Génération automatique de codes adjoints : Stratégies d'utilisation pour le logiciel Odyssée, Application au code météorologique Meso-NH

Isabelle Charpentier, Mohammed Ghémirès

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Génération automatique de codes adjoints : Stratégies d'utilisation pour le logiciel Odyssée, Application au code météorologique Meso-NH

Isabelle Charpentier et Mohammed Ghémirès

N° 3251

Septembre 1997

THÈME 4

ISSN 0249-6399

INRIA
RHÔNE-ALPES

Génération automatique de codes adjoints : Stratégies d'utilisation pour le logiciel Odyssée, Application au code météorologique Meso-NH

Isabelle Charpentier et Mohammed Ghémirès

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet Idopt

Rapport de recherche n° 3251 — Septembre 1997 — 39 pages

Résumé : L'outil de différentiation automatique Odyssée 1.6 est appliqué au modèle météorologique Meso-NH pour générer les codes linéaire tangent et linéaire cotangent (adjoint). Odyssée permet d'éviter la construction manuelle des codes linéaires, mais ce logiciel produit des codes exécutables souvent très gros et difficiles à utiliser pour faire de l'optimisation.
Cet inconvénient, qui découle principalement des calculs locaux de la trajectoire nécessaire à l'évaluation du code cotangent, peut être éliminé par sauvegarde de la trajectoire sur fichier. Dans ce rapport nous proposons un schéma algorithmique qui permet d'effectuer automatiquement les sauvegardes sur fichier en modifiant les codes linéaires tangent et cotangent, ce processus est optimisé pour chaque routine.
Cet algorithme a été utilisé pour la différentiation de Meso-NH par rapport à l'état initial. Après une étape de validation, nous avons observé que l'exécution de ces codes linéaires est peu coûteuse en temps calcul et en mémoire : une exécution des codes linéaires nécessite seulement 3 fois plus de temps qu'une exécution de Meso-NH et la place mémoire occupée est suffisamment petite pour exécuter des simulations météorologiques ayant un sens physique.

Mots-clé : Différentiation automatique, Odyssée 1.6, code adjoint, sauvegarde de trajectoires, tests de validation, Meso-NH 2.4.

Remerciements: *L'étude a été financée successivement par le projet IDOPT (INRIA, CNRS, UJF, INPG) et l'action incitative Mode Inverse Opérationnel (INRIA). Le développement de l'adjoint de Meso-NH, réalisé au Laboratoire de Modélisation et de Calcul (UMR 5523, Grenoble), au Laboratoire d'Aérologie (Toulouse) et au CEMRACS (CIRM, Luminy), a bénéficié des moyens informatiques offerts par l'IDRIS. Les auteurs remercient Jean Pierre Pinty (Laboratoire d'Aérologie) pour ses précieux conseils sur le modèle Meso-NH.*

(Abstract: pto)

Automatic generation of adjoint codes: Some strategic uses of Odyssée, Application to the meteorological code Meso-NH

Abstract:

The automatic differentiation tool Odyssée 1.6 is applied to the 3D meteorological model Meso-NH in order to generate both the tangent linear code and the cotangent linear code (adjoint). Odyssée allows to avoid hand coding construction of the linear codes, but this software often generates huge executable codes that are usually difficult to use when one wants to perform optimization process.

This drawback mainly due to the local computations of the path necessary to evaluate the cotangent linear code of the model can be removed by the use of file saves for the path. In this report we propose an algorithmic scheme that allows to automatically perform file saves by modication of the tangent linear and the cotangent linear codes, the saves are optimized for each routine.

This algorithm is used for the differentiation of Meso-NH with respect to the initial conditions. After the validation step we observe that the use of these linear codes is less costly in time and memory: a run of the linear codes only requires 3 times the time of one run of Meso-NH and the memory occupied is sufficiently small to run meteorological simulations with a physical meaning.

Key-words: Automatic differentiation, Odyssée 1.6, adjoint codes, trajectories saves, validation tests, Meso-NH 2.4.

Table des matières

1 Introduction

Meso-NH est le modèle atmosphérique Meso-échelle Non-Hydrostatique développé à Toulouse par le Centre National de Recherches en Météorologie (Météo-France) et le Laboratoire d'Aérologie (CNRS). Partiellement décrites dans la suite de ce document, ses caractéristiques sont développées dans les documentations scientifique [13] et algorithmique [14], et le guide d'utilisation[15].
Ce code récent simule des événements météorologiques très fins, sans être pour autant un code de prévision. Dans cet esprit, on peut envisager de le doter des codes dérivés associés qui permettront, par exemple, l'utilisation de méthodes variationnelles pour l'étude de sensibilités par rapport aux paramètres ou l'assimilation de données en vue de la prévision. D'autres utilisations du code adjoint sont possibles : couplage de modèles (hydrologie), introduction de modèles de pollution chimique ...
Avec Meso-NH en point de mire, l'objet de ce travail est une étude visant à déterminer la pertinence de l'utilisation de la différentiation automatique pour la dérivation de codes adjoints utilisables de manière opérationnelle, les concepts et outils décrits s'appliquant naturellement à la dérivation d'autres codes.

Tout comme Meso-NH, la plupart des schémas numériques actuels prennent en compte des équations non linéaires pour appréhender au plus près les phénomènes physiques qu'ils modélisent. Algorithmiquement représentées par un vecteur, les variables du modèle sont calculées via l'ensemble d'expressions symboliques (arithmétiques ou logiques) traduisant le système d'équations et constituant le code informatique.
Pour résoudre des problèmes d'optimisation sur de tels schémas, il est parfois intéressant, voire nécessaire, de connaître les dérivées successives de ces expressions : conception optimale, assimilation de données variationnelle, ..., un bon aperçu des applications et des méthodes de dérivation est présenté dans [1].

Obtenir les dérivées successives de schémas numériques codés informatiquement peut être réalisé de plusieurs manières. La première méthode, dérivation manuelle, est un peu trop fastidieuse pour être envisagée dans le cadre de ce travail.
La deuxième consiste en l'élimination des variables intermédiaires du programme, suivie de la dérivation symbolique des nouvelles expressions. Cette méthode, difficile à mener à terme, est coûteuse car les expressions élémentaires générées sont susceptibles d'être évaluées un grand nombre de fois.
Une alternative est d'utiliser une méthode de différences divisées i.e. d'approcher la dérivée du code dans une (ou plusieurs) direction par une formule de Taylor. L'avantage de cette méthode est de calculer une dérivée directionnelle en utilisant le code en "boîte noire" (aucune modification interne du code, aucune création d'autre code), mais son coût élevé, un calcul pour chaque dérivée directionnelle, et le manque de précision dû a l'approximation numérique, en font un outil d'une utilisation et d'une fiabilité parfois contestables.
Une autre solution est la différentiation automatique. Dans ce cas, le code est dérivé par un logiciel externe qui analyse le code original et écrit les codes dérivés dans le même langage informatique. Pour réussir cette opération, le logiciel utilise un ensemble de règles de dérivation simples et construit de manière systématique les expressions dérivées. Avec une telle approche, la dérivation est obtenue avec la précision machine. Les atouts de la dérivation automatique de codes par rapport aux autres méthodes ont conditionné notre choix pour la différentiation de Meso-NH.

Il existe de nombreux logiciels de différentiation automatique, chacun d'eux proposant divers modes de dérivation : mode tangent, calcul de dérivées d'ordre élevé, mode cotangent (adjoint). Suivant le logiciel, les codes acceptés sont écrits en : fortran 77, Fortran 90, C ou C++.
En fortran 77, le logiciel le plus connu est probablement Adifor [2]. Cet outil calcule les dérivées successives d'un code en mode tangent (linéarisation des expressions) mais ne calcule pas l'adjoint. De fait, Adifor n'est pas adapté à notre problématique de dérivation. Actuellement en cours de développement, le complément Adjifor, devrait dérivre en mode adjoint, est annoncé.
Également écrit pour des codes fortran 77, le logiciel Odyssée [6] permet de dériver les codes en modes tangent et cotangent. Comme la version actuelle (version 1.6) gère un code complet grâce à l'extraction du graphe fonctionnel et au suivi des variables de dérivation, Odyssée 1.6 peut dériver un code dans son entier. Les nouvelles facultés d'Odyssée 1.6 [6] en font un outil de dérivation automatique bien adapté à l'étude entreprise dans le cadre de ce rapport. Cependant, on notera que Meso-NH, écrit en Fortran 90, ne se prête pas directement à la dérivation par Odyssée.
Certes le dérivateur automatique Adol-F [10] (*produit dérivé* d'Adol-C [8]) opère sur des codes écrits en Fortran 90, mais il n'offre pas le suivi de variables et laisse à l'utilisateur le soin de gérer manuellement la propagation de la dérivation : pour chaque routine, le code doit être modifié afin d'indiquer les variables actives de dérivation.

En conclusion, l'argument de la correspondance des langages (Fortran 90 pour Adol-F et Meso-NH) nous est apparu mineur devant la possibilité offerte par Odyssée de dériver un code entier en une seule fois ; le choix d'Odyssée s'est donc tout naturellement imposé.
Des revues plus complètes sur les codes de dérivation automatique et leurs propriétés existent et pour exemple nous citons l'exposé *on-line* [17].

L'objectif de ce rapport étant d'étudier la différentiation automatique de Meso-NH (200 000 lignes), nous nous sommes orientés vers une utilisation systématique d'Odyssée. Mais l'utilisation d'Odyssée présente quelques inconvénients qui sont dus essentiellement à la taille des exécutables générés. En effet, en mode de dérivation cotangente, Odyssée recalcule, dans chaque routine, la trajectoire du modèle direct pour pouvoir évaluer la dérivée cotangente au point considéré, il en découle l'utilisation d'un grand nombre de variables intermédiaires. Pour éviter ces recalculs très coûteux pour des schémas itératifs, nous proposons d'effectuer systématiquement une sauvegarde de la trajectoire sur fichier : nous utilisons donc Odyssée en boîte noire pour la dérivation, puis complétons son action par insertion d'instructions de sauvegarde sur fichier dans les codes dérivés. Dans ce cas, la trajectoire de l'état direct est calculée et sauvée sur fichier dans le code tangent, puis lue et employée pour l'évaluation du code cotangent lors de son exécution.

L'étude entreprise est menée comme suit. En section 2, nous rappelons les principales fonctionnalités d'Odyssée. Décrites en section 3, deux méthodes de sauvegarde de la trajectoire sur fichier sont proposées. Les outils de validation des codes dérivés, tests du gradient et du produit scalaire, sont rappelés dans le chapitre 4 car, très coûteux en temps, ces tests n'en demeurent pas moins incontournables pour vérifier l'adjoint d'un code.
Les outils construits, nous décrivons rapidement le modèle atmosphérique Meso-NH et son algorithmique en section 5, pour finalement aborder la dérivation de la version adiabatique du modèle au chapitre 6. Le chapitre 7 est consacré aux résultats des différents tests numériques validant la linéarisation des codes dérivés. Des conclusions générales sur la dérivation automatique de codes et nos premières conclusions sur la différentiation de Meso-NH sont données au dernier chapitre.

2 Odyssée

Odyssée est le code de différentiation automatique conçu par le projet SAFIR (INRIA Sophia-Antipolis et UNSA). Écrit en CAML, il prend en entrée un code écrit en fortran 77 et construit deux codes dérivés, tangent et cotangent, compilables en fortran 77.
En un bref raccourci bibliographique, les documents [16], [3] et [7], rendent compte des évolutions d'Odyssée, la bibliographie complète peut être obtenue à partir du site Web de l'INRIA (http://www.inria.fr/). Dernièrement, Odyssée a été employé :

- pour dériver en mode adjoint le code Thyc-1D [3] (version 1D du code thermohydraulique tridimensionnel Thyc),
- pour contribuer à la résolution d'un problème d'optimisation de forme en aérodynamique [12].

L'étude menée est réalisée avec la version 1.6 d'Odyssée dont les principales caractéristiques sont exposées ci-dessous.

2.1 Définitions

On appelle routine informatique un morceau compilable de code constitué d'une entête, de déclarations de variables, d'expressions arithmétiques, d'expressions conditionnelles, d'appels à d'autres routines, de lignes de contrôle d'itérations et d'une instruction de fin.

Dans ce qui suit, on nomme *code direct*, le code original que l'on souhaite dériver. Il existe deux modes de différentiation :

- le mode direct ou tangent, le code dérivé produit est appelé *code linéaire tangent* et
- le mode inverse, adjoint ou cotangent, le code dérivé obtenu est dit *code linéaire cotangent*.

Par abus de langage nous sous-entendons le terme "linéaire" et les nommons plus simplement *code tangent* et *code cotangent*.
Ces deux modes de dérivation ont été et sont encore largement décrits dans de nombreux articles utilisant ou construisant ces logiciels de différentiation automatique, aussi nous n'aborderons pas réellement les techniques opérationnelles de différentiation pour nous concentrer sur l'aspect informatique des codes produits par Odyssée ; le lecteur pourra observer l'action de la dérivation sur les exemples proposés.

On appelle *variables d'entrée* d'une routine R, les arguments de la routine dont la valeur est connue pour l'exécution de cette routine. Les *variables d'entrée actives* sont les variables d'entrée par rapport auxquelles le code direct est dérivé.

On appelle *variables de sortie* d'une routine R, les arguments de la routine modifiés par exécution de celle-ci. On remarquera qu'une variable peut être à la fois variable d'entrée et de sortie d'une routine.

La dénomination *variables de contrôle* du problème s'appliquent aux variables actives du code direct et aux variables non locales dépendant de celles-ci.

2.2 Conventions

Lorsque l'on emploie Odyssée sur un code, les conventions d'écriture suivantes s'appliquent.

- Soit R la routine à dériver. Ses routines dérivées sont suffixées par TL ou CL suivant que le mode de dérivation tangent ou cotangent est appliqué i.e. on obtient les routines RTL et RCL.
- De même les variables de contrôle dérivées sont respectivement suffixées par TTL ou CCL suivant qu'elles apparaissent dans les codes tangent ou cotangent.

2.3 Atouts d'Odyssée 1.6

Les ajouts effectués aux versions précédentes améliorent de manière significative les performances du logiciel car ils évitent en grande partie les manipulations, sources d'erreurs, nécessaires à la dérivation routine par routine. En effet, Odyssée 1.6 autorise souvent le traitement d'un programme complet, notamment lorsque le graphe d'appel du code est un arbre. Dans ce cas, informatiquement fréquent, Odyssée peut dériver en une seule fois toutes les unités de ce programme grâce à deux nouvelles fonctions :

- l'analyse automatique des dépendances fonctionnelles,
- le suivi automatique des variables de contrôle dans le code.

Un autre apport fondamental par rapport aux versions antérieures d'Odyssée est la dérivation des routines de type fonction (*function*) ; pour mémoire, les précédentes versions ne proposaient que la dérivation des procédures (*subroutine*).

Remarque : Dans le cas d'une fonction, les routines dérivées sont de types procédure : Odyssée transforme la fonction en procédure en adjoignant une variable supplémentaire à sa liste d'arguments d'appel. Cette variable a le même nom que la fonction et contient le résultat de celle-ci.

2.4 Fonctionnalités d'Odyssée qui ont retenu notre attention

La plus importante de toutes est incontestablement la construction du code linéaire cotangent. En effet, ce mode de différentiation est très peu développé dans les autres logiciels de différentiation automatique; en particulier, le logiciel Adifor, le plus usité à notre connaissance, n'offre pas à ce jour, la possibilité de calculer le code linéaire cotangent bien que la version Adjifor devrait le faire.

2.4.1 Graphe de dépendance et Suivi des variables

Le graphe de dépendance d'un code est le graphe formé par tous les appels de routines : c'est une image algorithmique de la structure du programme qui est indissociable de l'analyse d'un code entier. Comme Odyssée 1.6 extrait et utilise le graphe du code, il est envisageable de dériver celui-ci comme un tout.

On notera qu'il est possible d'extraire le graphe d'un code incomplet ce qui laisse la possibilité à l'utilisateur de travailler avec des routines en boîte noire en les différentiant manuellement ou par une méthode de différences divisées (fonctions tabulées [3], par exemple).

Mais le graphe de dépendance n'est pas suffisant pour organiser la dérivation d'un code entier : il faut y ajouter un suivi algorithmique des variables de contrôle. Pour chaque variable, on connaît ainsi ses emplacements d'utilisation : routine d'affectation initiale, routine(s) de modification et routine(s) d'utilisation sans modification.

2.4.2 Split

Lorsque Odyssée travaille sur un code, il effectue un certain nombre de prétraitements nécessaires pour la construction de codes dérivés : pour exemple, Odyssée extrait les appels de fonctions avec arguments non terminaux et les affecte à des variables intermédiaires. Dans un premier temps, Odyssée procède à une réécriture du code en décomposant notamment les expressions arithmétiques en sous-expressions binaires [5], puis dérive celles-ci.

Remarque : Si l'option *minimal split* [5] est choisie, Odyssée recompose partiellement les expressions binaires dérivées ce qui facilite quelque peu la lecture des routines dérivées.

Les routines dérivées calculées avec ou sans l'option *minimal split* sont écrites différemment, mais produisent évidemment les mêmes résultats à l'exécution du code ; cette remarque est très importante lorsque l'on souhaite gérer une sauvegarde de la trajectoire sur fichier (en utilisant Odyssée en boîte noire).

2.4.3 Petits outils bien utiles

D'autres outils d'Odyssée ont également facilité l'implémentation des procédures de sauvegarde sur fichier et les procédures de validation pour les codes dérivés. Par exemple, la commande *getinout* d'Odyssée indique les variables actives d'une routine et précise leur rôle (variables actives d'entrée et/ou de sortie) ce qui est utile pour l'initialisation des variables dans les tests de validation.

2.4.4 Environnement

Odyssée propose à l'utilisateur deux manières de travailler :

- une interface graphique,
- un interpréteur de commande ;

tous deux sont d'utilisation agréable. La possibilité d'employer des fichiers d'exécution (**.batch*) automatise le chargement des fichiers contenant le code, le choix du mode de différentiation, des variables de contrôle et des options de dérivation ; le lecteur consultera la documentation livrée avec Odyssée pour de plus amples détails [6].

2.5 Les modes de dérivation, Sauvegarde de trajectoire

Définition : Pour un modèle et une initialisation donnés, la trajectoire partant de ce point initial est l'ensemble des valeurs prises par toutes les variables du modèle au cours de son exécution.

2.5.1 Trajectoire dans le code tangent

Le code linéaire tangent est construit en linéarisant les instructions du code direct par rapport aux variables de contrôle. On effectue ce calcul, instruction par instruction, en réalisant la linéarisation puis l'évaluation de chaque instruction : la dérivation en mode tangent proposée par Odyssée ne nécessite pas la connaissance de la trajectoire puisque le code direct est conservé dans le code tangent :

le code tangent produit par Odyssée contient trajectoire du code direct et expressions linéarisées.

Remarque : Il est classique de construire matriciellement le code tangent à partir de la matrice jacobienne dérivée par rapport aux variables d'entrée active du code.

2.5.2 Trajectoire dans le code cotangent

Le code cotangent est obtenu par transposition de la matrice jacobienne sus-citée ; une autre manière de décrire cela est de dire que le code cotangent est dérivé en commençant par les instructions finales de la routine que l'on étudie. À cause de cela, on ne peut pas mêlé le code direct aux lignes linéarisées pour obtenir la valeur des variables directes apparaissant dans code cotangent : il est nécessaire de protéger le contenu de ces variables. La méthode adoptée dans Odyssée est un recalcul de la trajectoire de la solution du modèle direct effectué au début de chaque routine.

Cependant, il n'est pas utile de sauver toute la trajectoire du modèle : il suffit de conserver les valeurs des variables directes utiles à l'évaluation du code cotangent telles que les variables apparaissant dans des expressions non linéaires ou les expressions conditionnelles.

Comme nous le verrons au chapitre 3, une alternative au recalcul de la trajectoire proposé par Odyssée, est une sauvegarde sur fichier des valeurs des variables directes impliquées dans l'évaluation du code cotangent.

Remarque : Les variables directes présentes dans le code cotangent sont les variables utilisées dans les expressions linéarisées et dans les instructions conditionnelles du code tangent.

2.6 Restrictions actuelles

Ce paragraphe expose les limitations d'Odyssée 1.6 indiquées dans la documentation ainsi que quelques observations personnelles. Pour une meilleure appréhension des problèmes que peuvent engendrer ces limitations, nous avons décidé de les classer suivant l'importance des changements à opérer sur les codes direct et dérivés.

2.6.1 Quelques Remarques

Lorsque l'on dérive une routine par rapport à une ou plusieurs de ces variables d'entrée, Odyssée calcule la dérivée de toutes les variables de sortie (associées à ces variables d'entrée) : cela peut engendrer un sur-coût de calcul, mais offre l'avantage de ne calculer "qu'un adjoint".

Dans la sauvegarde de la trajectoire en mode cotangent, la modification d'une partie d'un tableau de variables entraîne la sauvegarde de tout le tableau.

Les commentaires du code original sont effacés lors de la lecture par Odyssée, ils ne sont pas transmis aux codes dérivés.

2.6.2 Restrictions nécessitant des transformations de codes

Dans le code direct. Dans la plupart des cas, les limitations citées peuvent être levées en prenant garde à l'écriture du code direct, on veillera à respecter les règles suivantes :

- les modules de déclarations communs à plusieurs routines (*common*) doivent être identiquement déclarés dans toutes les routines du codes,

- pour favoriser la lisibilité des codes dérivés, les variables et les routines du code direct de nom finissant par TL ou CL sont à proscrire,
- les instructions de saut (*goto*) du code direct ne sont pas traités en mode inverse,
- les tableaux de taille variable ne sont pas reconnus,
- les tableaux de caractères figurant dans les arguments d'une routine peuvent provoquer des erreurs d'interprétation à la lecture du fichier par Odyssée.

Dans les codes dérivés. Les instructions de lecture et d'écriture (*read, write et format*) sont mises en commentaire dans le code direct lu par Odyssée, puis supprimés dans les codes dérivés : il est nécessaire de replacer ces instructions si l'on souhaite utiliser le code tangent produit par Odyssée comme code direct.

Par choix des développeurs d'Odyssée, la trajectoire est recalculée localement, mais lorsque des tableaux locaux sont nécessaires à la sauvegarde, les dimensions de ceux-ci sont fixées par l'intermédiaire de paramètres nommés "ODY*" et initialisés arbitrairement à 10 : l'utilisateur doit modifier ces paramètres.

Les fichiers inclus (*include*) sont reproduits dans les différents codes, cela ne perturbe pas les résultats mais peut nuire à la structuration du programme.

3 Sauvegarde de trajectoires

La seule procédure de sauvegarde de trajectoire offerte par Odyssée 1.6 consiste en un recalcul local de celle-ci. A priori, cette méthode ne convient pas à l'étude de gros codes informatiques et nous avons décidé d'employer le logiciel Odyssée en boîte noire pour la dérivation, puis d'implanter, dans les codes dérivés, une sauvegarde de la trajectoire sur fichier. Cette sauvegarde est automatisée via un ensemble de programmes écrits en Fortran 90.

La construction de la trajectoire sur fichier est essentiellement basée sur une utilisation du code tangent incluant le code direct (paragraphe 2.5.1) ; le travail supplémentaire de dérivation en mode tangent n'est pas vain puisque la validation du code cotangent réalisée par le test du produit scalaire emploie le code tangent (chapitre 4).

3.1 Sauvegarde de trajectoires dans Odyssée

Pour écrire le code cotangent d'une routine, Odyssée prend en entrée le programme direct et une liste de variables actives. Après dérivation, on observe que le code cotangent produit est composé :

- de déclarations de variables directes et dérivées,
- de déclarations de variables locales servant, entre autres, à :
  - la sauvegarde des valeurs des variables directes nommées SAVE,
  - la sauvegarde des valeurs des expressions conditionnelles nommées TEST,
- d'une partie "trajectoire" qui affecte les variables de sauvegarde SAVE et TEST et
- d'une partie contenant le code dérivé : ces expressions linéarisées utilisent les variables SAVE et TEST, appelées dans l'ordre inverse de leur création, ce qui permet l'évaluation du code cotangent.

Cette procédure systématique est visualisable sur la routine non linéaire XCARRE (Tab. 1), routine dont il n'est point nécessaire d'expliquer l'action. Par dérivation en X de la routine XCARRE, les routines tangente XCARRETL et cotangente XCARRECL produites par Odyssée sont :

TAB. 1 – *Routine XCARRE*

```
SUBROUTINE XCARRE (X, Y)

IMPLICIT NONE
REAL X, Y

Y = X**2     ! Jac(x^2) = 2x
END

SUBROUTINE XCARRETL (X, Y, XTTL, YTTL)

IMPLICIT NONE
REAL X, Y
REAL XTTL
REAL YTTL

YTTL = 0.
YTTL = 2*XTTL*X     ! y' = Jac.x' = 2x.x'
Y = X**2      ! trajectoire
END
```

```
SUBROUTINE XCARRECL (X, Y, XCCL,YCCL)

IMPLICIT NONE
REAL X, Y
REAL YCCL
REAL XCCL
REAL SAVE2
REAL SAVE1

C
C Trajectory      ! trajectoire !
C

SAVE1 = Y
SAVE2 = Y
Y = X**2      ! trajectoire
C
C Transposed linear forms
C

Y = SAVE2
XCCL = XCCL+YCCL*(2*X)      ! x' = t Jac. y' = 2x.y'
YCCL = 0.
YCCL = 0.
Y = SAVE1
END
```

Ce bref exemple est évidemment insuffisant pour apprécier les performances d'Odyssée à leurs justes valeurs.

Après utilisation sur des codes plus complexes, il nous apparaît qu'Odyssée est un outil précieux de dérivation en mode inverse. Cependant, la sauvegarde par recalcul de la trajectoire nous semble inadéquate pour la dérivation de codes industriels car les exécutables des codes cotangents produits sont coûteux par la mémoire vive que nécessite le stockage de la trajectoire.

Notre solution est donc de conserver les parties dérivées des codes fournis par Odyssée et de remplacer la procédure de recalcul de trajectoire par une sauvegarde des variables sur fichier.

La routine (Tab. 1) est très simple, et manipuler ses code dérivés pour y implanter une sauvegarde sur fichier est trivial. Mais la méthode manuelle atteint rapidement ses limites: modifier ainsi le code pour procéder à une sauvegarde sur fichier est inconcevable. Pour agir de manière systématique, nous avons construit des programmes analysant les codes dérivés et composant de nouveaux codes dérivés contenant la sauvegarde sur fichier.

3.2 Sauvegarde automatique de trajectoires

Lorsque le code direct est nonlinéaire, l'évaluation du code cotangent en un point donné requiert l'usage de la trajectoire produite par le code direct calculé en ce point: le code cotangent ne peut être évalué qu'après une exécution du code direct ou, plus astucieusement, qu'après une exécution du code linéaire tangent produit par Odyssée qui inclut le code direct.
L'intérêt d'utiliser le code tangent plutôt que le code direct tient au fait que les non-linéarités présentes dans le code cotangent sont aussi présentes dans le code tangent: pour connaître ponctuellement la valeur d'une telle variable "nonlinéaire" dans le code cotangent, il suffit de connaître la valeur qu'elle a, en ce même lieu, dans le code tangent.

Après construction de la trajectoire dans les fichiers dérivés et validation des codes, on peut supprimer les expressions linéarisées du fichier tangent pour obtenir un nouveau fichier direct contenant la trajectoire.

Remarque: Cette stratégie de détection des non-linéarités à partir du code tangent peut être utilisée pour optimiser la trajectoire locale des routines cotangentes produites par Odyssée, de sorte que seules les variables à caractère nonlinéaire soient affectées à des variables SAVE. En procédant ainsi, on limite considérablement le nombre de variables de sauvegarde, ce qui a pour effet de réduire la taille des exécutables des codes cotangents. Les procédures locales d'optimisation des sauvegardes sur fichier s'appliquent aussi à l'optimisation de la sauvegarde par variables de la partie trajectoire des codes cotangents d'Odyssée.

Dans cette section nous présentons le contenu de la sauvegarde puis, dans les paragraphes 3.2.2 et 3.2.3 nous décrivons deux manières d'insérer une sauvegarde sur fichier dans les codes dérivés par Odyssée.

3.2.1 Que sauve-t-on ?

Déterminer les valeurs des variables à préserver revient essentiellement à identifier puis extraire les parties non linéaires du code direct. En effet, pour évaluer un code cotangent, les variables à conserver sont les variables directes du code apparaissant dans une expression linéarisée et les variables apparaissant dans les instructions conditionnelles.

Remarque : La sauvegarde est réalisée sur des fichiers à accès direct.

Sauver la trajectoire est une chose, mais éviter la redondance de sauvegardes est mieux.
Pour respecter cette contrainte le plus souvent possible, on sauve les variables directes en tenant compte d'éventuelles modifications de celles-ci. En effet, si une variable n'a pas été évaluée depuis sa dernière écriture sur fichier, il n'est pas nécessaire de la sauver à nouveau.

Dans une routine donnée, les redondances peuvent être partiellement supprimées pendant la construction de la sauvegarde par la gestion d'un ensemble E_v de noms représentant les variables directes de la routine :

- si une variable directe est présente dans une expression linéarisée, on teste son appartenance à l'ensemble E_v : si elle n'est pas dans E_v, on ordonne son écriture sur fichier et on ajoute son nom à l'ensemble E_v.
- Si une variable directe est affectée dans la routine tangente, on retire le nom de cette variable à l'ensemble E_v.
- Les variables apparaissant dans les expressions conditionnelles sont sauvées systématiquement.

Pour simplifier l'exposé du schéma de sauvegarde, nous allons considérer le cas de variables scalaires puis le cas des variables de type tableau. Ce point de vue est important car les problèmes de redondance les plus difficiles à gérer proviennent de modifications partielles de tableaux.
Les cas particuliers telles que les expressions conditionnelles ou les boucles itératives sont traitées au paragraphe 3.3.

Sauvegarde des variables scalaires. Lorsque l'on veut sauver une variable scalaire, on applique simplement les règles définies ci-dessus. Dans le cas des expressions conditionnelles et des bornes des indices de boucles, on n'ajoute pas ces variables à l'ensemble E_v.

Remarque : Les variables contenues dans des expressions conditionnelles sont conservées car, bien qu'elles ne soient pas toujours dérivées par Odyssée, elles indiquent localement le traitement informatique à effectuer. Il en va de même pour les bornes des indices de boucle.

Sauvegarde pour les tableaux. Dans ce document, un tableau est considéré comme une seule variable, il est donc modifié aussitôt que l'une de ses valeurs est modifiée.

- Si le tableau est muni d'indice(s), alors on sauve la valeur du tableau correspondant aux indices, on ne change pas E_v car on n'a pas sauvé tout le tableau.
- L'absence d'indices (valable uniquement en Fortran 90) aux côtés d'une tableau signifie que l'on effectue globalement l'opération indiquée. Lorsque cette variable apparaît dans une expression linéarisée et que son nom est absent de E_v, on sauve globalement le tableau et on complète E_v.

Comme dans le cas des variables scalaires, on retire de E_v tout nom de tableau venant d'être modifiée (globalement ou partiellement).

3.2.2 Codes dérivés *full split*

Odyssée propose deux options de dérivation *full split* et *minimal split* qui produisent deux codes différent par la forme. Dans les deux cas, un prétraitement, décomposant le code en sous-expressions binaires et extraction des appels de fonctions, est appliqué sur le code direct, puis l'usage de l'option minimal split induit un post-traitement visant à réduire le nombre de variables intermédiaires, améliorant ainsi la lisibilité des codes dérivés. Les propriétés de l'option *minimal split* sont détaillées dans [5].
Le choix de l'une ou l'autre option impliquant des différences d'écriture dans les codes, cela rend caduque la méthode de sauvegarde présentée dans ce paragraphe pour les codes dérivés avec l'option *minimal split*. Cependant, nous avons jugé utile de conserver cette section, consacrée spécifiquement à l'option *full split*, car elle permet d'établir plus simplement la méthode de sauvegarde de la trajectoire sur fichier.

Lorsque Odyssée dérive avec l'option *full split*, il commence par réécrire le code direct en sous-expressions binaires. Pour illustrer cela, nous utilisons maintenant la routine SIN_CARRE_XYZ que l'on dérive par rapport à X, Y, Z.

TAB. 2 – *Routine SIN_CARRE_XYZ*

```
SUBROUTINE SIN_CARRE_XYZ (X, Y, Z)
IMPLICIT NONE
REAL X, Y, Z

Z = COS(X*Y*Z)
Z = 1-Z**2
END
```

La routine lue par Odyssée est transformée, puis dérivée (Tab. 3) par linéarisation de toutes les lignes suivant les règles classiques (consulter [7] par exemple).

TAB. 3 – *Routine SIN_CARRE_XYZ dérivée sous l'option full split*

```
SUBROUTINE SIN_CARRE_XYZ (X, Y, Z)

IMPLICIT NONE
REAL X, Y, Z
REAL SR01S

SR01S = X*Y*Z ! décomposition de COS(X*Y*Z)
Z = COS(SR01S)
Z = 1-Z**2
END



SUBROUTINE SIN_CARRE_XYZTL (X, Y, Z, XTTL, YTTL, ZTTL)

IMPLICIT NONE
REAL X, Y, Z
REAL SR01S
REAL XTTL
REAL YTTL
REAL ZTTL
REAL SR01S0
REAL SR01S0TTL
REAL SR01STTL

SR01S0TTL = 0.
SR01S0TTL = YTTL*X+Y*XTTL           ! dérivation de X*Y
SR01S0 = X*Y                        ! décomp. de X*Y*Z (1)
SR01STTL = ZTTL*SR01S0+Z*SR01S0TTL ! dérivation de (X*Y)*Z
SR01S = SR01S0*Z                    ! décomp. de X*Y*Z (2)
ZTTL = -SR01STTL*SIN(SR01S)         ! dérivation de COS
Z = COS(SR01S)                      ! décomp. de COS
SR01S0TTL = 0.
SR01S0TTL = -2*ZTTL*Z
SR01S0 = -Z**2                      ! décomp. 1-Z**2 (1)
ZTTL = SR01S0TTL
Z = 1+SR01S0                        ! décomp. 1-Z**2 (2)
END
```

```
SUBROUTINE SIN_CARRE_XYZCL (X, Y, Z, XCCL, YCCL, ZCCL)

IMPLICIT NONE
REAL X, Y, Z
REAL SR01S
REAL YCCL
REAL ZCCL
REAL XCCL
REAL SAVE5
REAL SAVE4
REAL SAVE3
REAL SR01S0
REAL SAVE2
REAL SR01S0CCL
REAL SAVE1
REAL SR01SCCL

C
C Initializations of local variables
C

SR01S0CCL = 0.
SR01SCCL = 0.
C
C Trajectory
C

SAVE1 = SR01S0
SR01S0 = X*Y
SAVE2 = SR01S
SR01S = SR01S0*Z
SAVE3 = Z
Z = COS(SR01S)
SAVE4 = SR01S0
SR01S0 = -Z**2
SAVE5 = Z
Z = 1+SR01S0
C
C Transposed linear forms
C

Z = SAVE5
SR01S0CCL = SR01S0CCL+ZCCL
ZCCL = 0.
SR01S0 = SAVE4
ZCCL = ZCCL-SR01S0CCL*(2*Z)
SR01S0CCL = 0.
SR01S0CCL = 0.
Z = SAVE3
SR01SCCL = SR01SCCL-ZCCL*SIN(SR01S)
ZCCL = 0.
SR01S = SAVE2
ZCCL = ZCCL+SR01SCCL*SR01S0
SR01S0CCL = SR01S0CCL+SR01SCCL*Z
SR01SCCL = 0.
SR01S0 = SAVE1
YCCL = YCCL+SR01S0CCL*X
XCCL = XCCL+SR01S0CCL*Y
SR01S0CCL = 0.
SR01S0CCL = 0.
END
```

Ce court exemple permet de vérifier que le code tangent est construit à partir de la matrice jacobienne associée au code direct. En multipliant la matrice jacobienne transposée par les variables dérivées de sorties de la routine directe, on obtient les expressions adjointes relatives aux variables d'entrée du code direct. Ainsi, les variables directes ayant un rôle nonlinéaire sont localisées en des positions identiques dans l'un et l'autre code dérivé.
L'utilisation de la matrice jacobienne est classique, et compte tenu des règles d'élaboration des codes tangent et cotangent, nous proposons d'utiliser successivement les codes tangent et cotangent pour mettre en œuvre la procédure de sauvegarde sur fichier suivante.

Algorithme

- Le code tangent est parcouru de bas en haut et de droite à gauche.
  On identifie toute variable directe apparaissant dans l'affectation d'une variable tangente. Pour chacune d'elle, on insère un ordre d'écriture sur fichier ; ces ordres sont placés entre une ligne linéarisée et un calcul du code direct (Tab. 4). Si plusieurs variables directes sont sur une même ligne, on commence par écrire la variable la plus à droite.

- Le code cotangent est parcouru de haut en bas (transposition des instructions).
  On commence par supprimer toutes les variables SAVE ainsi que la partie trajectoire calculée dans le cotangent. Puis, ligne par ligne,

  - on vérifie la présence d'une variable directe (unicité due à l'option *full split*)

- et on insère un ordre de lecture sur fichier pour cette variable (avant la ligne considérée).

Cette procédure construit les codes dérivés (Tab. 4).

Dans ces codes, les ordres de lecture et d'écriture (*write, read*) dans les fichiers sont écrits en gras et sont précédés d'un indice (INDICE_FICHIER) utilisé pour l'indexation des blocs de données dans le fichier à accès direct contenant les sauvegardes, l'indice de parcours du fichier est une variable globale gérée via le module "modd_adjoint".

TAB. 4 – *Routines SIN_CARRE_XYZ dérivées, sauvegarde sur fichier (full split)*

```
SUBROUTINE SINCARRE_XYZTL (X, Y, Z, )
:                          XTTL, YTTL, ZTTL)
USE modd_adjoint

IMPLICIT NONE
REAL X, Y, Z
REAL SR01S
REAL XTTL
REAL YTTL
REAL ZTTL
REAL SR01S0
REAL SR01S0TTL
REAL SR01STTL

SR01S0TTL = 0.
SR01S0TTL = YTTL*X+Y*XTTL
write(6,REC=INDICE_FICHIER) Y
INDICE_FICHIER = INDICE_FICHIER + 1
write(6,REC=INDICE_FICHIER) X
INDICE_FICHIER = INDICE_FICHIER + 1
SR01S0 = X*Y
SR01STTL = ZTTL*SR01S0+Z*SR01S0TTL
write(6,REC=INDICE_FICHIER) Z
INDICE_FICHIER = INDICE_FICHIER + 1
write(6,REC=INDICE_FICHIER) SR01S0
INDICE_FICHIER = INDICE_FICHIER + 1
SR01S = SR01S0*Z
ZTTL = -SR01STTL*SIN(SR01S)
write(6,REC=INDICE_FICHIER) SR01S
INDICE_FICHIER = INDICE_FICHIER + 1
Z = COS(SR01S)
SR01S0TTL = 0.
SR01S0TTL = -2*ZTTL*Z
write(6,REC=INDICE_FICHIER) Z
INDICE_FICHIER = INDICE_FICHIER + 1
SR01S0 = -Z**2
ZTTL = SR01S0TTL
Z = 1+SR01S0
END
```

```
SUBROUTINE SINCARRE_XYZCL (X, Y, Z,
:                          XCCL, YCCL, ZCCL)
USE modd_adjoint

IMPLICIT NONE
REAL X, Y, Z
REAL SR01S
REAL YCCL
REAL ZCCL
REAL XCCL
REAL SR01S0
REAL SR01S0CCL
REAL SR01SCCL

C
C Initializations of local variables
C

SR01S0CCL = 0.
SR01SCCL = 0.
C
C Transposed linear forms
C

SR01S0CCL = SR01S0CCL+ZCCL
ZCCL = 0.
read(6,REC=INDICE_FICHIER) Z
INDICE_FICHIER = INDICE_FICHIER - 1
ZCCL = ZCCL-SR01S0CCL*(2*Z)
SR01S0CCL = 0.
SR01S0CCL = 0.
read(6,REC=INDICE_FICHIER) SR01S
INDICE_FICHIER = INDICE_FICHIER - 1
SR01SCCL = SR01SCCL-ZCCL*SIN(SR01S)
ZCCL = 0.
read(6,REC=INDICE_FICHIER) SR01S0
INDICE_FICHIER = INDICE_FICHIER - 1
ZCCL = ZCCL+SR01SCCL*SR01S0
read(6,REC=INDICE_FICHIER) Z
INDICE_FICHIER = INDICE_FICHIER - 1
SR01S0CCL = SR01S0CCL+SR01SCCL*Z
SR01SCCL = 0.
read(6,REC=INDICE_FICHIER) X
INDICE_FICHIER = INDICE_FICHIER - 1
YCCL = YCCL+SR01S0CCL*X
read(6,REC=INDICE_FICHIER) Y
INDICE_FICHIER = INDICE_FICHIER - 1
XCCL = XCCL+SR01S0CCL*Y
SR01S0CCL = 0.
SR01S0CCL = 0.
END
```

Après validation du code par les procédures décrites au chapitre 4, on élimine les lignes linéarisées pour ne conserver du fichier tangent que le code direct et les ordres d'écriture : il n'y a donc pas de sur-coût lié à l'évaluation du code tangent lors des simulations.

Cette méthode de sauvegarde n'est pas tout à fait satisfaisante puisqu'elle tient fortement compte de la structure des codes tangent et cotangent (dérivés avec l'option *full split*). Afin de lever cette limitation, nous proposons une deuxième stratégie de sauvegarde de trajectoires utilisable, a priori, avec n'importe quel dérivateur automatique.

3.2.3 Codes dérivés *minimal split*, Ajout de trace

Nous travaillons maintenant avec l'option *minimal split* et Odyssée construit les programmes dérivés indiqués dans la table (Tab. 5). Ces codes ont une structure bien différente des codes précédents (Tab. 3), et l'inventaire des variables directes dans les différentes routines varie. En effet, le relevé des noms des variables directes dans les lignes linéarisées de chacun des deux codes dérivés donne

- Z, SR01S, X, Y, Z, X, Y, pour le code tangent lu de bas en haut et de droite à gauche et
- Z, SR01S, Y, X, X, Z, Y, Z, pour le code cotangent lu de haut en bas et de gauche à droite.

Ces listes ordonnées diffèrent de par la position et le nombre d'occurences des variables, il n'est donc plus possible d'utiliser la stratégie du paragraphe précédent.

Remarque : Cette différence provient du post-traitement recombinant les expressions binaires induit par l'usage de l'option *minimal split*.

TAB. 5 – *SIN_CARRE_XYZ dérivée avec l'option minimal split*

```
SUBROUTINE SIN_CARRE_XYZ (X, Y, Z)

IMPLICIT NONE
REAL X, Y, Z
REAL SR01S

SR01S = X*Y*Z
Z = COS(SR01S)
Z = 1-Z**2
END



SUBROUTINE SIN_CARRE_XYZTL (X, Y, Z, XTTL, YTTL, ZTTL)

IMPLICIT NONE
REAL X, Y, Z
REAL SR01S
REAL XTTL
REAL YTTL
REAL ZTTL
REAL SR01STTL

SR01STTL = 0.
SR01STTL = ZTTL*Y*X+Z*(YTTL*X+Y*XTTTL)
SR01S = X*Y*Z
ZTTL = -SR01STTL*SIN(SR01S)
Z = COS(SR01S)
ZTTL = -2*ZTTL*Z
Z = 1-Z**2
END
```

```
SUBROUTINE SIN_CARRE_XYZCL (X, Y, Z, XCCL, YCCCL, ZCCL)

IMPLICIT NONE
REAL X, Y, Z
REAL SR01S
REAL YCCL
REAL ZCCL
REAL XCCL
REAL SAVE3
REAL SAVE2
REAL SAVE1
REAL SR01SCCL

C
C Initializations of local variables
C

SR01SCCL = 0.
C
C Trajectory
C

SAVE1 = SR01S
SR01S = X*Y*Z
SAVE2 = Z
Z = COS(SR01S)
SAVE3 = Z
Z = 1-Z**2
C
C Transposed linear forms
C

Z = SAVE3
ZCCL = -ZCCL*(2*Z)
Z = SAVE2
SR01SCCL = SR01SCCL-ZCCL*SIN(SR01S)
ZCCL = 0.
SR01S = SAVE1
ZCCL = ZCCL+SR01SCCL*(Y*X)
YCCL = YCCL+SR01SCCL*(X*Z)
XCCL = XCCL+SR01SCCL*(Y*Z)
SR01SCCL = 0.
SR01SCCL = 0.
END
```

Pour procéder à une sauvegarde de trajectoire sur de tels codes, et plus généralement lors d'une utilisation en boîte noire d'un différentiateur automatique, il est important de localiser les variables dans les codes. À cette fin, on introduit le concept de trace.

Définition : Un *marqueur* est une instruction élémentaire indépendante des instructions du code direct.

Dans la suite de ce document, le marqueur est l'instruction élémentaire

TAB. 6 – *Marqueur*

```
IDOPT = IDOPT**2
```

qui, par dérivation, induit les lignes de code suivantes

TAB. 7 – *Marqueurs dérivés*

mode tangent	mode cotangent
IDOPTTTL = IDOPTTTL*(2*IDOPT) IDOPT = IDOPT**2	IDOPTCCL = IDOPTCCL*(2*IDOPT)

Remarque : Nous verrons au paragraphe 3.3 qu'il est préférable d'introduire des marqueurs distincts deux à deux.

Définition : La *trace* est l'ensemble des marqueurs introduits entre les lignes du code direct ; elle est effacée des codes dérivés après construction de la sauvegarde de trajectoire sur fichier.

Le code direct ainsi transformé mêle deux programmes indépendants : le code original et la trace. Dans son action, le différentiateur automatique analyse les instructions les unes après les autres et les linéarise en conservant les positions des instructions dérivées de la trace par rapport aux instructions dérivées du code direct original. Après dérivation de l'ensemble des lignes, la présence des marqueurs facilite l'identification des lignes des codes tangent et cotangent correspondant à une même ligne du code direct.

Pour appliquer la stratégie de trace sur l'exemple précédent (Tab. 2), on ajoute des marqueurs dans la routine SIN_CARRE_XYZ et la variable IDOPT aux arguments de la routine afin de pouvoir dériver la trace. Après ces modifications, la routine s'écrit :

TAB. 8 – *Ajout de la trace dans SIN_CARRE_XYZ*

```
SUBROUTINE SIN_CARRE_XYZ (X, Y, Z, IDOPT)

IMPLICIT NONE
REAL X, Y, Z
INTEGER IDOPT
REAL SR01S

IDOPT = IDOPT**2
Z = COS(X*Y*Z)      Instruction 1
IDOPT = IDOPT**2
Z = 1-Z**2          Instruction 2
IDOPT = IDOPT**2
END
```

En dérivant SIN_CARRE_XYZ par rapport au variables X, Y ,Z , IDOPT, puis en supprimant la trajectoire du code cotangent, on obtient les lignes de codes suivantes. Pour simplifier la lecture, nous annotons les lignes

- Instruction 1, pour la première instruction du code direct, ...
- I_tl, pour les instructions tangentes,
- I_cl, pour les instructions cotangentes.

On applique l'algorithme suivant sur les codes dérivés produits par le dérivateur automatique (Tab. 9).

TAB. 9 – *Dérivation de la trace et du code direct*

```
SUBROUTINE SIN_CARRE_XYZ (X, Y, Z, IDOPT)


IMPLICIT NONE
REAL X, Y, Z
INTEGER IDOPT
REAL SR01S

IDOPT = IDOPT**2
SR01S = X*Y*Z       Instruction 1.1
Z = COS(SR01S)      Instruction 1.2
IDOPT = IDOPT**2
Z = 1-Z**2          Instruction 2
IDOPT = IDOPT**2
END

SUBROUTINE SIN_CARRE_XYZTL (X, Y, Z, IDOPT,
:  XTTL, YTTL, ZTTL, IDOPTTTL)

IMPLICIT NONE
REAL X, Y, Z
INTEGER IDOPT
REAL SR01S
REAL XTTL
REAL YTTL
REAL ZTTL
INTEGER IDOPTTTL
REAL SR01STTL

IDOPTTTL = 2*IDOPTTTL*IDOPT
IDOPT = IDOPT**2
SR01STTL = 0.
SR01STTL = ZTTL*Y*X+Z*(YTTL*X+Y*XTTL)  I_tl 1.1
SR01S = X*Y*Z       Instruction 1.1
ZTTL = -SR01STTL*SIN(SR01S)                  I_tl 1.2
Z = COS(SR01S)      Instruction 1.2
IDOPTTTL = 2*IDOPTTTL*IDOPT
IDOPT = IDOPT**2
ZTTL = -2*ZTTL*Z                             I_tl 2
Z = 1-Z**2          Instruction 2
IDOPTTTL = 2*IDOPTTTL*IDOPT
IDOPT = IDOPT**2
END
```

```
SUBROUTINE SIN_CARRE_XYZCL (X, Y, Z, IDOPT,
:  XCCL, YCCL, ZCCL, IDOPTCCL)

IMPLICIT NONE
REAL X, Y, Z
INTEGER IDOPT
REAL SR01S
REAL YCCL
REAL ZCCL
INTEGER IDOPTCCL
REAL XCCL
REAL SR01SCCL

C
C Initializations of local variables
C

SR01SCCL = 0.
C
C Transposed linear forms
C

IDOPTCCL = IDOPTCCL*(2*IDOPT)
ZCCL = -ZCCL*(2*Z)                           I_cl 2
IDOPTCCL = IDOPTCCL*(2*IDOPT)
SR01SCCL = SR01SCCL-ZCCL*SIN(SR01S)  I_cl 1.2
ZCCL = 0.
ZCCL = ZCCL+SR01SCCL*(Y*X)             I_cl 1.1.1
YCCL = YCCL+SR01SCCL*(X*Z)             I_cl 1.1.2
XCCL = XCCL+SR01SCCL*(Y*Z)             I_cl 1.1.3
SR01SCCL = 0.
SR01SCCL = 0.
IDOPTCCL = IDOPTCCL*(2*IDOPT)
END
```

Algorithme de sauvegarde par ajout de trace

- On ajoute la trace dans le code direct, puis on dérive avec Odyssée.
- On parcourt les fichiers tangent et cotangent.
- Pour chaque routine du code tangent on identifie la routine cotangente correspondante :
  - On étudie chaque bloc de lignes contenu entre deux marqueurs dérivés en partant du dernier dans le code tangent et du premier dans le code cotangent. Les lignes retenues dans les deux fichiers correspondent ainsi à la même instruction du code direct.
  - Du groupe d'instructions tangentes, on extrait le nom des variables directes présentes pour constituer l'ensemble E_v défini au paragraphe 3.2.1 : on ne retient qu'une occurrence par variable.
  - Comme indiqué au paragraphe 3.2.2, on écrit dans les fichiers, les ordres d'écriture (fichier tangent) et les ordres de lecture (fichier cotangent) pour chacune de ces variables.
    Ceci est matérialisé en caractères gras dans l'exemple (Tab. 10) : écriture entre lignes linéarisées et lignes directes du tangent, lecture avant les lignes du cotangent.
- On supprime la trace des fichiers.

En appliquant cette méthode sur l'exemple courant, on obtient les routines (Tab.10).

TAB. 10 – *Routines dérivées munies d'une sauvegarde sur fichier*

```
SUBROUTINE SINCARRE_XYZ (X, Y, Z, IDOPT)

IMPLICIT NONE
INTEGER IDOPT
REAL X, Y, Z
REAL SR01S

IDOPT = IDOPT**2
SR01S = X*Y*Z
Z = COS(SR01S)
IDOPT = IDOPT**2
Z = 1-Z**2
IDOPT = IDOPT**2
END


SUBROUTINE SINCARRE_XYZTL (X, Y, Z,
: XTTL, YTTL, ZTTL)
USE modd_adjoint

IMPLICIT NONE
REAL X, Y, Z
REAL SR01S
REAL XTTL
REAL YTTL
REAL ZTTL
REAL SR01STTL

SR01STTL = 0.
SR01STTL = ZTTL*Y*X+Z*(YTTL*X+Y*XTTL)
INDICE_FICHIER = INDICE_FICHIER + 1
write(6,REC=INDICE_FICHIER) Y
INDICE_FICHIER = INDICE_FICHIER + 1
write(6,REC=INDICE_FICHIER) X
INDICE_FICHIER = INDICE_FICHIER + 1
write(6,REC=INDICE_FICHIER) Z
SR01S = X*Y*Z
ZTTL = -SR01STTL*SIN(SR01S)
INDICE_FICHIER = INDICE_FICHIER + 1
write(6,REC=INDICE_FICHIER) SR01S
Z = COS(SR01S)
ZTTL = -2*ZTTL*Z
INDICE_FICHIER = INDICE_FICHIER + 1
write(6,REC=INDICE_FICHIER) Z
Z = 1-Z**2
END
```

```
SUBROUTINE SINCARRE_XYZCL (X, Y, Z,
: XCCL, YCCL, ZCCL)
USE modd_adjoint
IMPLICIT NONE
REAL X, Y, Z
REAL SR01S
REAL YCCL
REAL ZCCL
REAL XCCL
REAL SR01SCCL
C
C Initializations of local variables
C
SR01SCCL = 0.
C
C Transposed linear forms
C
read(6,REC=INDICE_FICHIER) Z
INDICE_FICHIER = INDICE_FICHIER - 1
ZCCL = -ZCCL*(2*Z)
read(6,REC=INDICE_FICHIER) SR01S
INDICE_FICHIER = INDICE_FICHIER - 1
read(6,REC=INDICE_FICHIER) Z
INDICE_FICHIER = INDICE_FICHIER - 1
read(6,REC=INDICE_FICHIER) X
INDICE_FICHIER = INDICE_FICHIER - 1
read(6,REC=INDICE_FICHIER) Y
INDICE_FICHIER = INDICE_FICHIER - 1
SR01SCCL = SR01SCCL-ZCCL*SIN(SR01S)
ZCCL = 0.
ZCCL = ZCCL+SR01SCCL*(Y*X)
YCCL = YCCL+SR01SCCL*(X*Z)
XCCL = XCCL+SR01SCCL*(Y*Z)
SR01SCCL = 0.
SR01SCCL = 0.
END
```

Les cas exposés ne mentionnent ni boucles, ni instructions conditionnelles ; cet oubli volontaire est réparé dans le paragraphe suivant.

3.3 Amélioration de la sauvegarde, Cas particuliers

La sauvegarde de la trajectoire par ajout de marqueurs identiques (Tab. 6) implique un *accès séquentiel* aux instructions lors de l'identification des blocs d'instructions tangentes et cotangentes correspondant à une même instruction du code direct. En effet, les marqueurs que nous avons utilisés distinguent les instructions les unes par rapport aux autres sans toutefois les localiser dans le code : cette manière de procéder s'est révélée de

maintenance difficile dans le cas d'instructions conditionnelles.

Pour remédier à ce problème, on définit les marqueurs ($n \in IN \backslash \{0,1\}$)

TAB. 11 – *Marqueur assurant l'unicité*

```
| IDOPT = IDOPT**n |
```

Tous différents, ils autorisent un *accès direct* aux instructions puisqu'on identifie facilement les instructions dérivées obtenues d'une même instruction directe.

3.3.1 Boucle DO

La sauvegarde expliquée dans les paragraphes précédents ne convient pas directement lorsque le code possède des boucles de calculs. En effet, l'évaluation des instructions contenues dans une boucle est conditionnée par la valeur de ses bornes. Puisque ni le logiciel Odyssée, ni nos programmes n'évaluent le code, il n'est pas possible de prévoir le parcours de celle-ci. Par exemple, les instructions suivantes (Tab. 12) sont dérivées en X de manière identique bien que le deuxième bloc d'instructions ne soit pas jamais parcouru.

TAB. 12 – *Des boucles DO*

```
BORNE = 10
...
Y(1) = X(1)**3
DO I= 0,BORNE
Y(I) = X(I)**2
ENDDO
```

```
BORNE = -10

Y(1) = X(1)**3
DO I= 0,BORNE
Y(I) = X(I)**2
ENDDO
```

Si l'on écrit les blocs dérivés et que l'on applique la procédure de trace décrite précédemment, on obtient les procédures

TAB. 13 – *Sauvegarde erronée dans une boucle DO*

```
SUBROUTINE TEST_DOTL (X, Y, XTTL, YTTL)
USE modd_adjoint

INTEGER N, BORNE
PARAMETER (N = 10, BORNE = 5)
INTEGER I
REAL X(0:N), Y(0:N)
DIMENSION XTTL(0:N)
DIMENSION YTTL(0:N)

...
YTTL(1) = 3*XTTL(1)*X(1)**2
Y(1) = X(1)**3
DO I = 1, BORNE
  YTTL(I) = 2*XTTL(I)*X(I)
  write(6,REC=INDICE_FICHIER) X(I)
  INDICE_FICHIER = INDICE_FICHIER + 1
  Y(I) = X(I)**2
END DO
...
```

```
SUBROUTINE TEST_DOCL (X, Y, XCCL, YCCL)
USE modd_adjoint

INTEGER N, BORNE
PARAMETER (N = 10, BORNE = 5)
INTEGER I
REAL X(0:N), Y(0:N)
DIMENSION XCCL(0:N)
DIMENSION XCCL(0:N)

...
DO I = BORNE, 1, -1
  read(6,REC=INDICE_FICHIER) X(I)
  INDICE_FICHIER = INDICE_FICHIER - 1
  XCCL(I) = XCCL(I)+YCCL(I)*(2*X(I))
  YCCL(I) = 0.
END DO
XCCL(1) = XCCL(1)+YCCL(1)*(3*X(1)**2)
...
```

Dans ces routines, on remarque que la valeur de **X** est inconnue si on n'a pas emprunté la boucle DO : la valeur de **X(1)** est alors erronée : la méthode de gestion de l'ensemble E_v des occurrences des variables telle que décrite au chapitre 3 ne s'applique pas. Dans ce cas, on procède de manière un peu différente.

Algorithme de sauvegarde dans les boucles

On a rencontré une variable directe dans une instruction linéaire tangente (i.e. entre les lignes DO-ENDDO, ou DO-CONTINUE) :

- si cette variable n'appartient pas à E_v à l'entrée de la boucle, alors on la sauve hors de la boucle, et on ajoute son nom à E_v. De plus, si la variable est modifiée dans la boucle alors on sauve localement.

- Si la variable appartient à E_v, on sauve celle-ci uniquement si elle est modifiée dans la boucle.

Appliquant cet algorithme sur l'exemple (Tab. 13), il découle une sauvegarde convenable de la trajectoire (Tab. 14) qui est valable quelque soit la valeur de BORNE.

TAB. 14 – *Sauvegarde correcte dans une boucle DO*

```
SUBROUTINE TEST_DOTL (X, Y, XTTL, YTTL)

...
YTTL(1) = 3*XTTL(1)*X(1)**2
Y(1) = X(1)**3
DO I = 1, BORNE
  YTTL(I) = 2*XTTL(I)*X(I)
  Y(I) = X(I)**2
END DO
write(6,REC=INDICE_FICHIER) X
INDICE_FICHIER = INDICE_FICHIER + 1
...
```

```
SUBROUTINE TEST_DOCL (X, Y, XCCL, YCCL)

...
read(6,REC=INDICE_FICHIER) BORNE
INDICE_FICHIER = INDICE_FICHIER - 1
DO I = BORNE, 1, -1
  XCCL(I) = XCCL(I)+YCCL(I)*(2*X(I))
  YCCL(I) = 0.
END DO
XCCL(1) = XCCL(1)+YCCL(1)*(3*X(1)**2)

...
```

Remarque : On pourrait envisager d'analyser la cohérence des bornes de la boucle d'instructions lorsque les bornes sont des nombres entiers ou des paramètres du code.

3.3.2 Expression conditionnelle IF-THEN-ELSE

Pour illustrer nos propos, nous considérons la routine direct TEST_IF qui, pour simplifier l'exposé, appelle 5 routines nommées INSTRUCTION1, ..., INSTRUCTION5 représentant des blocs instructions (linéaires ou non).

TAB. 15 – *Ordonnancement des instructions directes dans un IF-THEN-ELSE*

```
SUBROUTINE TEST_IF (X, Y)

REAL X, Y
LOGICAL CONDITION

CALL INSTRUCTION1(X, Y)
IF (CONDITION) THEN
  CALL INSTRUCTION2(X, Y)
  CALL INSTRUCTION3(X, Y)
ELSE
  CALL INSTRUCTION4(X, Y)
END IF
CALL INSTRUCTION5(X, Y)
END
```

Après dérivation (Tab. 16) de la routine TEST_IF, et plus généralement de routines des expressions conditionnelles IF-THEN-ELSE, on remarque deux sources de problèmes :

- l'ordonnancement des instructions au sein des routine dérivées peut changé,

- la détermination de la branche par laquelle on est passé ;

toutes deux sont susceptibles de fausser la sauvegarde de trajectoires exposée au chapitre 3.

TAB. 16 – *Ordonnancement des instructions dérivées dans un IF-THEN-ELSE*

```
SUBROUTINE TEST_IFTL (X, Y, XTTL, YTTL)

REAL X, Y
LOGICAL CONDITION
EXTERNAL INSTRUCTION5TL
EXTERNAL INSTRUCTION1TL
EXTERNAL INSTRUCTION2TL
EXTERNAL INSTRUCTION3TL
EXTERNAL INSTRUCTION4TL

YTTL = 0.
CALL INSTRUCTION1TL(X, Y, XTTL, YTTL)
IF (CONDITION) THEN
  CALL INSTRUCTION2TL(X, Y, XTTL, YTTL)
  CALL INSTRUCTION3TL(X, Y, XTTL, YTTL)
ELSE
  CALL INSTRUCTION4TL(X, Y, XTTL, YTTL)
END IF
CALL INSTRUCTION5TL(X, Y, XTTL, YTTL)
END
```

```
SUBROUTINE TEST_IFCL (X, Y, XCCL, YCCL)

REAL X, Y
LOGICAL CONDITION
EXTERNAL INSTRUCTION5CL
EXTERNAL INSTRUCTION1CL
EXTERNAL INSTRUCTION2CL
EXTERNAL INSTRUCTION3CL
EXTERNAL INSTRUCTION4CL

CALL INSTRUCTION5CL(X, Y, XCCL, YCCL)
IF (CONDITION) THEN
  CALL INSTRUCTION3CL(X, Y, XCCL, YCCL)
  CALL INSTRUCTION2CL(X, Y, XCCL, YCCL)
ELSE
  CALL INSTRUCTION4CL(X, Y, XCCL, YCCL)
END IF
CALL INSTRUCTION1CL(X, Y, XCCL, YCCL)
YCCL = 0.
END
```

Ordonnancement des instructions. La routine tangente reprend l'ordre des instructions de la routine directe, par contre l'ordre des instructions est vraiment différent dans la routine cotangente : ce n'est plus la classique inversion des instructions.

Pour remédier à cela , deux solutions simples sont envisageables :

- soit on utilise une trace avec les marqueurs (Tab. 11) qui favorisent un accès direct à chaque instruction,
- soit on considère l'expression IF-THEN-ELSE comme une routine interne. Dans ce cas, on peut garder l'accès séquentiel dans le fichier car on a bien correspondance (Tab. 17) entre appels de "routines" de profondeur 1 dans les codes tangent et cotangent.

TAB. 17 – *Ordonnancement des appels de "routines" de profondeur 1*

```
YTTL = 0.                                    | CALL INSTRUCTION5CL(X, Y, XCCL, YCCL)
CALL INSTRUCTION1TL(X, Y, XTTL, YTTL)        | "routine IF-THEN-ELSE"
"routine IF-THEN-ELSE"                       | CALL INSTRUCTION1CL(X, Y, XCCL, YCCL)
CALL INSTRUCTION5TL(X, Y, XTTL, YTTL)        | YCCL = 0.
```

Dans ce cas, la gestion de la trajectoire sur fichier se fait en reversant localement les instructions placées entre les mots IF et ELSE (noté IF-ELSE), puis ELSE-ENDIF.

Sauvegarde dans une instruction IF-THEN-ELSE. Maintenant que nous savons à nouveau associer lignes tangentes et cotangentes, il reste à gérer la sauvegarde en essayant d'assurer une certaine optimalité de celle-ci. Comme précédemment, on utilise l'ensemble E_v des noms de variables, et une solution au problème d'optimalité est de considérer deux ensembles contenant les noms de variables appelés E_v^{IF} et E_v^{ELSE} pour les instructions situées respectivement entre les mots IF-ELSE et ELSE-ENDIF. Avec ces notations, on gère E_v de la manière suivante.

Algorithme de sauvegarde dans une séquence IF-THEN-ELSE

Soit E_v l'ensemble de noms de variables avant l'expression conditionnelle :

- on impose

  $E_v^{\text{IF}} = E_v$ et
  $E_v^{\text{ELSE}} = E_v$

  à l'entrée de la séquence IF-THEN-ELSE,
- on gère E_v^{IF} dans la partie IF-ELSE et E_v^{ELSE} dans la partie ELSE-ENDIF.
- À la sortie du bloc d'instructions conditionnées par la séquence IF-THEN-ELSE, on impose :

  $E_v = E_v^{\text{IF}} \cap E_v^{\text{ELSE}}$.

Ces deux procédés sont à mener de façon récursive dans le cas d'expressions conditionnelles imbriquées.

Remarque : Si on traite seulement une instruction conditionnelle du type IF-ENDIF, on conserve l'ensemble E_v d'entrée car, comme dans le cas des boucles DO, il est impossible de prévoir le parcours des instructions soumises à condition.

3.4 Optimalité de la sauvegarde

La sauvegarde proposée n'est pas optimale. C'est notamment le cas de la routine WZCARRE (Tab. 18) qui appelle deux fois la routine nonlinéaire XCARRE.

TAB. 18 – *Défaut d'optimalité en X : routines directes*

```
SUBROUTINE WZCARRETL (X, W, Z)

IMPLICIT NONE
REAL X, W, Z

CALL XCARRE(X, W)
CALL XCARRE(X, Z)
END
```

```
SUBROUTINE XCARRETL (X, Y)

IMPLICIT NONE
REAL X, Y

Y = X**2
END
```

Après ajout de trace, dérivation en X et IDOPT, et construction de la sauvegarde sur fichier, on obtient les routines :

TAB. 19 – *Défaut d'optimalité en X : routines tangentes*

```
SUBROUTINE WZCARRETL (X, W, Z,
:                     XTTL, WTTL, ZTTL)
USE modd_adjoint

IMPLICIT NONE
REAL X, W, Z
REAL XTTL
REAL WTTL
REAL XTTL
EXTERNAL XCARRETL

WTTL = 0.
CALL XCARRETL(X, W, XTTL, WTTL)
CALL XCARRETL(X, Z, XTTL, ZTTL)
END
```

```
SUBROUTINE XCARRETL (X, Y,
:                    XTTL, YTTL)
USE modd_adjoint

IMPLICIT NONE
REAL X, Y
REAL XTTL
REAL YTTL

YTTL = 0.
YTTL = 2*XTTL*X
INDICE_FICHIER = INDICE_FICHIER + 1
write(6,REC=INDICE_FICHIER) X
Y = X**2
END
```

TAB. 20 – *Défaut d'optimalité en X : routines cotangentes*

```
SUBROUTINE WZCARRECL (X, W, Z,
:                     XCCL, WCCL, ZCCL)
USE modd_adjoint

IMPLICIT NONE
REAL X, W, Z
REAL XCCL
REAL WCCL
REAL XCCL
EXTERNAL XCARRECL

CALL XCARRECL(X, Z, XCCL, WCCL)
ZCCL = 0.
CALL XCARRECL(X, W, XCCL, ZCCL)
WCCL = 0.
END
```

```
SUBROUTINE XCARRECL (X, Y,
:                    XCCL, YCCL)
USE modd_adjoint

IMPLICIT NONE
REAL X, Y
REAL XCCL
REAL YCCL

read(6,REC=INDICE_FICHIER) X
INDICE_FICHIER = INDICE_FICHIER - 1
XCCL = XCCL+YCCL*(2*X)
YCCL = 0.
YCCL = 0.
END
```

Sur cet exemple, on observe que la routine WZCARRETL utilise deux fois XCARRETL qui sauve la valeur de X à chaque fois : on a donc 2 sauvegardes de la variable X, bien qu'elle n'ait pas été modifiée.

Le problème d'optimalité est dû à la localité de la méthode de sauvegarde : on sauve les variables directes au moment où elles sont utilisées.
Pour remédier à cela, on pourrait envisager de sauver les variables avant leur modification, mais se pose alors le problème de l'utilisation ultérieure de la variable sauvée : comment savoir si le contenu d'une variable doit être protéger ou non ?

La solution optimale serait de coupler :

- la méthode de sauvegarde sur fichier utilisant notre approche fichiers tangent/cotangent qui détecte les non-linéarités, et insère les ordres de lecture/écriture,

- à un suivi des variables actives du code qui détermine le lieu des affectations.

Comme Odyssée 1.6 effectue déjà un suivi de variables pour opérer la dérivation du code entier, il sera probablement plus facile de coder la sauvegarde sur fichier dans Odyssée que d'implémenter celle-ci à partir des codes dérivés produits par Odyssée.

Remarque : Compte tenu de l'utilisation de conventions d'écriture dans Meso-NH et d'une programmation du code en Fortan 90 (définition explicite des variables d'entrée et de sortie), il est possible d'organiser, assez simplement, le suivi des variables.

3.5 Conclusions sur les sauvegardes sur fichier

Nous avons proposé deux constructions de la trajectoire sur fichier. La première s'adresse plus spécifiquement à des codes produits par Odyssée, alors que la stratégie de trace, plus robuste, s'applique de manière générale pour un dérivateur automatique produisant code tangent et code cotangent.

Dans le cadre d'une utilisation du code adjoint pour optimiser une fonctionnelle, nos procédures peuvent être moins coûteuses en temps que l'évaluation proposée par Odyssée. En effet, notre méthode de sauvegarde revient à évaluer une fois le code tangent (direct+linéarisé ou direct seul) et une fois le cotangent, alors que le code cotangent issu d'Odyssée évalue à nouveau le code direct pour son calcul de la trajectoire. De plus, la place nécessaire en mémoire vive est moins importante lorsque l'on sauve la trajectoire sur fichier ; en contrepartie, le fichier est susceptible d'être très très gros.

Remarque : Compte tenu du recalcul local de la trajectoire par Odyssée, l'évaluation du code cotangent nécessite souvent plus d'une évaluation du code direct.

4 Validation automatique de l'adjoint

Comme moyen de vérification des codes dérivés construits, on utilise classiquement le test du gradient et le test du produit scalaire. Le premier étudie l'adéquation du code direct avec les codes dérivés par comparaison de gradients directionnels, le second permet de valider la dérivation (de codes éventuellement non différentiables) au sens où l'on vérifie seulement l'adéquation entre le code linéaire tangent et le code linéaire cotangent. Pour chacun des tests, nous donnons formellement les explications "mathématiques" et informatiques (mise en œuvre).

4.1 Le test du gradient d'un fonctionnelle

Soit P un programme informatique tel que

$$P \ : IR^n \rightarrow IR^m \tag{1}$$
$$u \ \rightarrow P(u) = v. \tag{2}$$
$$\tag{3}$$

Pour valider les procédures de dérivation de notre programme, on choisit une fonctionnelle J :

$$J \ : IR^m \rightarrow IR \tag{4}$$
$$v \ \rightarrow J(v) \quad [= J(P(u))], \tag{5}$$
$$\tag{6}$$

dans notre cas, J peut être la norme du vecteur v.

4.1.1 Version "mathématique"

Pour vérifier l'exactitude du vecteur gradient de la fonctionnelle J, un test expérimentalement fiable consiste à employer un développement de Taylor de J au point v. En notant $h \in IR^m$ une perturbation dans la direction $\delta v \in IR^m$ ($\| \delta v \|$=1), il découle que

$$J(v+h) = J(v) + (\vec{\nabla} J(v), h) + o(\| h \|). \tag{7}$$

Si, pour toute direction $\delta v \in I\!R^m$, on vérifie

$$lim_{\alpha \to 0} \frac{J(v + \alpha\, \delta v) - J(v)}{\alpha.(\vec{\nabla} J(v), \delta v)} = 1 \tag{8}$$

alors le test du gradient est satisfait : la linéarisation du code direct est convenable.

4.1.2 Version informatique, Algorithme

Soit un code P formé par un ensemble de N routines, il s'écrit formellement

TAB. 21 – *Routine P*

```
ROUTINE P (U, V)
  CALL R1(U,V)
  ...
  CALL RN(U,V)
END ROUTINE
```

avec U et V, vecteur d'entrée et de sortie, tous deux constitués de variables scalaires et de tableaux de variables scalaires de taille et de nature variées (entier, réel, complexe). Pour valider ses codes dérivés tangent PTL et cotangent PCL, on emploie la procédure suivante :

Algorithme pour tester le gradient

Soit d une direction de perturbation et U un point donné, on effectue les opérations suivantes :

- exécuter P(U, V_0) (résultat dans V_0),
- calculer $J_0 =$ J(V_0),
- calculer le gradient ∇J_0 (par utilisation de PTL ou PCL)
- Pour I=1,10
  - $\alpha = \dfrac{1.}{10^I}$
  - exécuter P(U+α δV, V_α) (résultat dans V_α),
  - calculer $J_\alpha =$ J(V_α),
  - évaluer
    $\text{Limite(I)} = \dfrac{J_\alpha - J_0}{\alpha(\nabla J_0, \delta V)}$

Si les composantes du vecteur Limite tendent vers 1 lorsque I devient grand, alors le test du gradient est satisfait dans la direction δV.

4.2 Le test du produit scalaire

Le test du produit scalaire consiste en une vérification du code cotangent utilisant le code tangent, des perturbations de l'état direct (solution du code direct) et des calculs de produits scalaires.

Généralement, ce test très coûteux est effectué sur les codes tangent et cotangent entiers et pour un petit nombre de directions de perturbations. Mais dans l'éventualité d'une dissonance entre les deux codes, il peut être fastidieux d'identifier manuellement l'erreur commise dans l'un ou l'autre code.
Pour se prémunir contre cette éventualité, une automatisation du test est nécessaire.

Remarque : Ce test permet aussi de valider notre procédure de sauvegarde de la trajectoire.

4.2.1 Version "mathématique"

Soit une partie de code r (ensemble de routines de P) qui, à partir d'un vecteur d'entrée u de dimension n, calcule un vecteur de sortie v de dimension m par

$$v = r(u). \tag{9}$$

Soit $r'(u)$ le code linéaire tangent, supposé correct, calculé en un point u donné. On choisit une direction δu de u, puis, on calcule l'approximation au premier ordre δv correspondant à v par la relation

$$\delta v = r'(\delta u). \tag{10}$$

Soit r^* le code cotangent de r, on calcule δu^* à partir de δv comme suit

$$\delta u^* = r^*(\delta v). \tag{11}$$

Si l'égalité suivante est satisfaite pour toute direction δu de u

$$< \delta v, r'(\delta u) > = < r^*(\delta v), \delta u > \tag{12}$$

i.e.

$$|\delta v|^2 = < \delta u^*, \delta u > \tag{13}$$

alors le code linéaire cotangent est validé eu égard au code linéaire tangent.

Remarque : Les codes dérivés r' et r^* sont linéaires en u : les relations ci-dessus peuvent s'écrire matriciellement en définissant cette fois r' et r^* comme la matrice Jacobienne de r et sa transposée.

4.2.2 Version informatique

Ce paragraphe peut être utilisé dans un contexte général, mais est dédié à Odyssée par le formalisme et les notations utilisées.

Soit une routine R (ou un ensemble de routines) ayant un ensemble d'arguments d'entrée U, un ensemble d'arguments de sortie V et formée d'une liste d'instructions r (de dérivées r' et r^* dont nous n'indiquons que les arguments d'entrée) :

TAB. 22 – *Routine R*

```
ROUTINE R (U, V)
  V = r(U)
END ROUTINE
```

Calculés par Odyssée, les codes dérivés tangent et cotangent s'écrivent respectivement de la manière suivante :

TAB. 23 – *Routines dérivées RTL et RCL*

```
ROUTINE RTL (U, V, UTTL, VTTL)
c   δv = r'(δu)
    VTTL = r'(U, UTTL)
    V = r(U)   (calcul local du code direct)
END ROUTINE
```

```
ROUTINE RCL (U, V, UCCL, VCCL)
    calcul local du code direct
c   δu* = r*(δv)

    UCCL = r*(U,VCCL)
END ROUTINE
```

Avec ces notations, effectuer le test revient à exécuter le sous-programme suivant :

4.2.3 Algorithme pour tester le produit scalaire

TAB. 24 – *Test du produit scalaire*

```
c   U est connu

Faire un certain nombre de fois

     choisir UTTL la perturbation UTTL
     initialisation des variables

     ROUTINE TEST_SCALAIRE (copie de U, UTTL)
          CALL RTL (U, V, UTTL, VTTL)
          VCCL = VTTL
          CALL RCL (U,V,UCCL,VCCL)
          calcul du produit scalaire <VCCL,VTTL>
          calcul du produit scalaire <UCCL,UTTL>
          comparaison de <VCCL,VTTL> = <UCCL,UTTL>
     END ROUTINE
```

Remarque : Le test est fait avec une copie de U car cette variable est modifiée dans la routine tangente.

4.3 Détails importants

Les programmes de validation et les programmes de sauvegarde de trajectoire sont écrits en Fortran 90 car la gestion des tableaux y est plus commode (fonctions prédéfinies).

Dans l'absolu, les tests doivent être satisfaits pour toute direction δU de U. Cependant, la dimension de U, bien que finie, est généralement très grande et ne permet pas d'effectuer les tests pour toutes les directions canoniques de U en un temps calcul raisonnable. Lorsque le nombre de directions est trop grand, on choisit les directions aléatoirement.

Remarque : En plus des tests aléatoires, il est possible d'orienter quelque peu le choix de la perturbation afin de vérifier le comportement des codes dérivés aux endroits stratégiques du modèle, par exemple aux frontières du domaine de calcul.

5 Meso-NH 2.4 : le modèle

5.1 Description succincte du modèle météorologique

Le modèle atmosphérique Meso-NH est développé conjointement par le Centre National de Recherches Météorologiques (Météo-France) et le Laboratoire d'Aérologie (CNRS) de Toulouse. Ce modèle est susceptible de simuler des phénomènes allant de quelques mètres (micro-échelle) à 100 km (meso-échelle), une description complète de ses spécificités est donnée dans la documentation scientifique [13], algorithmique [14] et le manuel d'utilisation [15].

Meso-NH, modèle meso-échelle non-hydrostatique, présente les caractéristiques suivantes.

5.1.1 Modèle anélastique

La propagation rapide des ondes acoustiques lors d'une discrétisation eulérienne du système d'équations impose une contrainte forte sur la taille du pas de temps. Il existe plusieurs méthodes pour éliminer ces problèmes et l'équipe de Meso-NH a choisi l'approximation anélastique. Dans ce cas, les ondes acoustiques sont supprimées des équations de continuité par l'utilisation d'un profil constant de densité du fluide : d'une certaine façon cela revient à travailler avec un fluide incompressible, la pression étant ensuite obtenue comme solution d'une équation elliptique (sous cette approximation, la conservation de la masse est assurée par ajoût d'une équation). Les avantages de cette approximation sont un système sans onde de gravité et la possibilité d'employer une discrétisation temporelle explicite. Cependant il apparaît :

- de petites imprécisions sur les vitesses verticales,

- des perturbations de pression liées aux ondes acoustiques filtrées qui ne se déplacent plus nécessairement avec la vitesse du son (transmission instantanée),

mais de l'avis des concepteurs de Meso-NH, ces inconvénients de modélisation n'altèrent pas de manière significative les résultats du modèle. Aussi, dans l'attente de preuves contraires, ils ont adopté la stratégie anélastique de Lipps et Hemler [11], tout prévoyant un éventuel changement de modélisation dans le code (retour à une formulation compressible des équations).

5.1.2 Discrétisation spatiale et modèles emboîtés

Meso-NH, codé par des schémas aux différences finies, offre la possibilité de réaliser avec le même code, des simulations 1D vertical, 2D ou 3D en espace, sur des grilles de discrétisation provenant de projection conforme ou cartésienne, éventuellement étirées. Récemment développée (meso-NH 3), la possibilité d'utiliser de grilles d'espace emboîtées permet de modéliser finement des phénomènes locaux à moindre coûts.
Dans cette dernière approche, Meso-NH associe un modèle nommé *model$n* à chacune de ces grilles ; les caractéristiques locales de chaque modèle peuvent alors différées (choix du schéma de résolution, paramètres différents, ...) ce qui améliore les résultats. Informatiquement, le code est dupliqué autant de fois que l'on a de modèles imbriqués.

Contrairement aux modèles de circulation générale (planétaire) tels que ARPEGE (Météo-France) ou ECMWF (modèle du Centre Européen) qui utilisent naturellement des conditions aux limites périodiques sur les frontières latérales du domaine géographique, les modèles petite ou moyenne échelle tels que Meso-NH nécessite une description des variables aux frontières. À cet effet, Meso-NH [13] propose des modélisations des conditions limites très variées. Pour exemple, les conditions latérales sont gérées comme suit :

- lorsque les domaines sont emboîtés, ils s'échangent les données d'interface.
- Lorsque l'on ne travaille qu'avec un modèle (ou pour le modèle contenant les autres modèles), les conditions aux frontières latérales peuvent être cycliques, ouvertes (radiatives), de type mur rigide, ou être fournies par des champs provenant le modèle atmosphérique à grande échelle comme ARPEGE ou ECMWF.

Outre ces modélisations, Meso-NH est susceptible d'introduire finement l'orographie dans le modèle, condition indispensable à l'étude d'événements atmosphériques réels.

5.1.3 Les phases de l'eau

Meso-NH propose 7 modélisations différentes pour l'eau : vapeur, nuage contenant de l'eau liquide, pluie, nuage contenant de la glace, neige, grésil et grêle. Ces différentes substances sont représentées par un rapport de mélange calculé par division de la masse de la substance considérée par la masse de l'air sec.

5.1.4 Utilisation de Meso-NH

Les possibilités de modélisation de phénomènes atmosphériques offertes par Meso-NH vont de la simulation de cas idéaux, à la reconstitution de phénomènes météorologiques réels. Toutes les variantes d'utilisation sont décrites dans [15].

5.2 Algorithmique générale

Chaque modèle *model$n* suit le même schéma algorithmique [14].
La partie adiabatique du modèle, qui ne retient que la dynamique, est écrite en caractères gras (Tab. 25) ; ce modèle réduit est appelé modèle adiabatique dans la suite du document.

Sans prétendre expliquer les phénomènes physiques soutendant cet algorithme, nous exposons succintement le rôle de quelques unes des routines dans la modélisation météorologique :

- **advection** : Cette routine réalise l'advection des variables du modèle, tant pour les champs de vents que pour variables scalaires. Plusieurs schémas de résolution sont proposés par Meso-NH, mais nous n'avons dérivé que la branche présentée.

- **dyn_sources**: Cette partie calcule l'influence des forces de Coriolis et de gravitation ainsi que le terme de courbure en fonction du système de coordonnées utilisé.

- **num_diff** et **relaxation**: Leur utilisation évite certains phénomènes numériques comme la réflection d'ondes aux frontières du domaine. Plus spécifiquement, num_diff applique une diffusion numérique d'ordre 4 destinée à lisser spatialement les valeurs des variables; ce lissage est effectué par rapport aux valeurs des variables calculées sur une grille grossière (variables dites "à grande échelle"). Quant à elle, relaxation supprime les phénomènes de réflection numérique par utilisation de conditions aux limites absorbantes avec amortissement variable.

- *phys_param$n*, *les_flex_spectra* et *resolved_cloud* sont un ensemble de sous-programmes traitant les paramétrisations physiques telles que micro-physique, phases de l'eau, ...

TAB. 25 – *Structure du modèle météorologique direct*

model$n	→	initialisations				
	→	**boundaries**				
	→	**initial_guess**				
	→	**advection**	→	**metrics**		
			→	**contrav**		
			→	**advecuvw**		
			→	**advecscalar**		
	→	**dyn_sources**				
	→	**num_diff**				
	→	**relaxation**				
	→	phys_param$n	→	sunpos$n		
			→	radiations		
			→	nature du sol		
			→	turb		
	→	les_flx_spectra				
	→	**rad_bound**				
	→	**pressure**	→	**metrics**		
			→	**mass_leak**		
			→	**gdiv**	→	**contrav**
			→	**flat_inv**	→	**(fft)**
			→	**richardson**	→	**flat_inv**
			→	**p_abs**		
	→	resolved_cloud				
	→	**endstep_dyn**				
	→	**endstep_scalar**				
	→	set_coupling				
	→	resultats				

- **pressure**: Tout comme pour l'advection, le calcul de la pression peut être effectué par différentes méthodes, et nous avons choisi de traiter la branche écrite en (Tab. 25).

- **rad_bound**: Cette procédure évalue le gradient de pression aux frontières latérales du domaines.

- **endstep_dyn** et **endstep_scalar**: Toutes deux gèrent les incrémentations temporelles.

- *set_coupling*: Cette routine est utilisée pour coupler le model$n avec des informations exterieures (le couplage entre model$n et model$m n'est pas présent dans la version de Meso-NH 2.4).

Les routines présentées dans ce schéma ne sont pas terminales, elles font généralement appel à des opérateurs de calcul de champs moyens et de calcul des schémas aux différences finies. Ainsi, le graphe présenté comporte 5 niveaux ; comme Meso-NH 3 offre maintenant la possibilité d'utiliser plusieurs modèles emboîtés pour effectuer des simulations, l'algorithme final comporte donc 6 niveaux, pour un code de 200 000 lignes d'instuctions et de commentaires.

5.3 Normes de programmation

Compte tenu de la taille du code, les développeurs du code Meso-NH se sont astreints à respecter scrupuleusement un ensemble de normes de programmation,le code est ainsi très lisible. Par exemple, des règles de préfixage pour les variables et paramètres (Tab. 26) sont utilisées.

TAB. 26 – *Préfixe des variables dans la norme de programmation de Meso-NH*

variable	INTEGER	REAL	LOGICAL	CHARACTER	TYPE
globale (MODULE)	N	X	L (pas LP)	C	T (pas TP, TS, TZ)
argument (de routine)	K (pas PP)	P	O	H	TP
locale	I (pas IS)	Z (pas ZS)	G (pas GS)	Y (pas YS, YP)	TZ
boucle	J (pas JP)				
PARAMETER	JP	PP	LP	YP	-
SAVE	IS	ZS	GS	YS	TS

Puisque nous allons utiliser le logiciel de dérivation automatique, nous ajoutons aux règles préexistantes, les conventions d'Odyssée concernant le suffixage des noms de variables et de routines dérivées.

TAB. 27 – *Suffixe des variables dérivées dans la norme Odyssée*

variable	INTEGER	REAL	LOGICAL	CHARACTER	TYPE
tangente	TTL	TTL	TTL	-	-
cotangente	CCL	CCL	CCL	-	-

Remarque : Bien qu'écrit en Fortran 90, ce code est dépourvu de pointeur.

5.4 Absence d'un code adjoint pour Meso-NH

Avant ce travail, les codes dérivés de Meso-NH n'existaient pas. Comme indiqué en introduction, notre objectif est une écriture "automatique" de ceux-ci.

6 Odyssée et Meso-NH

Dans les chapitres précédents nous avons décrit :

- les potentialités du logiciel Odyssée pour des codes écrits en fortran 77,
- une méthode de sauvegarde de trajectoire sur fichier et une procédure de validation,
- le code de météorologie Meso-NH écrit en Fortran 90.

Cette différence de langage interdit l'emploi direct Odyssée sur Meso-NH et implique une réécriture partielle du code adiabatique. Cette étape franchie, nous dérivons le code adiabatique afin de garantir le suivi des variables de contrôle. Ensuite nous étudions plus spécifiquement la partie advection de Meso-NH : implantation de la procédure de sauvegarde sur fichier et validation de toutes les routines appelées.

6.1 Un "Meso-NH" pour Odyssée

Nous avons décidé d'écrire rapidement un traducteur F90 vers f77 adapté à notre problématique :

dériver Meso-NH avec Odyssée tout en conservant, le plus souvent possible, les normes de programmation (paragraphe 5.3).

Les transformations suivantes sont à réaliser :

- inclusion des modules (spécifiques à Fortan 90) dans le code.
- transformation de fonctions spécifiques à Fortran 90 en paramètres du code (SIZE : dimension de tableau),
- déclaration externe de fonctions spécifiques à Fortran 90 (SPREAD : duplication de parties de tableau). Ces fonctions, retirées du code, ne sont pas explicitement dérivées par Odyssée.
- Remplacement des dimensions matricielles masquées $PA(:,:,:)$ par des dimensions fictives $PA(Q,Q,Q)$ où Q est un paramètre entier ajouté aux déclarations de la routine. En procédant ainsi, on conserve la structure des opérations matricielles de Fortran 90, sans avoir à écrire les boucles nécessaires (f77) à de telles opérations.

Avec ces modifications du code source, le code converti est compilable en fortran 77 et interprétable par Odyssée, mais inexploitable dans le cadre de la simulation.

6.2 Les paramètres de dérivation

En concertation avec les météorologistes toulousains, les variables d'entrée actives du modèle sont :

- les trois composantes de la vitesse de l'air dans la représentation cartésienne u, v et w,
- la température potentielle sèche θ (par opposition la température virtuelle T calculée en fonction des sept rapports de mélange de l'eau),
- l'énergie cinétique turbulente e,
- les rapports de mélange $r_n = \text{eau}_n/\text{air}$ (7 phases),
- la pression Φ,
- les termes source des variables scalaires passives s_n.

Toutes ces variables dépendent du temps.

6.3 Graphe fonctionnel du code adiabatique tangent

Les codes convertis sont dérivés grâce à Odyssée ; le graphe du code tangent est écrit ci-dessous, on y retrouve mélé les routines linéarisées et les routines directes telles que *metrics* : cette routine non dérivée est utilisée pour évaluer le code direct. Nous ne proposons pas ici le graphe fonctionnel du code cotangent produit par Odyssée qui est très long car le calcul local de la trajectoire implique un grand nombre d'appels aux routines.

TAB. 28 – *Structure du code adiabatique tangent*

modeltl$n	→	(initialisations)	non dérivée			
	→	boundariestl				
	→	initial_guesstl				
	→	advectiontl	→	metrics		
			→	contravtl		
			→	advecuvwtl		
			→	advecscalartl		
	→	dyn_sourcestl				
	→	num_difftl				
	→	relaxationtl				
	→	(phys_param$n)	non dérivée			
	→	(les_flx_spectra)	non dérivée			
	→	(rad_boundtl)	non dérivée			
	→	pressuretl	→	metrics		
			→	mass_leaktl		
			→	gdivtl	→	contravtl
			→	flat_invtl	→	(ffttl)
			→	richardsontl	→	flat_invtl
			→	p_abstl		
	→	(resolved_cloud)				
	→	endstep_dyntl				
	→	endstep_scalartl				
	→	(set_coupling)	non dérivée			
	→	(resultats)	non dérivée			

Sur un tel modèle, la méthode de conservation de la trajectoire est cruciale, et le choix d'un recalcul local de celle-ci expose rapidement ses faiblesses tant par la place mémoire indispensable à l'exécution du code cotangent que par le temps nécessaire aux multiples recalculs. Ce problème de place mémoire est levé par une sauvegarde de la trajectoire sur fichier ; des stratégies de sauvegarde/recalcul peuvent aussi être envisagées de manière à réduire le nombre de sauvegarde sur disque.

Pour motiver cette affirmation, on calcule le nombre d'appels de la routine *advecuvw* nécessaire pour une évaluation di code cotangent. *Advecuvw* est appelée :

- dans la trajectoire (de model$n) : appel d'advection puis d'advecuvw,
- dans advectioncl : appel d'advecuvw dans sa partie trajectoire,
- dans advecuvwcl : copie d'advecuvw dans sa partie trajectoire.

Au total, on emploie trois fois la routine advecuvw losque l'on veut évaluer la routine cotangente modelcl$n. Ce phénomène est d'autant plus important que la profondeur de l'arbre associé au code est grande.

6.4 La partie advection

Pour illustrer les deux manières de procéder vis à vis de la trajectoire, nous exposons les graphes fonctionnels relatifs à la routine *advection* obtenus par application des deux méthodes préservant la trajectoire. Dans la trajectoire du code cotangent produit par Odyssée, on observe aisément l'appel aux routines du code direct (Tab. 29) dû au recalcul local de la trajectoire, on rappelle que ces routines directes sont contenues dans les routines tangentes ce qui explique leur absence du code tangent. Cette constatation est à la base de notre stratégie de sauvegarde de trajectoire sur fichier.

6.4.1 Graphe d'appel des codes dérivés

Odyssée (brut)

TAB. 29 – *Routines dérivées d'advection, recalcul de la trajectoire par Odyssée*

advectiontl	→	metrics	advectioncl	→ (trajectoire)	metrics
	→	contravtl		→	contrav
	→	advecuvwtl		→	advecuvw
	→	advecscalartl		→	advecscalar
				→ (adjoint)	advecscalarcl
				→	advecuvwcl
				→	contravcl

Sauvegarde sur fichier

TAB. 30 – *Routines dérivées d'advection, sauvegarde sur fichier*

advectiontl	→	metrics	advectioncl	→ (adjoint)	advecscalarcl
	→	contravtl		→	advecuvwcl
	→	advecuvwtl		→	contravcl
	→	advecscalartl			

Remarque : Les fichiers cotangents ont le même nom, mais diffèrent quant à leur contenu : trajectoire pour le premier (Tab. 29), sauvegarde sur fichier pour le second (Tab. 30).

Ensuite, nous avons validé la partie advection du modèle en vérifiant le test du produit scalaire ; cela fonctionne bien, des résulats complémentaires seront donnés par la suite.

Mais les codes dérivés actuels requierent un nombre important de sauvegardes sur fichier, et plutôt que de valider des codes dérivés non optimaux, nous avons préféré travailler à nouveau la méthode de sauvegarde.

6.4.2 Améliorations des codes dérivés actuels

Dans Meso-NH, une routine modifie souvent une seule variable (éventuellement vectorielle), aussi la sauvegarde sur fichier que nous avons mise en œuvre conduit immanquablement à un grand nombre de sauvegardes inutiles. En effet, il est inutile de sauver localement des variables non modifiées même si elles apparaissent dans une expression nonlinéaire. Pour obtenir des codes "optimaux", nous avons conservé la méthode de sauvegarde présentée dans ce rapport, puis épuré les fichiers en opérant un suivi manuel des variables ; ce suivi est facile à réaliser sur un code Fortran 90 aussi bien écrit que Meso-NH.

Jusqu'à présent, nous avons également honni toute procédure de recalcul, même partiel, de la trajectoire. Cette stratégie est souvent intéressante. Par exemple, si l'on considère une partie de la routine advection (Tab. 31), on s'apercoit que l'on sauve 3 fois la variable intermédiaire SR01S alors qu'elle est calculée à partir d'une seule et même variable PRHODJ (non modifiée) (les opérateurs MXM, MYM, MZM calculent des moyennes).

TAB. 31 – *Advection (F90), Advectiontl: Calcul des composantes contravariantes du champ de vent*

```
ZRUT = PUT * MXM(PRHODJ)        SR01S = MXM(PRHODJ)
                                ZRUTTTL = 0.
                                INDICE_FICHIER = INDICE_FICHIER + 1
                                write(6,REC=INDICE_FICHIER)SR01S
                                ZRUTTTL = SR01S*PUTTTL
                                ZRUT = PUT*SR01S

ZRVT = PVT * MYM(PRHODJ)        SR01S = MYM(PRHODJ)
                                ZRVTTTL = 0.
                                INDICE_FICHIER = INDICE_FICHIER + 1
                                write(6,REC=INDICE_FICHIER)SR01S
                                ZRVTTTL = SR01S*PVTTTL
                                ZRVT = PUT*SR01S

ZRWT = PWT * MZM(PRHODJ)        SR01S = MZM(PRHODJ)
                                ZRWTTTL = 0.
                                INDICE_FICHIER = INDICE_FICHIER + 1
                                write(6,REC=INDICE_FICHIER)SR01S
                                ZRWTTTL = SR01S*PWTTTL
                                ZRWT = PUT*SR01S
```

Il est donc préférable de sauver une seule fois la variable PRHODJ, et de procéder à des recalculs dans le code cotangent, cela donne les intructions (Tab. 32).

TAB. 32 – *Advectiontl, Advectioncl localement optimisés*

```
SR01S = MXM(PRHODJ)             read(6,REC=INDICE_FICHIER)PRHODJ
ZRUTTTL = 0.                    INDICE_FICHIER = INDICE_FICHIER - 1
ZRUTTTL = SR01S*PUTTTL
ZRUT = PUT*SR01S                SR01S = MZM(PRHODJ)
                                PWTCCL = PWTCCL + ZRWTCCL*SR01S
SR01S = MYM(PRHODJ)             ZRWTCCL = 0.
ZRVTTTL = 0.
ZRVTTTL = SR01S*PVTTTL          SR01S = MYM(PRHODJ)
ZRVT = PVT*SR01S                PVTCCL = PVTCCL + ZRVTCCL*SR01S
                                ZRVTCCL = 0.
SR01S = MZM(PRHODJ)
ZRWTTTL = 0.                    SR01S = MXM(PRHODJ)
ZRWTTTL = SR01S*PWTTTL          PUTCCL = PUTCCL + ZRUTCCL*SR01S
ZRWT = PWT*SR01S                ZRUTCCL = 0.

INDICE_FICHIER = INDICE_FICHIER + 1
write(6,REC=INDICE_FICHIER)PRHODJ
```

Ces améliorations sont en cours d'implantation, les codes dérivés seront alors testés plus amplement.

6.4.3 Quelques résultats numériques

Sur la partie advection optimisée, l'utilisation de la routine tangente, contenant trajectoire et code direct, induit une multiplication par 1,95 du temps calcul consommé par l'exécution de la routine directe. Le temps nécessaire à l'evaluation du code cotangent est du même ordre : on procède au même nombre d'opérations sur le fichier tout en évaluant seulement les expressions linéarisées.

Il n'est pas possible, sauf à réécrire toute cette partie de code en fortran 77, d'obtenir le temps de calcul utilisé pour l'exécution des routines produites par Odyssée. À titre de comparaison, nous exposons dans le tableau (Tab. 33), le nombre de variables locales SAVE induites par Odyssée et le nombre de sauvegardes sur fichier. Les valeurs indiquent le nombre de tableaux à trois dimensions (*Odyssée 3* et *fichier 3*) et à quatre dimensions (*Odyssée 4* et *fichier 4*), les chiffres entre paranthèses signifient que la sauvegarde faite pour la routine advection, est utilisée dans d'autres routines.

Remarque : Un tableau à trois dimensions représente la grille spatiale tridimensionnelle, un tableau à quatre dimensions modélise les sept phases de l'eau sur la grille spatiale.

TAB. 33 – *Nombre de variables nécessaires à la sauvegarde de la trajectoire*

	Odyssée 3	*Odyssée 4*	*fichier 3*	*fichier 4*
advectioncl	20	2	0 (+4)	0 (+1)
advecscalarcl	36	24	2	1
advecuvwcl	45	0	0	0
contravcl	14	0	5	0
total	115	26	11	2
total 3	115+(26×7) = 297		11 +(2×7) = 25	

Remarque : Les résultats des colonnes "Odyssée" ont été obtenues avec l'algorithme initial proposé dans Odyssée 1.6.

La lecture de ces résultats suffit à justifier notre volonté d'optimiser la sauvegarde de trajectoire proposée par Odyssée. Tant qu'à faire des modifications sur les codes, l'approche par fichier nous a semblé plus intéressante car la capacité de sauvegarde sur disque est souvent plus importante que la capacité de stockage en mémoire vive.

Remarque : Dans cette optique, on peut aussi envisager de mettre en œuvre la technique de *checkpointing* proposée par A. Griewank et qui suggère d'effectuer la sauvegarde de la trajectoire selon un schéma hierarchique.

7 Validation des codes dérivés

7.1 Simulation testée

Pour valider les codes dérivés du modèle adiabatique, nous avons choisi la confifuration ayant les caractéristiques physiques suivantes :

- simulation : 2D,
- dimensions du domaine : 20 km × 2,5 km,
- système de coordonnées : cartésien,
- relief géographique : gaussien, hauteur = 250 m, base = 10000 m,
- condition initiale : idéalisée (Option : CSTN [13]),
- conditions limites latérales : périodiques,

- conditions limites verticales : flux verticaux nuls (haut), flux de surface nuls (sol),
- phases de l'eau : absentes,
- turbulence : absente.

Du point de vue numérique, la discrétisation utilise :

- pas d'espace : $90\times\ 3\ \times\ 60$, longueur des pas d'espace : 2000 m, 2000 m, 250 m,
- longueur du pas de temps : 20 s, durée de la simulation : 2000 s,
- advection : classique (pas de ftc),
- pression : calcul par une méthode de richardson (4 itérations).

Les discrétisations spatiale et temporelle employées sont physiquement acceptables pour la réalisation d'études météorologiques.

7.2 Vérification du code direct

Dans un premier test, nous avons vérifié que le code tangent de Meso-NH donne les mêmes résultats directs que Meso-NH. Pour le vecteur de vent U, W et la température tracés aux instants 1000s et 2000s, nous obtenons les figures Fig. 1 à Fig. 4 (je ne les ai pas jointes car j'ai.

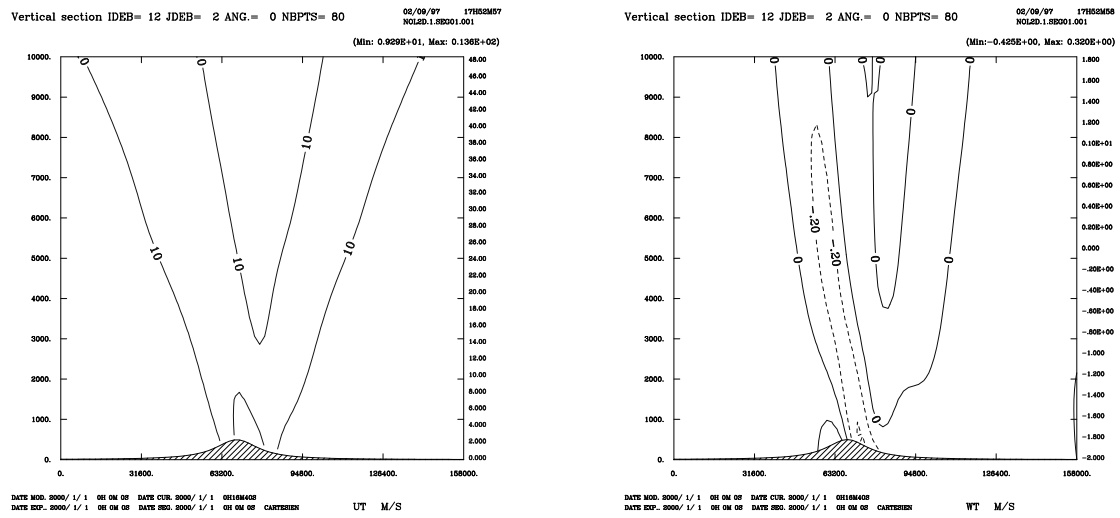

FIG. 1 – *Champs de vent : U, W, après 1000s.*

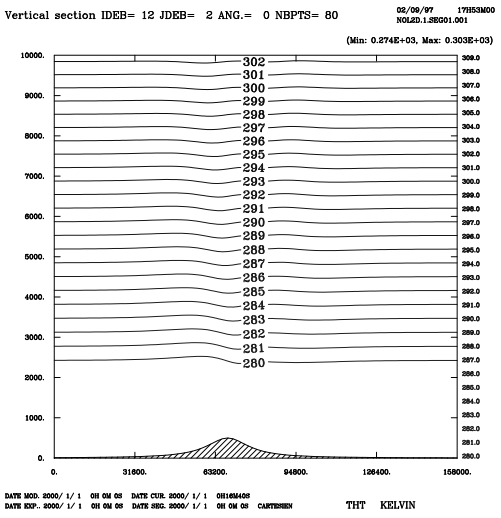

FIG. 2 – *Température après 1000s.*

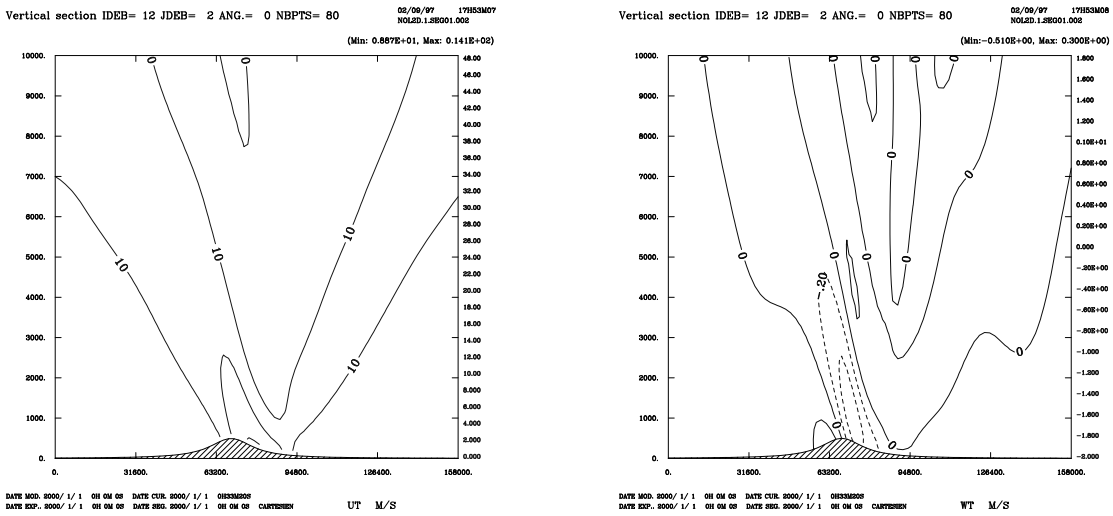

FIG. 3 – *Champs de vent : U, W., après 2000s.*

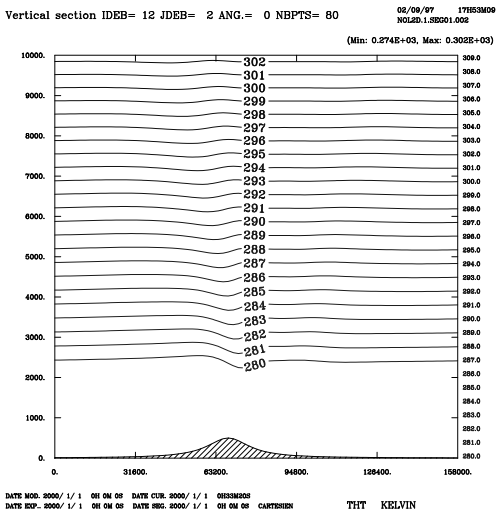

FIG. 4 – *Température après 2000s.*

7.3 Test du produit scalaire

Il est satisfait en tous les points où ont été réalisés les calculs : les codes linéaires tangent et cotangent se correspondent.

7.4 Test de Taylor

Comme nous l'avons déjà écrit au chapitre 4, un test expérimentalement fiable pour vérifier l'exactitude du vecteur gradient d'une fonctionnelle J consiste à employer un développement de Taylor de J au point v i.e.

$$J(v + \alpha\,\delta v) = J(v) + (\vec{\nabla} J(v), \alpha\,\delta v) + o(\alpha), \tag{14}$$

relation dans laquelle α est un réel et $\delta v \in IR^m$ une direction de perturbation ($\| \delta v \|$=1).

Si, pour toute direction $\delta v \in IR^m$, on vérifie

$$lim_{\alpha \to 0} \frac{J(v + \alpha\,\delta v) - J(v)}{\alpha.(\vec{\nabla} J(v), \delta v)} = 1 \tag{15}$$

alors le test du gradient est satisfait : la linéarisation du code direct est convenable.

Il va de soit que l'on ne peut vérifier ce test en tout point de la discrétisation et pour toutes les directions de perturbation des variables, aussi nous ne proposons les résultats de ce test qu'en un point du maillage et pour une direction de perturbation. On a

TAB. 34 – *Test de Taylor*

α	Rapport des gradients	α	Rapport des gradients
2	1.227298032300915	2^{-11}	0.9999830392757225
2^{-1}	1.026688611458553	2^{-13}	0.9999956103496999
2^{-3}	0.9981554834260713	2^{-15}	0.9999990304248598
2^{-5}	0.9990608061460371	2^{-17}	0.999999483416051
2^{-7}	0.9997368652204202	2^{-19}	1.000014323107955
2^{-9}	0.9999324766869648		

On observe (Tab. 34) que le rapport des gradients tend vers 1 lorsque α ($\alpha \geq 2^{-17}$) tend vers 0.
De ces résultats (et d'autres résultats non exposés), on déduit que les codes linéaires tangent et cotangent linéarisent effectivement Meso-NH (pour ce type d'utilisation de Meso-NH).

8 Conclusions

Meso-NH est un code destiné à la simulation d'événements météorologiques réels, ce code de grande taille est en constant développement et des modifications importantes sont susceptibles d'y être apportées. Compte tenu de ces observations, il était difficilement concevable d'utiliser une méthode de différences divisées (imprécise et coûteuse), ou la dérivation manuelle pour produire le code adjoint de Meso-NH. Dans ce contexte, la méthode choisie est la différentiation automatique qui génère de manière systématique les codes dérivés par rapport à certains paramètres (sous réserve que le code soit différentiable).

Après une brève revue des logiciels existants, nous avons retenu le logiciel Odyssée pour ses capacités à différentier des codes entiers. Détaillées au chapitre 2, les améliorations apportées à Odyssée 1.6, suivi de variable et graphe fonctionnel, facilitent la dérivation de codes complets. Cependant la stratégie de différentiation en mode cotangent pèche par le recalcul local de la trajectoire nécessaire à l'évaluation du code adjoint.
Ce point peut sembler anodin à la lecture des codes dérivés, mais constitue un vrai problème lors de l'exécution de ceux-ci. En effet, le recalcul de la trajectoire dans chaque routine du code, implique l'emploi, en grand nombre (Tab. 33), de variables locales préservant la solution directe. Utilisé sur des codes industriels, coûteux en mémoire, cette stratégie de recalcul local induit des codes dérivés encore plus coûteux en mémoire et, éventuellement, en temps.

L'alternative choisie dans ce document (chapitre 3), est la mise en oeuvre d'une sauvegarde sur fichier de la trajectoire. Afin d'assurer une certaine optimalité de la sauvegarde, nous avons cherché à identifier les parties non linéaires du code pour ne préserver que les valeurs des variables directes effectivement utilisées dans le code cotangent. Les codes dérivés, en mode tangent ou cotangent, étant tous deux construits par linéarisation du code direct, la détermination des parties non linéaires peut être réalisée sur l'un ou l'autre.
L'évaluation du fichier tangent, tout comme celle du fichier cotangent, requiert la connaissance de l'état direct. Dans Odyssée, la trajectoire est connue grâce à la présence des lignes du code direct au sein du fichier tangent. Ainsi, on connaît la valeur de toute variable directe apparaissant dans une expression linéarisée du code linéaire tangent, cette présence permettant d'extraire les variables directes qui nécessitent une sauvegarde sur fichier.
Pour obtenir une méthode de sauvegarde sur fichier, il suffit de placer à bon escient les ordres d'écriture, resp. de lecture, dans le fichier tangent, resp. cotangent, des variables directes utiles. Pour réaliser cela, nous avons introduit le concept de trace dans les fichiers à dériver ; celle-ci sépare les lignes du code direct par des marqueurs (Tab. 11) et contribue ainsi à l'identification, aisée, des lignes linéarisées tangentes et cotangentes correspondant à une même ligne du code direct.
Le concept de trace est externe à Odyssée et peut être généralisé à d'autres logiciels de différentiation automatique.

Les algorithmes de sauvegarde décrits tiennent compte, routine par routine, des modifications apportées aux variables sauvées ce qui évite la redondance locale de sauvegarde. Pour assurer globalement l'optimalité de la sauvegarde, il faudrait sauver ces variables juste avant qu'elles ne soient affectées. Mais procéder ainsi pose le problème de la détermination des variables à sauver : comment savoir si le contenu d'une variable doit être protégée ?

Une solution plus optimale serait de coupler la méthode de sauvegarde sur fichier utilisant l'approche *fichiers tangent/cotangent* à un suivi des variables actives du code (celui d'Odyssée par exemple).

Les méthodes de construction des codes dérivés étant établies, Odyssée + sauvegarde sur fichier, nous avons programmé des outils de validation pour les codes dérivés : test du gradient et test du produit scalaire. Implémantées en Fortran 90, l'utilisation de ces routines a été systématique ce qui nous a permis de valider la dérivation des codes et la sauvegarde de trajectoire.

Les outils construits, nous avons entrepris la dérivation de la partie adiabatique du modèle atmosphérique Meso-NH, nous avons procédé comme suit :

- traduction du code direct Meso-NH en fortran 77,
- insertion de la trace dans le code converti,
- dérivation du code direct et de la trace, obtention des codes dérivés tangent et cotangent,
- analyse des codes dérivés, insertion des ordres d'écriture, resp. lecture, dans le code tangent, resp. cotangent,
- destruction de la trace dans les fichiers dérivés,
- validation des codes dérivés,
- suppression des instructions linéarisées dans le code tangent,
- réécriture des adjoints de Meso-NH en Fortran 90,
- optimisation des codes dérivés,
- validation des codes dérivés optimisés.

Le résultat d'une telle procédure, automatisée en grande partie, est la création d'un code direct contenant les ordres d'écriture sur fichier des variables nécessaires à l'évaluation du code adjoint, et la création d'un code adjoint contenant les instructions dérivées en mode cotangent et les ordres de lecture sur fichier de ces variables.

Quelques résultats de la dérivation par sauvegarde sur fichier sont donnés à la fin du chapitre 6, et quelques résultats sur la vérification des codes dérivés sont proposées au chapitre 7 : le test de Taylor converge pour la simulation réalisée, les codes linéaires sont corrects dans ce cas.
La version adiabatique de Meso-NH a été entièrement linéarisée, puis vérifiée sur la simulation 7.1 : les codes dérivés sont á vérifier pour les très nombreux types de simulation proposés par Meso-NH.

Meso-NH dérivé, nous nous orientons vers une exploitation des codes dérivés en collaboration avec les météorologues toulousains.

En quelques mots, la dérivation automatique de codes est un bon outil pour la construction des codes dérivés de codes industriels, mais elle mérite les améliorations qui en feront un outil incontournable.

Références

[1] Computational Differentiation : techniques, Applications and Tools,
Editors: M. Berz, C. Bischof, George Corliss and A. Griewank, Second International Workshop on Computational Differentiation (Santa Fe 1995), Siam, 1996, http://www.beamtheory.nscl.msu.edu/cd96/

[2] C. Bischof, A. Carle, P. Khademi and A. Mauer, *The ADIFOR2.0 System for the Automated Differentiation of Fortran 77 Programs*, Argonne Preprint ANL-MCS-P481-1194, and CRPC Technical Report CRPC-TR94491.

[3] F. Eyssette, C. Faure, J.C. Gilbert and N. Rostaing-Schmidt, *Applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques dans un tube chauffant*, Rapport de Recherche INRIA RR-2745,
http://www.inria.fr/RRRT/RR-2795.html

[4] C. Faure, *Quelques aspects de la simplification en calcul formel*, Thèse de l'Université de Nice-Sophia Antipolis, 1992.

[5] C. Faure, *Splitting of Algebraic Expressions for Automatic Differentiation*, dans les actes du Second International Workshop on Computational Differentiation (Santa Fe 1995), Siam, 1996.

[6] C. Faure, *Documentation succinte d'Odyssée version 1.6*, décembre 1996.

[7] J.C. Gilbert, G. Le Vey, J. Masse, La Différentiation automatique de fonctions representées par des programmes, Rapport de Recherche INRIA RR-1557,
http://www.inria.fr/RRRT/RR-1557.html

[8] A. Griewank, D. Juedes, J. Srinivasan and Ch. Tyner, *ADOL-C, A Package for the Automatic Differentiation of Algorithms Written in C/C++*, ACM Transactions on Mathematical Software, vol. 22(2), Juin 1996, pp 131-167, Algor.755
ftp://info.mcs.anl.gov/pub/ADOLC/ADOLC_BETA/adolc.ps.Z

[9] A. Griewank, *Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation*, Optimization Methods and Software, 1(1):35-54, 1992.

[10] A. Griewank, D. Shiriaev and J. Utke, *A User Guide to ADOL-F*,
ftp://info.mcs.anl.gov/pub/ADOLC/ADOLC_BETA/adolf.ps.Z

[11] F.B. Lipps and R.S. Hemler, *A scale analysis of deep moist convection and some related calculations*, J. Atmos. Sci., 39, 2192-2210, 1982.

[12] J.-M. Malé, B.Mohammadi and N. Rostaing-Schmidt, *Automatic differentiation in direct and reverse modes: Application to optimum shapes design in fluid mechanics*, dans les actes du Second International Workshop on Computational Differentiation (Santa Fe 1995), Siam, 1996.

[13] Météo-France, CNRS *The Meso-NH Atmospheric Simulation System: Scientific Documentation*, version de septembre 1995.

[14] Météo-France, CNRS (J.P. Lafore, P. Vincent) *The Meso-NH Atmospheric Simulation System: Algorithmic Documentation (MASDEV2_3)*, version de juillet 1996.

[15] Météo-France, CNRS *The Meso-NH User's Guide*, version de septembre 1996.

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

[17] *A Compilation of Automatic Differentiation tools*, International Conference on Industrial and Applied Mathematics, 1995.
http://www.mcs.anl.gov/Projects/autodiff/AD_Tools/index.html#contents

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e 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)
http://www.inria.fr
ISSN 0249-6399