nficuv, nficul, nfics
      COMMON /FICH04/nficrp, nficrs, nficrc
      COMMON /IMPRES/imprime, ipasimp, idessin
      COMMON /FICH00/nficdon, nficres, motcle
      COMMON /COUR01/cnu, cnul, cnuv, cnur, cnumax, cnulmax, cnuvmax,
     : cnurmax, cnmax
      COMMON /STAT01/pmax, pmin, pmoy, pmaxt, pmint
      COMMON /STAT02/urmax, urmin, urmoy, urmaxt, urmint
      COMMON /STAT03/qmax, qmin, qmoy, qmaxt, qmint
      COMMON /STAT04/timax, timin, timoy, timaxt, timint
      COMMON /STAT05/smax, smin, smoy, smaxt, smint
      COMMON /STAT06/dtmin, dtmax, dtmoy
      COMMON /DEBIT1TL/qtl(ndim), qntl(ndim), q1tl(ndim), q2tl(ndim)
      COMMON /TITRE1TL/tittl(ndim), titntl(ndim), tit1tl(ndim),
     : tit2tl(ndim)
      COMMON /VITRE2TL/dur1tl(ndim), dur2tl(ndim), dur3tl(ndim),
     : dtittl(ndim), dtit1tl(ndim), dtit2tl(ndim)
      COMMON /VARPR2TL/rotl(ndim), roltl(ndim), rovtl(ndim),
     : unsrovltl(ndim)
      COMMON /VARPR9TL/sigmatl(ndim)
      COMMON /VARDE1TL/utl(ndim), vtl(ndim), uvtl(ndim), ultl(ndim)
      COMMON /RESOUDTL/atl(ndim), btl(ndim), ctl(ndim), dtl(ndim)
      COMMON /PARA2TL/tautl, qsitl
      REAL p, pn, p1, p2
      REAL q, qn, q1, q2
      REAL ur, urn, ur1, ur2
      REAL tit, titn, tit1, tit2
      REAL dp1, dp2, dp3
      REAL dq1, dq2, dq3
      REAL dur1, dur2, dur3, dtit, dtit1, dtit2
      REAL s, sn
      REAL ds
      REAL cson, vp, ga
      REAL ro, rol, rov, unsrovl
      REAL diago, rotit, titit
      REAL h, chl, t, tith
      REAL alliq, cpliq
      REAL rdp, rdc, rds
      REAL slsat, svsat
      REAL dslsdp, dsvsdp
      REAL sigma
      REAL u, v, uv, ul
      REAL alpha1, alpha2
      REAL beta1, beta2
      REAL tit_u, ro_u, unsrovl_u
```

```
      REAL rotit_u, romvl_u, sigma_u
      REAL x
      REAL a, b, c, d
      INTEGER n
      REAL dx, dt, dtdx, xlong
      REAL alamv, alaml, cd, temp, r, grav
      REAL temps, ttot
      REAL psortie, qentree, uentree, urentre, titentr, sentree
      REAL puisvol, qvol
      REAL tau, qsi
      REAL qt
      INTEGER nficur, nficp, nficq, nficrov, nfictit
      INTEGER nficro, nficu, nficuns, nficson
      INTEGER nficuv, nficul, nfics
      INTEGER nficrp, nficrs, nficrc
      INTEGER imprime, ipasimp, idessin
      INTEGER nficdon, nficres
      CHARACTER*(1) motcle*40
      REAL cnu, cnul, cnuv, cnur, cnumax, cnulmax, cnuvmax, cnurmax,
     : cnmax
      REAL pmax, pmin, pmoy, pmaxt, pmint
      REAL urmax, urmin, urmoy, urmaxt, urmint
      REAL qmax, qmin, qmoy, qmaxt, qmint
      REAL timax, timin, timoy, timaxt, timint
      REAL smax, smin, smoy, smaxt, smint
      REAL dtmin, dtmax, dtmoy
      REAL qtl, qntl, q1tl, q2tl
      REAL tittl, titntl, tit1tl, tit2tl
      REAL dur1tl, dur2tl, dur3tl, dtittl, dtit1tl, dtit2tl
      REAL rotl, roltl, rovtl, unsrovltl
      REAL sigmatl
      REAL utl, vtl, uvtl, ultl
      REAL atl, btl, ctl, dtl
      REAL tautl, qsitl
      REAL flurmtl
      REAL sr01stl
      REAL sr04stl
      REAL sr02stl
      REAL flurptl
      REAL sr03stl
      EXTERNAL jacobitl

      DO i = 2, n-1
         sr01s = dx/dt
         IF (0..GE.uv(i-1)) THEN
           sr02s = 0.
           sr02stl = 0.
         ELSE
           sr02stl = uvtl(i-1)
           sr02s = uv(i-1)
         END IF
         atl(i) =
     :     ((((-rotl(i-1)*sr02s)-ro(i-1)*sr02stl)*ro(i)+(sr02s*ro(i-1))*
     :    rotl(i))/ro(i)**2)*((-1.)/(sr01s+dx/tau))-((-sr02s*ro(i-1))/ro
     :    (i))*((-dx*((-tautl)/tau**2))/(sr01s+dx/tau)**2)
         a(i) = ((-1.)/(sr01s+dx/tau))*((-sr02s*ro(i-1))/ro(i))
         sr01s = dx/dt
         IF (0..GE.uv(i)) THEN
           sr02s = 0.
           sr02stl = 0.
         ELSE
           sr02stl = uvtl(i)
```

```
        sr02s = uv(i)
      END IF
      IF (0..LE.uv(i-1)) THEN
        sr03s = 0.
        sr03stl = 0.
      ELSE
        sr03stl = uvtl(i-1)
        sr03s = uv(i-1)
      END IF
      btl(i) =
   :   (sr02stl-sr03stl+(-((qntl(i)-qntl(i-1))*ro(i)-(qn(i)-qn(i-1))
   :  *rotl(i))/ro(i)**2))*(1./(sr01s+dx/tau))+(sr02s-sr03s+(-(qn(i)
   :  -qn(i-1))/ro(i)))*((-dx*((-tautl)/tau**2))/(sr01s+dx/tau)**2)
      b(i) =
   :   1.+(1./(sr01s+dx/tau))*(sr02s-sr03s+(-(qn(i)-qn(i-1))/ro(i)))
      sr01s = dx/dt
      IF (0..LE.uv(i)) THEN
        sr02s = 0.
        sr02stl = 0.
      ELSE
        sr02stl = uvtl(i)
        sr02s = uv(i)
      END IF
      ctl(i) =
   :   (((rotl(i+1)*sr02s+ro(i+1)*sr02stl)*ro(i)-(sr02s*ro(i+1))*rot
   :  l(i))/ro(i)**2)*((-1.)/(sr01s+dx/tau))-((sr02s*ro(i+1))/ro(i))
   :   *((-dx*((-tautl)/tau**2))/(sr01s+dx/tau)**2)
      c(i) = ((-1.)/(sr01s+dx/tau))*((sr02s*ro(i+1))/ro(i))
      IF (0..GE.uv(i)) THEN
        sr01s = 0.
        sr01stl = 0.
      ELSE
        sr01stl = uvtl(i)
        sr01s = uv(i)
      END IF
      sr02stl = rotl(i)*sr01s+ro(i)*sr01stl
      sr02s = sr01s*ro(i)
      IF (0..LE.uv(i)) THEN
        sr03s = 0.
        sr03stl = 0.
      ELSE
        sr03stl = uvtl(i)
        sr03s = uv(i)
      END IF
      sr04stl = rotl(i+1)*sr03s+ro(i+1)*sr03stl
      sr04s = sr03s*ro(i+1)
      flurptl =
   :   (titntl(i)*sr02s+titn(i)*sr02stl)+(titntl(i+1)*sr04s+titn(i+1
   :  )*sr04stl)
      flurp = sr02s*titn(i)+sr04s*titn(i+1)
      IF (0..GE.uv(i-1)) THEN
        sr01s = 0.
        sr01stl = 0.
      ELSE
        sr01stl = uvtl(i-1)
        sr01s = uv(i-1)
      END IF
      sr02stl = rotl(i-1)*sr01s+ro(i-1)*sr01stl
      sr02s = sr01s*ro(i-1)
      IF (0..LE.uv(i-1)) THEN
        sr03s = 0.
        sr03stl = 0.
```

```
    ELSE
      sr03stl = uvtl(i-1)
      sr03s = uv(i-1)
    END IF
    sr04stl = rotl(i)*sr03s+ro(i)*sr03stl
    sr04s = sr03s*ro(i)
    flurmtl =
:    (titntl(i-1)*sr02s+titn(i-1)*sr02stl)+(titntl(i)*sr04s+titn(i
:   )*sr04stl)
    flurm = sr02s*titn(i-1)+sr04s*titn(i)
    sr01s = dx/dt
    sr02s = 1./dt
    sr03s = dx/dt
    sr04stl =
:    titntl(i)*(1./(sr03s+dx/tau))+titn(i)*((-dx*((-tautl)/tau**2)
:   )/(sr03s+dx/tau)**2)
    sr04s = (1./(sr03s+dx/tau))*titn(i)
    dtl(i) =
:    (((flurptl-flurmtl)*((-1.)/(sr01s+dx/tau))-(flurp-flurm)*((-d
:   x*((-tautl)/tau**2))/(sr01s+dx/tau)**2))*ro(i)-(((-1.)/(sr01s+
:   dx/tau))*(flurp-flurm))*rotl(i))/ro(i)**2+((sigmatl(i)*(1./(sr
:   02s+1./tau))+sigma(i)*((-(-tautl)/tau**2)/(sr02s+1./tau)**2))*
:   ro(i)-((1./(sr02s+1./tau))*sigma(i))*rotl(i))/ro(i)**2+(((qntl
:   (i)-qntl(i-1))*sr04s+(qn(i)-qn(i-1))*sr04stl)*ro(i)-(sr04s*(qn
:   (i)-qn(i-1)))*rotl(i))/ro(i)**2
    d(i) =
:    (((-1.)/(sr01s+dx/tau))*(flurp-flurm))/ro(i)+((1./(sr02s+1./t
:   au))*sigma(i))/ro(i)+(sr04s*(qn(i)-qn(i-1)))/ro(i)
 END DO
 btl(1) = 0.
 b(1) = 1.
 atl(1) = 0.
 a(1) = 0.
 ctl(1) = 0.
 c(1) = 0.
 dtl(1) = -titntl(1)
 d(1) = titentr-titn(1)
 btl(n) = 0.
 b(n) = 1.
 atl(n) = 0.
 a(n) = 1.
 ctl(n) = 0.
 c(n) = 0.
 dtl(n) = 0.
 d(n) = 0.
 CALL jacobitl(c, a, d, b, dtit1, n, ctl, atl, dtl, btl, dtit1tl)
 DO i = 1, n
    tit1tl(i) = titntl(i)+dtit1tl(i)
    tit1(i) = titn(i)+dtit1(i)
    IF (tit1(i).GE.1.) THEN
      tit1tl(i) = 0.
      tit1(i) = 1.
      dtit1tl(i) = -titntl(i)
      dtit1(i) = 1.-titn(i)
    ELSE
      IF (tit1(i).LE.0.) THEN
        tit1tl(i) = 0.
        tit1(i) = 0.
        dtit1tl(i) = -titntl(i)
        dtit1(i) = -titn(i)
      END IF
    END IF
```

```
      END DO
      RETURN
      END
```

D.2 Code linéaire cotangent

```
COD Unite : solvtit1ad
COD Derivee de l'unite : solvtit1
COD IN  globals: tau uv ro titn sigma a tit1 b c qn d
COD OUT globals: dtit1 d c b tit1 a
COD Dependances entre IN et OUT
COD dtit1 <--  tau uv qn ro b titn sigma d a c
COD d <--  titn tau uv ro sigma qn d
COD c <--  tau uv ro c
COD b <--  tau uv qn ro b
COD tit1 <--  titn tau uv qn ro b sigma d a c tit1
COD a <--  tau uv ro a


      SUBROUTINE solvtit1ad ()

      include "param_ad.inc"

      PARAMETER (ndim = 100)
      IMPLICIT REAL (a-h, o-z)
      REAL sr01s, sr02s, sr03s, sr04s
      COMMON /PRESS1/p(ndim), pn(ndim), p1(ndim), p2(ndim)
      COMMON /DEBIT1/q(ndim), qn(ndim), q1(ndim), q2(ndim)
      COMMON /VITRE1/ur(ndim), urn(ndim), ur1(ndim), ur2(ndim)
      COMMON /TITRE1/tit(ndim), titn(ndim), tit1(ndim), tit2(ndim)
      COMMON /PRESS2/dp1(ndim), dp2(ndim), dp3(ndim)
      COMMON /DEBIT2/dq1(ndim), dq2(ndim), dq3(ndim)
      COMMON /VITRE2/dur1(ndim), dur2(ndim), dur3(ndim), dtit(ndim),
     : dtit1(ndim), dtit2(ndim)
      COMMON /ENTRO1/s(ndim), sn(ndim)
      COMMON /ENTRO2/ds(ndim)
      COMMON /VARPR1/cson(ndim), vp(ndim), ga(ndim)
      COMMON /VARPR2/ro(ndim), rol(ndim), rov(ndim), unsrovl(ndim)
      COMMON /VARPR3/diago(ndim), rotit(ndim), titit(ndim)
      COMMON /VARPR4/h(ndim), chl(ndim), t(ndim), tith(ndim)
      COMMON /VARPR5/alliq(ndim), cpliq(ndim)
      COMMON /VARPR6/rdp(ndim), rdc(ndim), rds(ndim)
      COMMON /VARPR7/slsat(ndim), svsat(ndim)
      COMMON /VARPR8/dslsdp(ndim), dsvsdp(ndim)
      COMMON /VARPR9/sigma(ndim)
      COMMON /VARDE1/u(ndim), v(ndim), uv(ndim), ul(ndim)
      COMMON /VARDE2/alpha1(ndim), alpha2(ndim)
      COMMON /VARDE3/beta1(ndim), beta2(ndim)
      COMMON /VARDE4/tit_u(ndim), ro_u(ndim), unsrovl_u(ndim)
      COMMON /VARDE5/rotit_u(ndim), romvl_u(ndim), sigma_u(ndim)
      COMMON /AMAILL/x(ndim)
      COMMON /RESOUD/a(ndim), b(ndim), c(ndim), d(ndim)
      COMMON /GEO1/n
      COMMON /GEO2/dx, dt, dtdx, xlong
      COMMON /PARA1/alamv, alaml, cd, temp, r, grav
      COMMON /DATE/temps, ttot
      COMMON /CONDI/psortie, qentree, uentree, urentre, titentr,
     : sentree
      COMMON /PUIS/puisvol, qvol(ndim)
      COMMON /PARA2/tau, qsi
      COMMON /DEBIT3/qt(ndim)
```

```
      COMMON /FICH01/nficur, nficp, nficq, nficrov, nfictit
      COMMON /FICH02/nficro, nficu, nficuns, nficson
      COMMON /FICH03/nficuv, nficul, nfics
      COMMON /FICH04/nficrp, nficrs, nficrc
      COMMON /IMPRES/imprime, ipasimp, idessin
      COMMON /FICH00/nficdon, nficres, motcle
      COMMON /COUR01/cnu, cnul, cnuv, cnur, cnumax, cnulmax, cnuvmax,
     : cnurmax, cnmax
      COMMON /STAT01/pmax, pmin, pmoy, pmaxt, pmint
      COMMON /STAT02/urmax, urmin, urmoy, urmaxt, urmint
      COMMON /STAT03/qmax, qmin, qmoy, qmaxt, qmint
      COMMON /STAT04/timax, timin, timoy, timaxt, timint
      COMMON /STAT05/smax, smin, smoy, smaxt, smint
      COMMON /STAT06/dtmin, dtmax, dtmoy
      COMMON /DEBIT1AD/qad(ndim), qnad(ndim), q1ad(ndim), q2ad(ndim)
      COMMON /TITRE1AD/titad(ndim), titnad(ndim), tit1ad(ndim),
     : tit2ad(ndim)
      COMMON /VITRE2AD/dur1ad(ndim), dur2ad(ndim), dur3ad(ndim),
     : dtitad(ndim), dtit1ad(ndim), dtit2ad(ndim)
      COMMON /VARPR2AD/road(ndim), rolad(ndim), rovad(ndim),
     : unsrovlad(ndim)
      COMMON /VARPR9AD/sigmaad(ndim)
      COMMON /VARDE1AD/uad(ndim), vad(ndim), uvad(ndim), ulad(ndim)
      COMMON /RESOUDAD/aad(ndim), bad(ndim), cad(ndim), dad(ndim)
      COMMON /PARA2AD/tauad, qsiad
      REAL p, pn, p1, p2
      REAL q, qn, q1, q2
      REAL ur, urn, ur1, ur2
      REAL tit, titn, tit1, tit2
      REAL dp1, dp2, dp3
      REAL dq1, dq2, dq3
      REAL dur1, dur2, dur3, dtit, dtit1, dtit2
      REAL s, sn
      REAL ds
      REAL cson, vp, ga
      REAL ro, rol, rov, unsrovl
      REAL diago, rotit, titit
      REAL h, chl, t, tith
      REAL alliq, cpliq
      REAL rdp, rdc, rds
      REAL slsat, svsat
      REAL dslsdp, dsvsdp
      REAL sigma
      REAL u, v, uv, ul
      REAL alpha1, alpha2
      REAL beta1, beta2
      REAL tit_u, ro_u, unsrovl_u
      REAL rotit_u, romvl_u, sigma_u
      REAL x
      REAL a, b, c, d
      INTEGER n
      REAL dx, dt, dtdx, xlong
      REAL alamv, alaml, cd, temp, r, grav
      REAL temps, ttot
      REAL psortie, qentree, uentree, urentre, titentr, sentree
      REAL puisvol, qvol
      REAL tau, qsi
      REAL qt
      INTEGER nficur, nficp, nficq, nficrov, nfictit
      INTEGER nficro, nficu, nficuns, nficson
      INTEGER nficuv, nficul, nfics
      INTEGER nficrp, nficrs, nficrc
```

```
 INTEGER imprime, ipasimp, idessin
 INTEGER nficdon, nficres
 CHARACTER*(1) motcle*40
 REAL cnu, cnul, cnuv, cnur, cnumax, cnulmax, cnuvmax, cnurmax,
: cnmax
 REAL pmax, pmin, pmoy, pmaxt, pmint
 REAL urmax, urmin, urmoy, urmaxt, urmint
 REAL qmax, qmin, qmoy, qmaxt, qmint
 REAL timax, timin, timoy, timaxt, timint
 REAL smax, smin, smoy, smaxt, smint
 REAL dtmin, dtmax, dtmoy
 REAL qad, qnad, q1ad, q2ad
 REAL titad, titnad, tit1ad, tit2ad
 REAL dur1ad, dur2ad, dur3ad, dtitad, dtit1ad, dtit2ad
 REAL road, rolad, rovad, unsrovlad
 REAL sigmaad
 REAL uad, vad, uvad, ulad
 REAL aad, bad, cad, dad
 REAL tauad, qsiad
 REAL save41(2:0dyn-1)
 REAL save37(2:0dyn-1)
 REAL save20(2:0dyn-1)
 REAL save16(1:0dyn)
 REAL save7
 REAL flurpad
 REAL save57(2:0dyn-1)
 REAL save40(2:0dyn-1)
 REAL save36(2:0dyn-1)
 REAL save15(1:0dyn)
 REAL save6
 REAL save56(2:0dyn-1)
 REAL save35(2:0dyn-1)
 REAL save14(1:0dyn)
 REAL save5
 REAL save55(2:0dyn-1)
 LOGICAL save34(2:0dyn-1)
 REAL save13(1:0dyn)
 REAL save4
 REAL sr03sad
 REAL save54(2:0dyn-1)
 REAL save33(2:0dyn-1)
 LOGICAL save29(2:0dyn-1)
 LOGICAL save12(1:0dyn)
 REAL save3
 REAL flurmad
 REAL save53(2:0dyn-1)
 REAL save49(2:0dyn-1)
 LOGICAL save32(2:0dyn-1)
 REAL save28(2:0dyn-1)
 LOGICAL save11(1:0dyn)
 REAL save2
 REAL save52(2:0dyn-1)
 REAL save48(2:0dyn-1)
 REAL save31(2:0dyn-1)
 LOGICAL save27(2:0dyn-1)
 REAL save10(1:0dyn)
 REAL save1
 REAL sr01sad
 REAL save51(2:0dyn-1)
 REAL save47(2:0dyn-1)
 REAL save30(2:0dyn-1)
 REAL save26(2:0dyn-1)
```

```
      REAL save50(2:Odyn-1)
      REAL save46(2:Odyn-1)
      LOGICAL save25(2:Odyn-1)
      REAL save45(2:Odyn-1)
      REAL save24(2:Odyn-1)
      REAL sr04sad
      REAL save44(2:Odyn-1)
      REAL save23(2:Odyn-1)
      REAL save19(2:Odyn-1)
      REAL save43(2:Odyn-1)
      REAL save39(2:Odyn-1)
      LOGICAL save22(2:Odyn-1)
      LOGICAL save18(2:Odyn-1)
      REAL save9(ndim)
      REAL sr02sad
      REAL save42(2:Odyn-1)
      REAL save38(2:Odyn-1)
      LOGICAL save21(2:Odyn-1)
      REAL save17(2:Odyn-1)
      REAL save8
      EXTERNAL jacobiad
C
C Initializations of local variables
C
      CONTINUE
      flurp = 0.
      sr01s = 0.
      flurm = 0.
      sr03s = 0.
      sr02sad = 0.
      sr04sad = 0.
      flurpad = 0.
      sr01sad = 0.
      sr02s = 0.
      flurmad = 0.
      sr04s = 0.
      sr03sad = 0.
C
C Trajectory
C
      CONTINUE
      DO i = 2, n-1
         save17(i) = sr01s
         sr01s = dx/dt
         save18(i) = 0..GE.uv(i-1)
         IF (save18(i)) THEN
           save57(i) = sr02s
           sr02s = 0.
         ELSE
           save56(i) = sr02s
           sr02s = uv(i-1)
         END IF
         save19(i) = a(i)
         a(i) = ((-1.)/(sr01s+dx/tau))*((-sr02s*ro(i-1))/ro(i))
         save20(i) = sr01s
         sr01s = dx/dt
         save21(i) = 0..GE.uv(i)
         IF (save21(i)) THEN
           save55(i) = sr02s
           sr02s = 0.
         ELSE
           save54(i) = sr02s
```

```
          sr02s = uv(i)
        END IF
        save22(i) = 0..LE.uv(i-1)
        IF (save22(i)) THEN
          save53(i) = sr03s
          sr03s = 0.
        ELSE
          save52(i) = sr03s
          sr03s = uv(i-1)
        END IF
        save23(i) = b(i)
        b(i) =
     :   1.+(1./(sr01s+dx/tau))*(sr02s-sr03s+(-(qn(i)-qn(i-1))/ro(i)))
        save24(i) = sr01s
        sr01s = dx/dt
        save25(i) = 0..LE.uv(i)
        IF (save25(i)) THEN
          save51(i) = sr02s
          sr02s = 0.
        ELSE
          save50(i) = sr02s
          sr02s = uv(i)
        END IF
        save26(i) = c(i)
        c(i) = ((-1.)/(sr01s+dx/tau))*((sr02s*ro(i+1))/ro(i))
        save27(i) = 0..GE.uv(i)
        IF (save27(i)) THEN
          save49(i) = sr01s
          sr01s = 0.
        ELSE
          save48(i) = sr01s
          sr01s = uv(i)
        END IF
        save28(i) = sr02s
        sr02s = sr01s*ro(i)
        save29(i) = 0..LE.uv(i)
        IF (save29(i)) THEN
          save47(i) = sr03s
          sr03s = 0.
        ELSE
          save46(i) = sr03s
          sr03s = uv(i)
        END IF
        save30(i) = sr04s
        sr04s = sr03s*ro(i+1)
        save31(i) = flurp
        flurp = sr02s*titn(i)+sr04s*titn(i+1)
        save32(i) = 0..GE.uv(i-1)
        IF (save32(i)) THEN
          save45(i) = sr01s
          sr01s = 0.
        ELSE
          save44(i) = sr01s
          sr01s = uv(i-1)
        END IF
        save33(i) = sr02s
        sr02s = sr01s*ro(i-1)
        save34(i) = 0..LE.uv(i-1)
        IF (save34(i)) THEN
          save43(i) = sr03s
          sr03s = 0.
        ELSE
```

```
          save42(i) = sr03s
          sr03s = uv(i-1)
        END IF
        save35(i) = sr04s
        sr04s = sr03s*ro(i)
        save36(i) = flurm
        flurm = sr02s*titn(i-1)+sr04s*titn(i)
        save37(i) = sr01s
        sr01s = dx/dt
        save38(i) = sr02s
        sr02s = 1./dt
        save39(i) = sr03s
        sr03s = dx/dt
        save40(i) = sr04s
        sr04s = (1./(sr03s+dx/tau))*titn(i)
        save41(i) = d(i)
        d(i) =
     :    (((-1.)/(sr01s+dx/tau))*(flurp-flurm))/ro(i)+((1./(sr02s+1./t
     :   au))*sigma(i))/ro(i)+(sr04s*(qn(i)-qn(i-1)))/ro(i)
      END DO
      save1 = b(1)
      b(1) = 1.
      save2 = a(1)
      a(1) = 0.
      save3 = c(1)
      c(1) = 0.
      save4 = d(1)
      d(1) = titentr-titn(1)
      save5 = b(n)
      b(n) = 1.
      save6 = a(n)
      a(n) = 1.
      save7 = c(n)
      c(n) = 0.
      save8 = d(n)
      d(n) = 0.
      DO n1 = 1, ndim
         save9(n1) = dtit1(n1)
      END DO
      CALL jacobi(c, a, d, b, dtit1, n)
      DO i = 1, n
         save10(i) = tit1(i)
         tit1(i) = titn(i)+dtit1(i)
         save11(i) = tit1(i).GE.1.
         IF (save11(i)) THEN
           save15(i) = tit1(i)
           tit1(i) = 1.
           save16(i) = dtit1(i)
           dtit1(i) = 1.-titn(i)
         ELSE
           save12(i) = tit1(i).LE.0.
           IF (save12(i)) THEN
             save13(i) = tit1(i)
             tit1(i) = 0.
             save14(i) = dtit1(i)
             dtit1(i) = -titn(i)
           END IF
         END IF
      END DO
C
C Transposed linear forms
C
```

```
      CONTINUE
      DO i = n, 1, -1
         IF (save11(i)) THEN
           dtit1(i) = save16(i)
           titnad(i) = titnad(i)-dtit1ad(i)
           dtit1ad(i) = 0.
           tit1(i) = save15(i)
           tit1ad(i) = 0.
         ELSE
           IF (save12(i)) THEN
             dtit1(i) = save14(i)
             titnad(i) = titnad(i)-dtit1ad(i)
             dtit1ad(i) = 0.
             tit1(i) = save13(i)
             tit1ad(i) = 0.
           END IF
         END IF
         tit1(i) = save10(i)
         titnad(i) = titnad(i)+tit1ad(i)
         dtit1ad(i) = dtit1ad(i)+tit1ad(i)
         tit1ad(i) = 0.
      END DO
      DO n1 = ndim, 1, -1
         dtit1(n1) = save9(n1)
      END DO
      CALL jacobiad(c, a, d, b, dtit1, n, cad, aad, dad, bad, dtit1ad)
      d(n) = save8
      dad(n) = 0.
      c(n) = save7
      cad(n) = 0.
      a(n) = save6
      aad(n) = 0.
      b(n) = save5
      bad(n) = 0.
      d(1) = save4
      titnad(1) = titnad(1)-dad(1)
      dad(1) = 0.
      c(1) = save3
      cad(1) = 0.
      a(1) = save2
      aad(1) = 0.
      b(1) = save1
      bad(1) = 0.
      DO i = n-1, 2, -1
         d(i) = save41(i)
         flurpad =
     :    flurpad+dad(i)*((1./ro(i)**2)*(((-1.)/(sr01s+dx/tau))*ro(i)))
         flurmad =
     :    flurmad-dad(i)*((1./ro(i)**2)*(((-1.)/(sr01s+dx/tau))*ro(i)))
         sigmaad(i) =
     :    sigmaad(i)+dad(i)*((1./ro(i)**2)*((1./(sr02s+1./tau))*ro(i)))
         tauad =
     :    tauad+dad(i)*((-(1./ro(i)**2)*((((1./(sr01s+dx/tau)**2)*((1./
     :   tau**2)*dx))*(flurp-flurm))*ro(i)))+(1./ro(i)**2)*((((1./(sr02
     :   s+1./tau)**2)*(1./tau**2))*sigma(i))*ro(i)))
         qnad(i) = qnad(i)+dad(i)*((1./ro(i)**2)*(sr04s*ro(i)))
         qnad(i-1) = qnad(i-1)-dad(i)*((1./ro(i)**2)*(sr04s*ro(i)))
         sr04sad =
     :    sr04sad+dad(i)*((1./ro(i)**2)*((qn(i)-qn(i-1))*ro(i)))
         road(i) =
     :    road(i)+dad(i)*((-(1./ro(i)**2)*(((-1.)/(sr01s+dx/tau))*(flur
     :   p-flurm)))-(1./ro(i)**2)*((1./(sr02s+1./tau))*sigma(i))+(-(1./
```

```
     :   ro(i)**2)*(sr04s*(qn(i)-qn(i-1)))))
         dad(i) = 0.
         sr04s = save40(i)
         titnad(i) = titnad(i)+sr04sad*(1./(sr03s+dx/tau))
         tauad =
     :    tauad+sr04sad*(((1./(sr03s+dx/tau)**2)*((1./tau**2)*dx))*titn
     :   (i))
         sr04sad = 0.
         sr03s = save39(i)
         sr02s = save38(i)
         sr01s = save37(i)
         flurm = save36(i)
         titnad(i-1) = titnad(i-1)+flurmad*sr02s
         sr02sad = sr02sad+flurmad*titn(i-1)
         titnad(i) = titnad(i)+flurmad*sr04s
         sr04sad = sr04sad+flurmad*titn(i)
         flurmad = 0.
         sr04s = save35(i)
         road(i) = road(i)+sr04sad*sr03s
         sr03sad = sr03sad+sr04sad*ro(i)
         sr04sad = 0.
         IF (save34(i)) THEN
           sr03sad = 0.
           sr03s = save43(i)
         ELSE
           sr03s = save42(i)
           uvad(i-1) = uvad(i-1)+sr03sad
           sr03sad = 0.
         END IF
         sr02s = save33(i)
         road(i-1) = road(i-1)+sr02sad*sr01s
         sr01sad = sr01sad+sr02sad*ro(i-1)
         sr02sad = 0.
         IF (save32(i)) THEN
           sr01sad = 0.
           sr01s = save45(i)
         ELSE
           sr01s = save44(i)
           uvad(i-1) = uvad(i-1)+sr01sad
           sr01sad = 0.
         END IF
         flurp = save31(i)
         titnad(i) = titnad(i)+flurpad*sr02s
         sr02sad = sr02sad+flurpad*titn(i)
         titnad(i+1) = titnad(i+1)+flurpad*sr04s
         sr04sad = sr04sad+flurpad*titn(i+1)
         flurpad = 0.
         sr04s = save30(i)
         road(i+1) = road(i+1)+sr04sad*sr03s
         sr03sad = sr03sad+sr04sad*ro(i+1)
         sr04sad = 0.
         IF (save29(i)) THEN
           sr03sad = 0.
           sr03s = save47(i)
         ELSE
           sr03s = save46(i)
           uvad(i) = uvad(i)+sr03sad
           sr03sad = 0.
         END IF
         sr02s = save28(i)
         road(i) = road(i)+sr02sad*sr01s
         sr01sad = sr01sad+sr02sad*ro(i)
```

```
    sr02sad = 0.
    IF (save27(i)) THEN
      sr01sad = 0.
      sr01s = save49(i)
    ELSE
      sr01s = save48(i)
      uvad(i) = uvad(i)+sr01sad
      sr01sad = 0.
    END IF
    c(i) = save26(i)
    road(i+1) =
:    road(i+1)+cad(i)*(((1./ro(i)**2)*(sr02s*ro(i)))*((-1.)/(sr01s
:   +dx/tau)))
    sr02sad =
:    sr02sad+cad(i)*(((1./ro(i)**2)*(ro(i+1)*ro(i)))*((-1.)/(sr01s
:   +dx/tau)))
    road(i) =
:    road(i)-cad(i)*(((1./ro(i)**2)*(sr02s*ro(i+1)))*((-1.)/(sr01s
:   +dx/tau)))
    tauad =
:    tauad-cad(i)*(((1./(sr01s+dx/tau)**2)*((1./tau**2)*dx))*((1./
:   ro(i))*(sr02s*ro(i+1))))
    cad(i) = 0.
    IF (save25(i)) THEN
      sr02sad = 0.
      sr02s = save51(i)
    ELSE
      sr02s = save50(i)
      uvad(i) = uvad(i)+sr02sad
      sr02sad = 0.
    END IF
    sr01s = save24(i)
    b(i) = save23(i)
    sr02sad = sr02sad+bad(i)*(1./(sr01s+dx/tau))
    sr03sad = sr03sad-bad(i)*(1./(sr01s+dx/tau))
    qnad(i) =
:    qnad(i)-bad(i)*(((1./ro(i)**2)*ro(i))*(1./(sr01s+dx/tau)))
    qnad(i-1) =
:    qnad(i-1)+bad(i)*(((1./ro(i)**2)*ro(i))*(1./(sr01s+dx/tau)))
    road(i) =
:    road(i)+bad(i)*(((1./ro(i)**2)*(qn(i)-qn(i-1)))*(1./(sr01s+dx
:   /tau)))
    tauad =
:    tauad+bad(i)*(((1./(sr01s+dx/tau)**2)*((1./tau**2)*dx))*(sr02
:   s-sr03s+(-(qn(i)-qn(i-1))/ro(i))))
    bad(i) = 0.
    IF (save22(i)) THEN
      sr03sad = 0.
      sr03s = save53(i)
    ELSE
      sr03s = save52(i)
      uvad(i-1) = uvad(i-1)+sr03sad
      sr03sad = 0.
    END IF
    IF (save21(i)) THEN
      sr02sad = 0.
      sr02s = save55(i)
    ELSE
      sr02s = save54(i)
      uvad(i) = uvad(i)+sr02sad
      sr02sad = 0.
    END IF
```

```
        sr01s = save20(i)
        a(i) = save19(i)
        road(i-1) =
     :   road(i-1)-aad(i)*(((1./ro(i)**2)*(sr02s*ro(i)))*((-1.)/(sr01s
     :  +dx/tau)))
        sr02sad =
     :   sr02sad-aad(i)*(((1./ro(i)**2)*(ro(i-1)*ro(i)))*((-1.)/(sr01s
     :  +dx/tau)))
        road(i) =
     :   road(i)+aad(i)*(((1./ro(i)**2)*(sr02s*ro(i-1)))*((-1.)/(sr01s
     :  +dx/tau)))
        tauad =
     :   tauad+aad(i)*(((1./(sr01s+dx/tau)**2)*((1./tau**2)*dx))*((1./
     :  ro(i))*(sr02s*ro(i-1))))
        aad(i) = 0.
        IF (save18(i)) THEN
          sr02sad = 0.
          sr02s = save57(i)
        ELSE
          sr02s = save56(i)
          uvad(i-1) = uvad(i-1)+sr02sad
          sr02sad = 0.
        END IF
        sr01s = save17(i)
      END DO
      RETURN
      END
```

Annexe E

Passage en simple précision

cd = 0.7

5.E − 08	2.37039	2.37039	−14.3051
5.E − 07	2.37039	2.37039	−2.86102
5.E − 06	2.37040	2.37037	−2.90871
5.E − 05	2.37053	2.37025	−2.74658
5.E − 04	2.37174	2.36906	−2.68149
5.E − 03	2.38392	2.35704	−2.68769
5.E − 02	2.51310	2.24340	−2.69707
5.E − 01	5.75859	1.51767	−4.24092

qsi = 1.

5.E − 07	2.37039	2.37039	−23.8419E − 02
5.E − 06	2.37039	2.37039	−16.6893E − 02
5.E − 05	2.37039	2.37039	−1.19209E − 02
5.E − 04	2.37039	2.37038	−1.19209E − 02
5.E − 03	2.37045	2.37034	−1.08719E − 02
5.E − 02	2.37099	2.36985	−1.14083E − 02
5.E − 01	2.37596	2.36471	−1.12560E − 02

tau = 1.E-01

5.E − 09	2.37039	2.37039	119.209
5.E − 08	2.37039	2.37039	16.6893
5.E − 07	2.37039	2.37039	−0.953674
5.E − 06	2.37039	2.37039	−2.38419E − 02
5.E − 05	2.37037	2.37040	0.283718
5.E − 04	2.37030	2.37051	0.215054
5.E − 03	2.36923	2.37157	0.234795
5.E − 02	2.36125	2.38250	0.212548
5.E − 01	2.55043	2.43209	−0.118338

puisvol = 5.93589E+08

5E + 01	2.37039	2.37039	1.66893E − 08
5E + 02	2.37038	2.37039	1.23978E − 08
5E + 03	2.37035	2.37043	8.01086E − 09
5E + 04	2.37001	2.37083	8.21352E − 09
5E + 05	2.36618	2.37460	8.42357E − 09
5E + 06	2.32846	2.41234	8.38833E − 09
5E + 07	1.96450	2.80219	8.37692E − 09
5E + 08	0.382257	7.10947	6.72721E − 09

urentre = 0.

$5.00000E-06$	2.37039	2.37039	0.
$5.00000E-05$	2.37039	2.37039	$3.09944E-02$
$5.00000E-04$	2.37039	2.37039	$-1.43051E-03$
$5.00000E-03$	2.37039	2.37039	$-4.76837E-05$
$5.00000E-02$	2.37039	2.37039	$1.43051E-05$
0.500000	2.37039	2.37039	$2.38419E-07$
5.00000	2.37039	2.37039	$-2.38419E-08$
50.0000	2.37039	2.37039	$-4.52995E-08$

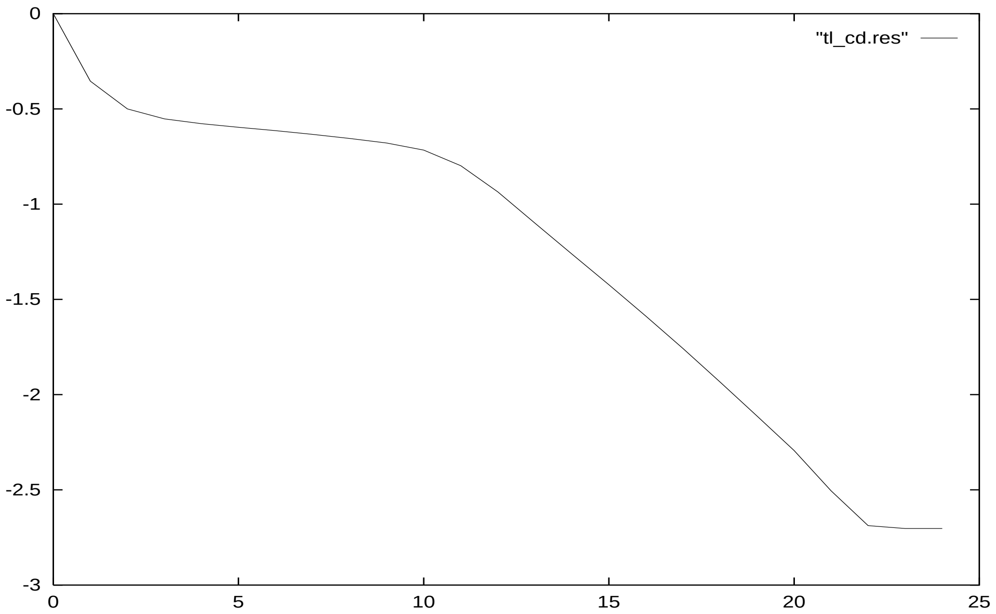

FIG. E.1 – *Dérivée en linéaire tangent par rapport à* `cd`

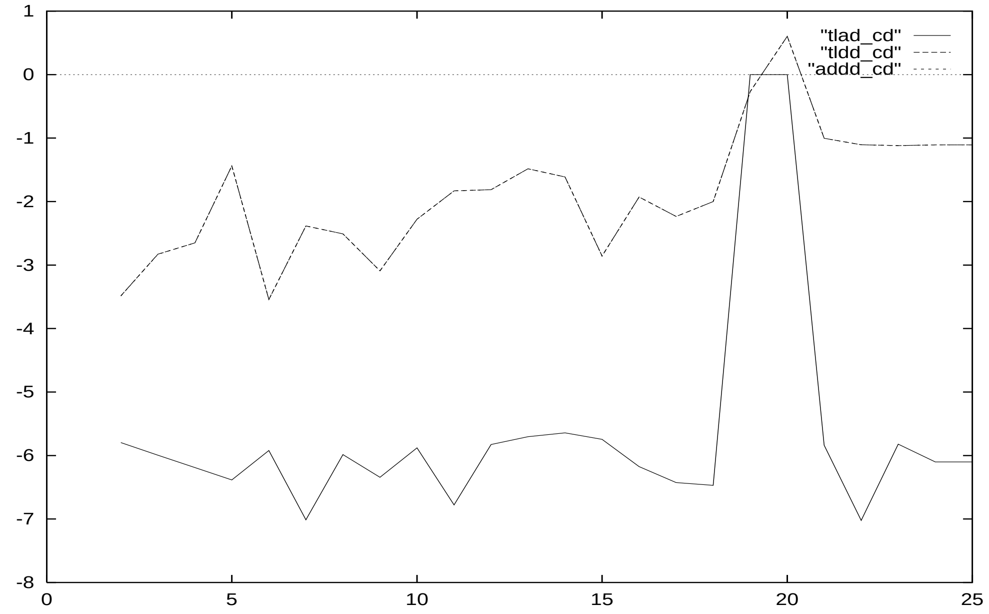

FIG. E.2 – *Différences entre les dérivées par rapport à* `cd`

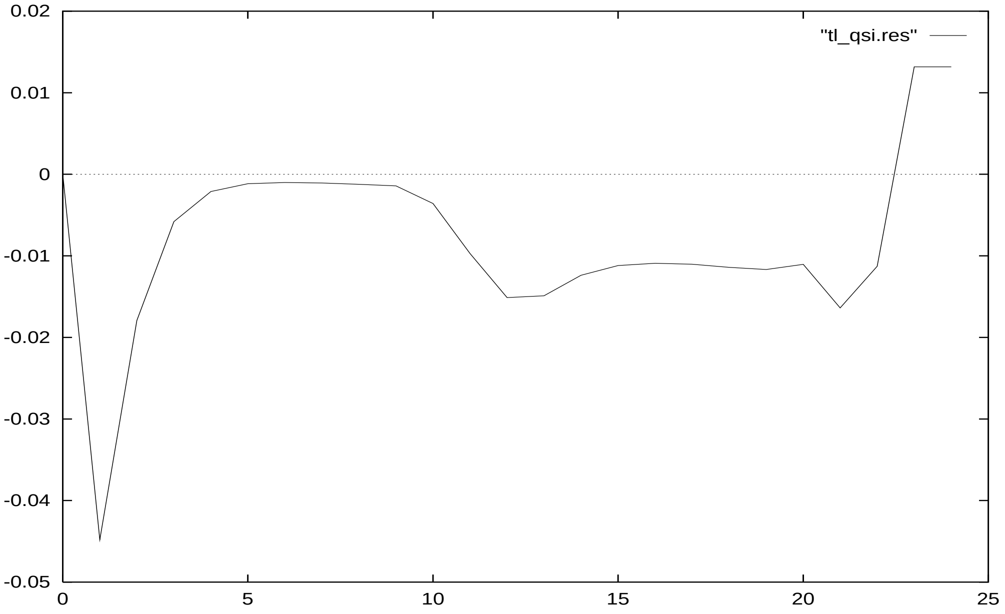

FIG. E.3 – *Dérivée en linéaire tangent par rapport à* `qsi`

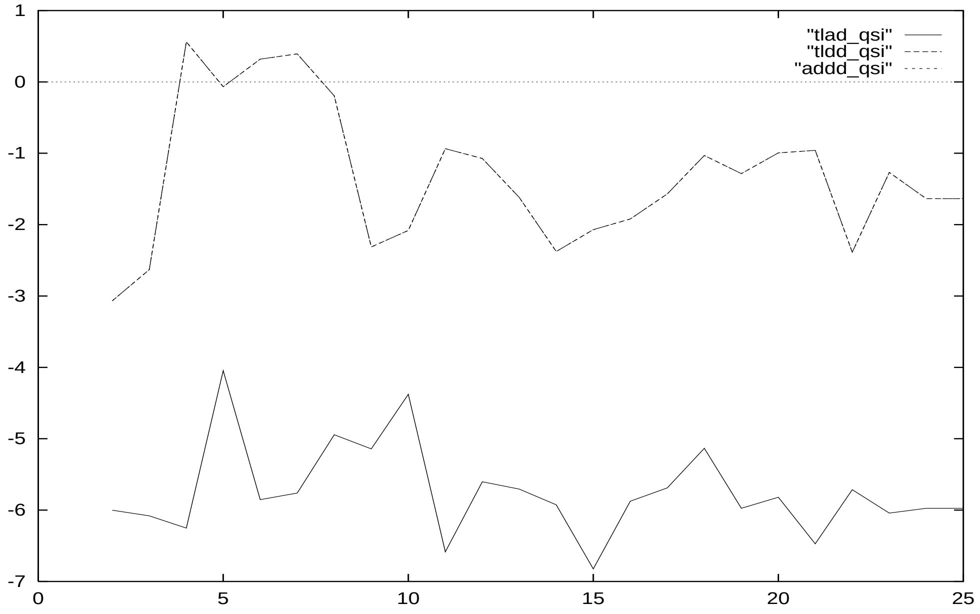

FIG. E.4 – *Différences entre les dérivées par rapport à* `qsi`

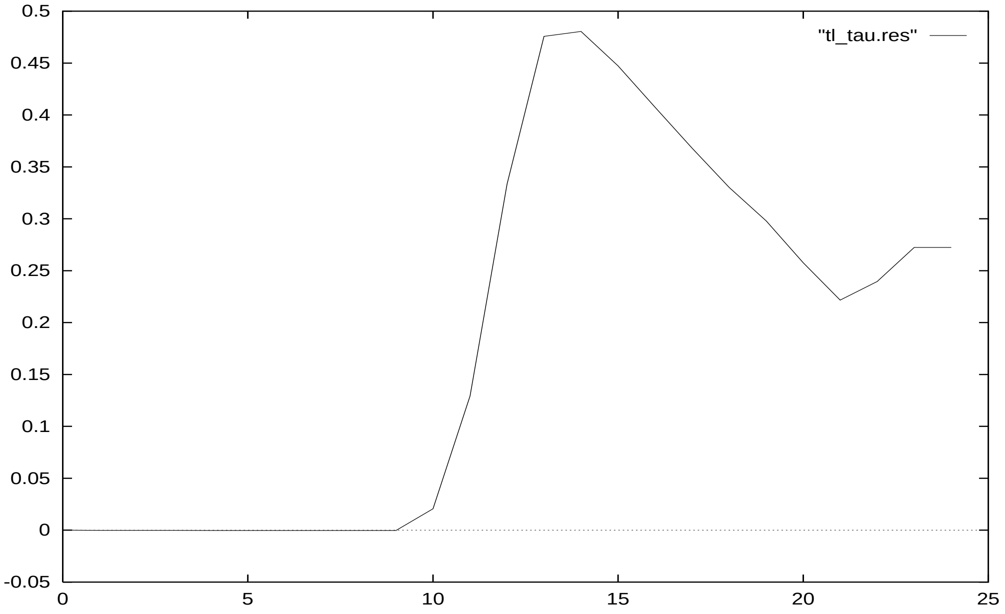

FIG. E.5 – *Dérivée en linéaire tangent par rapport à* `tau`

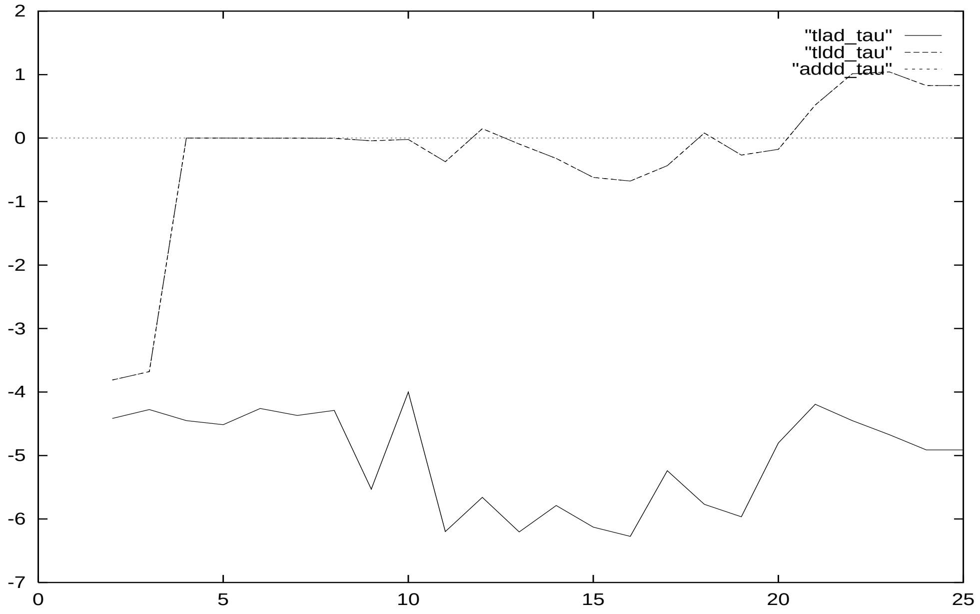

FIG. E.6 – *Différences entre les dérivées par rapport à* `tau`

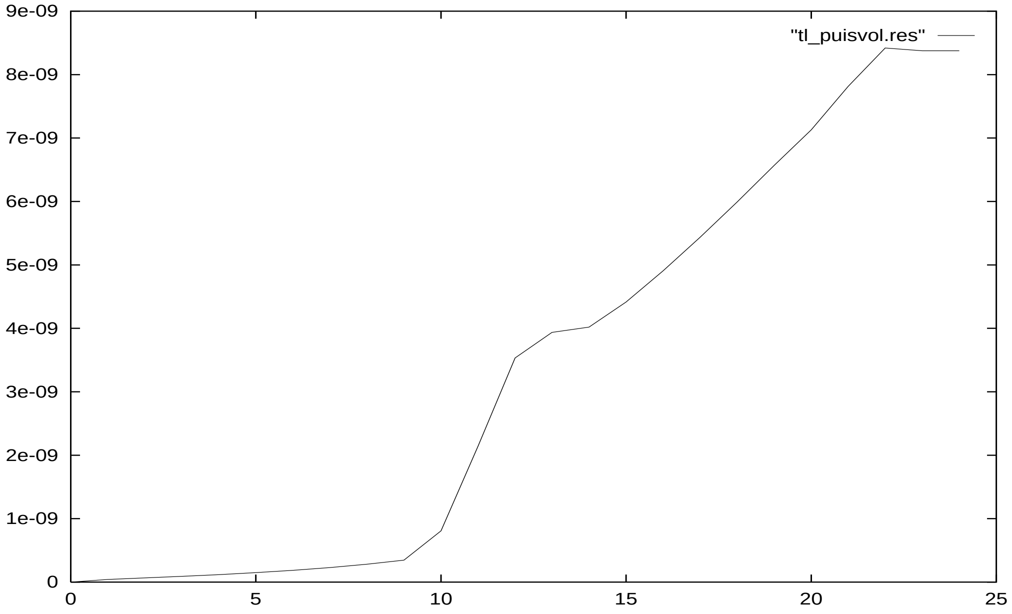

FIG. E.7 – *Dérivée en linéaire tangent par rapport à* `puisvol`

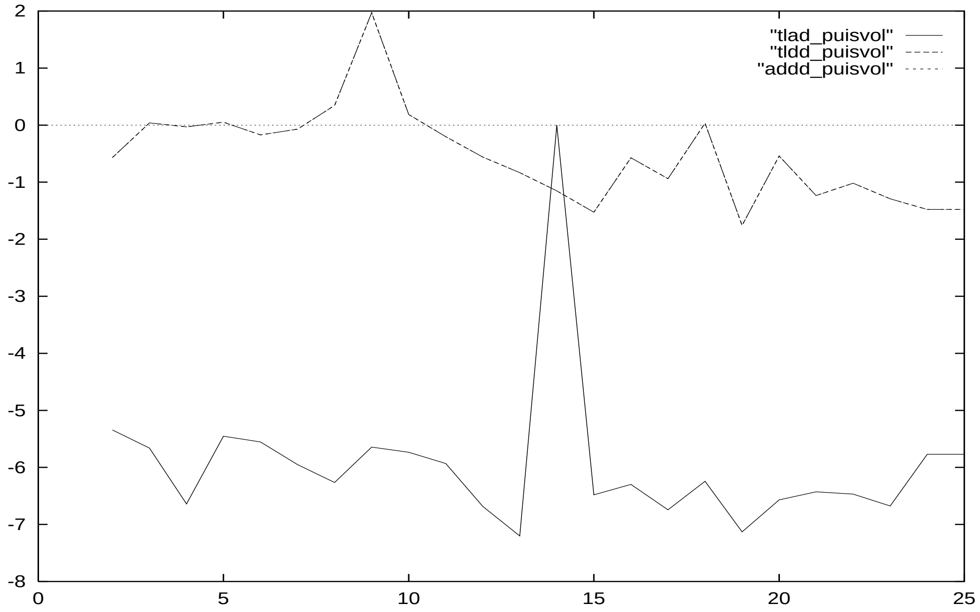

FIG. E.8 – *Différences entre les dérivées par rapport à* `puisvol`

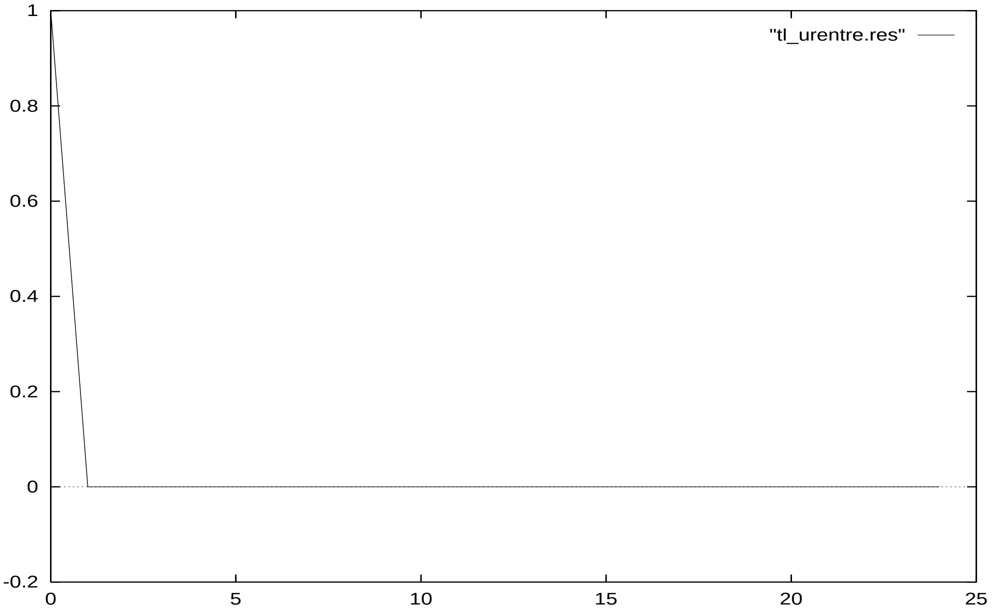

FIG. E.9 – *Dérivée en linéaire tangent par rapport à* `urentre`

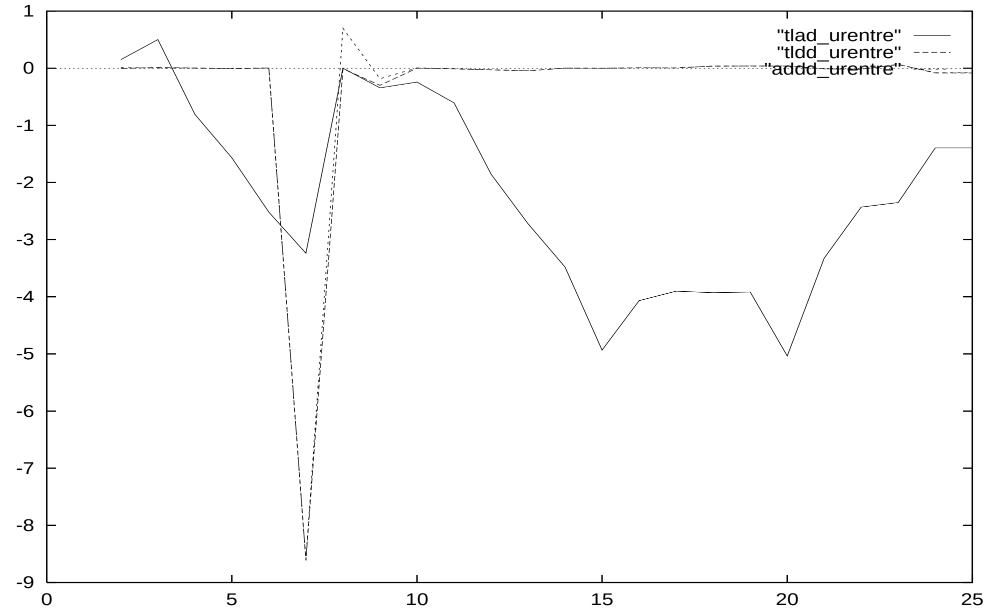

FIG. E.10 – *Différences entre les dérivées par rapport à* `urentre`

Annexe F

Manuel d'ODYSSÉE

ODYSSÉE possède un langage de commandes permettant d'écrire deux sortes de fichiers ; le fichier `.odysseerc` chargé automatiquement à l'initialisation du système et des fichiers batch permettant d'exécuter une séquence de commandes telle que : lire/dériver/imprimer.

Un autre langage permet de décrire la base d'informations par l'écriture d'un fichier `head_BI.od` lu automatiquement lors de la dérivation du programme dont l'unité de tête est `head`.

Ces deux langages ont dans cette version une syntaxe dérivée de celle de `Caml light`, ce qui implique que chaque commande doit être suivie de `;;`, que la liste des trois éléments `x,y,z` s'écrit `[x;y;z]`, qu'une chaîne de caractères est entre `"`.

F.1 Le langage de commande

F.1.1 Les commandes

Gestion des variables globales Les commandes d'accès (lecture/écriture) aux variables globales sont :

`set search_path ["dir"]` Positionne la variable globale `search_path` au répertoire "dir".

`get include_path` Accède à la valeur de la variable globale `include_path`.

Les variables décrivant le mode d'impression des fichiers Fortran sont :

all_uppercase (booléen) Si vrai, le programme engendré est entièrement écrit en majuscules, si faux seuls les mots-clé le sont.

do_indentation (entier) Nombre de caractères d'indentation du corps du `do`.

if_indentation (entier) Nombre de caractères d'indentation du corps du `if`.

continuation_char (chaîne) Caractère de continuation.

blank_line_after_stat (booléen) Si vrai, le programme engendré contient une ligne blanche après chaque instruction.

parenthesis_level (entier) Règle le sur-parenthésage.

Les chemins permettant l'accès aux fichiers sont :

search_path (liste de chaînes) Liste des répertoires dans lesquels les fichiers Fortran, les fichiers de description de la base d'informations et les fichiers de batch sont lus.

include_path (liste de chaînes) Liste des répertoires dans lesquels les fichiers inclus sont lus.

home_dir (chaîne) Répertoire racine de l'utilisateur (`$home`).

Les variables décrivant la stratégie de dérivation sont :

minimal_split (booléen) Si vrai, ODYSSÉE isole les constantes et les arguments des appels fonctions et de sous-routines dans des variables locales. Si faux, le système découpe toutes les expressions algébriques en sous-expressions ne contenant qu'une seule variable active.

Gestion de la librairie La librairie contient toutes les unités accessibles, c'est-à-dire toutes les unités chargées, ainsi que celles générées par dérivation.

`make_library ["toto.f";"titi.f"]` Ajout des unités contenues dans les fichiers "toto.f" et dans "titi.f" à la librairie.

`complete_library "toto.f"` Ajout des unités contenues dans "toto.f" à la librairie.

`clear_library ()` Suppression de toutes les unités de la librairie.

`content_library ()` Impression à l'écran des noms de toutes les unités de la librairie accompagnés du type de l'unité (`subroutine, function`).

`graph_library ()` Impression à l'écran du graphe d'appel des unités de la librairie.

Dérivation Il existe deux fonctions de dérivation dont les noms sont de la forme `diff_X_unit` où X remplace `tangent` ou `cotangent`.

`diff_X_program "essai" ["a";"b";"c"]` Dérivation du programme dont la tête est "essai" par rapport aux entrées `["a";"b";"c"]`. Cette fonction effectue automatiquement le chargement du fichier "essai_BI.od" s'il existe. Elle extrait les unités du programme "essai" de la librairie et introduit les unités dérivées dans cette même librairie.

Impression Il existe deux modes d'impression, le préfixe `print` indique une impression à l'écran, le préfixe `save` indique une impression dans un fichier (il faut alors ajouter en dernier argument le nom du fichier ou écrire).

Les commandes suivantes doivent être préfixées par `print` où `save`.

`_unit "essai"` Impression de l'unité de la librairie de nom "essai".

`_symboltable_unit "essai"` Impression dans le fichier `essai_symboltable` de la table des symboles de l'unité "essai".

`_program ()` Impression de toutes les unités du programme courant (n'existe qu'après dérivation).

`_in_out ()` Impression des entrées et des sorties de toutes les unités du programme courant.

`_dependency ()` Impression des dépendances sorties/entrées de toutes les unités du programme courant.

`_diffprogram ()` Impression de toutes les unités du programme dérivé courant (n'existe qu'après dérivation).

`_active_in_out ()` Impression des entrées et des sorties actives de toutes les unités du programme dérivé courant.

Système Les commandes système.

`exec "file"` Chargement d'un fichier batch.

`quit ()` Ferme ODYSSÉE sans vérification ni sauvegarde de la librairie.

F.1.2 Les fichiers de commande pour ODYSSÉE

Il existe deux sortes de fichier pouvant actuellement être lus par ODYSSÉE le fichier `.odysseerc` devant se trouver à la racine et les fichiers batchs devant se trouver dans les répertoires de `search_path`.

.odysseerc Ce fichier est chargé automatiquement à l'initialisation d'ODYSSÉE et contient donc des informations globales. Il permet de configurer ODYSSÉE et en particulier de positionner les chemins pour trouver des fichiers lus.

batch Les fichiers batch chargés par ODYSSÉE permettent uniquement de lire-dériver-imprimer en une seule commande. Ils peuvent être exécutés à l'interface, où dans l'interprète grâce à la commande `exec`.

F.2 Le langage de description de la `BI`

F.2.1 Déclaration d'informations

`set_info_dummys name [args]` positionne la liste des arguments de l'unité de nom `name`.

`set_info_in_out name [in_args] [in_globs] [out_args] [out_globs]` positionne les listes des arguments d'entrée `in_args`, des globales `in_globs`, des arguments de sortie `out_args`, des commons de sortie `out_globs` de l'unité `name`. Cela positionne aussi les dépendances au défaut.

`set_info_dependency name [[out;[in1;in2...]] ...]` positionne les dépendances entre une sortie `out` et une liste d'entrées `[in1;in2...]` pour chaque sortie de l'unité `name`.

F.2.2 Les fichiers de description de la `BI`

La `BI` associée au programme `toto` doit avoir pour nom `toto_BI.od` et doit contenir une description des unités de ce programme par les commandes décrites précédemment. Il est chargé automatiquement par ODYSSÉE lors de la dérivation de `toto`. Il doit se trouver dans l'un des répertoires contenus dans la variable `search_path`.

F.3 Utilisation d'ODYSSÉE

Il y a deux moyens d'utiliser ODYSSÉE, soit à travers l'interface graphique, soit à travers le langage de commande directement.

Dans chaque cas, les opérations à effectuer pour dériver un programme sont :

1. Chargement de fichiers Fortran ⇒ construction de la librairie (éventuellement composée de plusieurs programmes au sens d'ODYSSÉE).
2. Sélection de l'unité de tête, de la méthode et des variables de dérivation ⇒ construction du programme dérivé et augmentation de la librairie.
3. Impression du résultat dans un fichier ou à l'écran.

La base d'informations est chargée automatiquement durant l'étape `2`.

F.3.1 À l'interface

ODYSSÉE possède une interface utilisateur qui peut être appelée par : `xodyssee`.

Les opérations à effectuer pour dériver un programme sont :

File : Load Source Lecture des fichiers sources.

Current Unit : Sélection de l'unité de tête du programme.

Differentiation : Mode Sélection du mode de dérivation, puis des entrés actives dans une fenêtre qui apparaît à l'écran.

File : Save Source Sauvegarde de l'unité sélectionnée.

Ces opérations peuvent être également effectuées en une commande par chargement d'un fichier batch à l'interface.

Remarque : Cette version de l'interface offre des possibilités dans ces menus incompatibles avec la version `1.5` d'ODYSSÉE.

F.3.2 Sous le langage de commande

ODYSSÉE possède un interprète associé à son langage de commande et directement accessible à l'utilisateur par : `odyssee`.

Les opérations à effectuer pour dériver un programme sont :

make_library Lecture des fichiers sources (en n'oubliant pas l'extension `.f` par exemple).

diff_X_program Dérivation du programme dont l'unité de tête est spécifiée en mode `X`, par rapport aux entrées actives elles aussi spécifiées dans la commande.

save_diffprogram Sauvegarde du (dernier) programme dérivé dans le fichier spécifié.

Sous l'interprète, toutes les commandes décrites (F.1) sont valides.

Bibliographie

[1] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, S.M. Watt (1988). Maple reference manuel. Symbolic Computation Group, Department of Computer Science, University of Waterloo, Ontario, Canada.

[2] G. Corliss, A. Griewank (Editeurs). *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, number 53 in Proceedings in Applied Mathematics. SIAM, Philadelphia, (1991).

[3] J.E. Dennis, R.B. Schnabel (1983). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Englewood Cliffs.

[4] C. Faure (1996). Splitting of algebraic expressions for automatic differentiation. Présenté au "2nd International workshop on Computational Differentiation" à Santa Fe (12-15 fév. 1996).

[5] J.Ch. Gilbert (1992). Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1, 13–21.

[6] J.Ch. Gilbert, G. Le Vey, J. Masse (1991). La différentiation automatique de fonctions représentées par des programmes. Rapport de Recherche n^o 1557, INRIA, BP 105, F-78153 Le Chesnay, France.

[7] A. Griewank (1989). On automatic differentiation. In M. Iri, K. Tanabe (Editeurs), *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, Dordrecht.

[8] A. Griewank (1992). Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1, 35–54.

[9] J. Grimm, L. Pottier, N. Rostaing-Schmidt (1996). Optimal time and minimum space-time product for reversing a certain class of programs. Présenté au "2nd International workshop on Computational Differentiation" à Santa Fe (12-15 fév. 1996).

[10] M. Iri (1984). Simultaneous computation of functions, partial derivatives and estimates of rounding errors, complexity and practicality. *Japan Journal of Applied Mathematics*, 1, 223–252.

[11] M. Iri, K. Kubota (1987). Methods of fast automatic differentiation and applications. Research Memorandum RMI 87-02, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, Hongo 7-3-1, Bunkyo-ku, Tokyo, Japan.

[12] K.V. Kim, Yu.E. Nesterov, B.V. Cherkasskiĭ (1984). An estimate of the effort in computing the gradient. *Soviet Math. Dokl.*, 29, 384–387.

[13] J. Morgenstern (1985). How to compute fast a function and all its derivatives, a variation on the theorem of Baur-Strassen. *SIGACT News*, 16, 60–62.

[14] N. Rostaing, S. Dalmas, A. Galligo (1993). Automatic differentiation in Odyssée. *Tellus*, 45, 558–568.

[15] N. Rostaing-Schmidt (1993). *Différentiation automatique : application à un problème d'optimisation en météorologie.* Thèse, Université de Nice-Sophia Antipolis, France.

[16] J.W. Sawyer (1984). First partial differentiation by computer with an application to categorial data analysis. *The American Statistician*, 38, 300–308.

[17] B. Speelpenning (1980). *Compiling fast partial derivatives of functions given by algorithms.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

[18] O. Talagrand (1991). The use of adjoint equations in numerical modelling of the atmospheric circulation. In A. Griewank, G. Corliss (Editeurs), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Proceedings in Applied Mathematics 53, pages 169–180. SIAM.

[19] Vax Unix Macsyma: Reference manual, version 11 (1985). Symbolics.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399