Approaches to Assess Validity of Derivatives and to Improve Efficiency in Automatic Differentiation of Programs

Une thèse présentée

par

Mauricio Araya Polo

à

INRIA - Université de Nice - Sophia-Antipolis (UNSA)

pour obtenir le titre de

Docteur en Sciences

spécialité science de l'ingénieur

UNSA

Sophia-Antipolis, PACA

Novembre 2006

Directeur de thèse
Laurent Hascoët

Auteur
Mauricio Araya Polo

Approaches to Assess Validity of Derivatives and to Improve Efficiency in Automatic Differentiation of Programs

Résumé

Ce travail concerne la Différentiation Automatique (DA) de codes. La DA transforme un programme calculant une fonction mathématique en un nouveau programme calculant ses dérivées, gagnant ainsi un temps de développement conséquent. L'usage de la DA se répand en Calcul Scientifique, mais souffre encore de quelques problèmes. Cette thèse propose des éléments de solution à deux de ces problèmes. Le premier problème est la non-différentiabilité des programmes réels, pour certaines entrées. Les outils de DA négligent souvent ce problème, alors que les utilisateurs ont besoin d'une grande confiance dans ces dérivées avant de les utiliser, par exemple dans des boucles d'optimisation. Quelle que soit son origine réelle, cette non-différentiabilité se traduit dans la structure de contrôle des programmes. Plutôt que d'étudier des extensions de la notion de dérivée, nous préférons ici caractériser le domaine autour des entrées courantes pour lequel le contrôle reste constant et la différentiabilité est conservée. Nous proposons plusieurs approches et évaluons leurs complexités. Nous étudions formellement une construction du domaine entier, mais sa complexité limite son application. Alternativement, nous proposons une méthode directionnelle moins coûteuse, que nous avons implémentée et validée sur plusieurs exemples. Le second problème est l'efficacité du mode inverse de la DA, qui produit des codes "adjoints" calculant des gradients. Ces codes utilisent les valeurs intermédiaires du programme initial dans l'ordre inverse, ce qui nécessite une combinaison de sauvegarde et de recalcul de ces valeurs. Une tactique fondamentale, nommée "ckeckpointing", économise de la mémoire au prix de la reexécution de segments de code. Dans notre travail, nous formalisons les analyses de flot de données nécessaires à la différentiation inverse, y compris dans le cas du checkpointing. A partir de cette formalisation, nous proposons deux avancées aux stratégies de la DA inverse. D'une part ces analyses nous fournissent des ensembles de valeurs à sauvegarder, que nous pouvons prouver minimaux. D'autre part nous en tirons des indications sur les meilleurs segments de code candidats au checkpointing. Pour expérimenter ces choix, nous étendons les analyses et l'algorithme de différentiation inverse de notre outil de DA. Nous montrons les bénéfices que l'on peut attendre sur des codes réels.

Table des matières

Title Page i
Resume ii
Table of Contents ii
Liste des figures vi
Liste des Tableaux vii
Références aux publications viii

1 Introduction **1**
1.1 Utilisation abusive des dérivées 3
1.2 Dépassement de mémoire dans le mode inverse 4

2 Différentiation Automatique **7**
2.1 DA: Cadre 7
2.2 Programmes et fonctions mathématiques 8
2.2.1 Programmes et instructions 8
Contrôle de flot et graph de flot 9
Appels de sous-programmed et the call-graph 10
2.2.2 Fonctions mathématiques implémentées par programmes . . . 11
2.3 La règle de dérivation en chaînes 11
2.4 Le mode tangent de DA 12
2.4.1 Définition et modèle 12
2.4.2 Example 13
2.5 Le mode inverse de DA 14
2.5.1 Définition et modèle 14
2.5.2 Example 15

3 Le domaine de la validité des dérivés **16**
3.1 Le Problème de Validité 17
3.1.1 La Differentiabilité des fonctions intrinsèques 17
3.1.2 Differentiabilité des rapports conditionnels 19
3.2 Définir et propager l'information de validité 21
3.2.1 Définition de l'Information de Validité 21
3.2.2 Propagation se l'Information de Validité 23

3.3 Propagation Arrière des Demi-Espaces (PADE) 23
3.3.1 Définition 23
3.3.2 Problèmes avec PADE 24
3.4 Propagation vers l'Avant Directionnelle (PAD) 24
3.4.1 Définition 25
3.4.2 Problèmes avec la PAD 26
3.4.3 Implementation de PAD 27
3.5 Résultats Expérimentaux 27
3.5.1 Expériences avec la méthode de Newton 27
3.6 Conclusions 32

4 Analyses de flot et Checkpointing pour le mode renversé de DA 34
4.1 Le problème de consommation de mémoire du mode inverse 35
4.1.1 Stratégie Stocker-Tout (ST) 36
4.1.2 Stratégie Stocker-Tout et contrainte de mémoire 37
4.2 Stratégies classiques pour l'approche Stocker-Tout 38
4.2.1 Stratégies échelle fine 38
Analyses de flot de données 38
Recalcul versus stockage 39
4.2.2 Stratégies échelle macro 39
Checkpointing 39
Hypothèse concernant la bande et le snapshot 41
Placements des points de Checkpointing 41
Définition du Snapshot 43
4.3 Un modèle formel du mode inverse Stocker-Tout de DA 43
4.4 Contributions aux stratégies échelle fine 44
4.4.1 Modèle amélioré du mode Stocker-Tout inverse de la DA . . . 45
4.5 Contributions aux stratégies échelle macro 45
4.5.1 Un modèle formel du mode inverse Stocker-Tout de la DA avec Checkpointing 45
4.5.2 La Définition Améliorée du Snapshot 46
4.5.3 Le Checkpointing Systématique 47
Simulation de stratégies hybrides supposant que $Snapshot < tape$ 48
Simulation de stratégies hybrides supposant que $Snapshot > tape$ 49
4.5.4 Observation expérimentale des problèmes et des résultats . . . 50
Codes Exemples 50
UNS2D 51
4.6 Conclusions 53

5 Conclusions et perspectives de recherche 55
5.1 Sommaire et conclusions 55
5.1.1 Le Problème De Validité 55

5.1.2 Le problème de mémoire du mode inverse 56
5.2 Remarques-conclusion 58

Bibliographie **59**

Liste des figures

1.1 Optimisation du boum sonique, CFD utilisant une méthode de gradient [13]. . . . 1
1.2 Modèle pour implémenter des dérivées. . . . 2
1.3 L'implémentation de ces formules mènent à des problèmes, à cause de la non différentiabilité de certaines entrées. . . . 3
1.4 Mode inverse de DA avec la stratégie ST. . . . 5

2.1 Two valid execution paths. . . . 9
2.2 Call-graph: nested calls. . . . 10

3.1 Differentiated flow-graph for example. . . . 19
3.2 Examples of problematic functions. . . . 20
3.3 Représentation directionnelle pour un exemple arbitraire. . . . 26
3.4 Méthode de Newton utilisant des dérivées générées par AD. . . . 29
3.5 Exemples de fonctions définies par morceaux pour la méthode de Newton. 29
3.6 Fonctions définies par morceaux. . . . 30

4.1 L'axe horizontal représente la quantité de valeurs stockée à un moment précis. . . . 37
4.2 Checkpointing sur le mode inverse DA. . . . 40
4.3 Checkpointing pour tous les appels dans le mode inverse de DA (mode joindre-tout). . . . 47
4.4 Pas Checkpointing dans le mode inverse DA (mode Dédoubler-tout). 48
4.5 Approche hybride (split-joindre) 48
4.6 Simulation des résultats, tape = 10, snapshot = 6. . . . 49
4.7 Résultats numériques génériques, tape = 6, snapshot = 10. . . . 50
4.8 Organigramme de UNS2D. . . . 51

Liste des Tableaux

2.1 Code différencié tangent produit automatique. 13
2.2 Automatic generated reverse differentiated code. 15

3.1 Remplacements de fonctions intrinsèques et de leurs dérivées pour la DA. 18
3.2 Fonction $f()$ et sa version différentielle tangente. 28
3.3 Fonction $f_piecewise()$ et sa version dérivée tangente. 30
3.4 Résultats numériques de la méthode de Newton pour une fonction définie par morceaux. 31
3.5 Résultats numériques de la méthode de Newton et la méthode PAD pour une fonction définie par morceaux. 31

4.1 les règles de la ST 36
4.2 Profiling information sur les snapshots, les bandes et le nombre d'appel aux sous-programmes de UNS2D. 52
4.3 Performance en mémoire et en temps pour le code UNS2D. 52

Références aux publications

Une grande partie du chapitre 3 a été publiée dans les papiers suivants :

> "Domain of Validity of Derivatives Computed by Automatic Differentiation", Mauricio Araya-Polo, Laurent Hascoët, INRIA Research Report #5237, june 2004.
>
> "Certification of Directional Derivatives Computed by Automatic Differentiation", Mauricio Araya-Polo, Laurent Hascoët, Article in WSEAS Transactions on Circuits and Systems, WSEAS, 2005.

La plus grande partie du chapitre 4 a été publiée dans :

> "Enabling User-driven checkpointing strategies in Reverse-mode Automatic Differentiation", Laurent Hascoët, Mauricio Araya-Polo, INRIA Research Report and also in Proceedings of the ECCOMAS conference, Egmond aan Zee, The Netherlands, September 2006.
>
> "Data Flow Algorithms in the TAPENADE tool for Automatic Differentiation", Mauricio Araya-Polo, Laurent Hascoët, Proceedings of the ECCOMAS conference, Jyvaskyla, Finland, july 2004.
>
> "The Adjoint Data-Flow Analyses: Formalization, Properties, and Applications", Laurent Hascoët, Mauricio Araya-Polo, Proceedings of the AD2004 Conference, Chicago, Illinois, july 2004.

Des préimpressions électroniques sont disponibles sur Internet aux adresses suivantes :

```
http://www-sop.inria.fr/tropics/
http://www-sop.inria.fr/tropics/Mauricio.Araya
```

Chapitre 1

Introduction

Le calcul scientifique s'intéresse à l'élaboration de programmes informatiques qui implémentent l'analyse et les solutions pour les problèmes scientifiques et d'ingénierie. L'analyse et les solutions sont en majorité basées sur des modèles mathématiques. Par exemple, certaines applications du calcul scientifique sont : la Mécanique des Fluides Numériques (CFD, "Computational Fluid Dynamics"), les modèles de prévision météorologique, les modèles biologiques, les simulations de collision de particules.

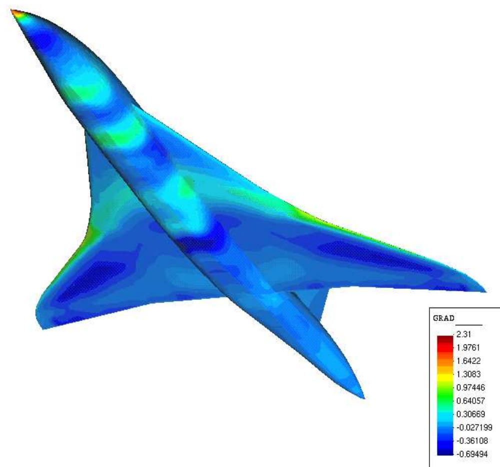

Figure 1.1: Optimisation du boum sonique, CFD utilisant une méthode de gradient [13].

Tous ces modèles mathématiques sont basés sur des méthodes et des algorithmes qui ont besoin de dérivées. Par conséquent, les modèles sont implémentés comme des programmes qui nécessitent d'implémenter des dérivées. Sur la figure 1.1, nous pouvons observer un exemple d'application du calcul scientifique : de la recherche CFD spécifique à l'aéronautique qui inclut une optimisation de forme [14]. Leurs méthodes

mathématiques reposent fortement sur des dérivées, en particulier des gradients. En fait, la figure 1.1 montre la variation des gradients pour une fonction objective qui prend en compte la forme de l'avion et la pression.

Comme nous pouvons l'observer sur la figure 1.2, lorsque l'on a besoin de dérivées, on peut soit le faire à la main soit utiliser la Différentiation Automatique (DA) [2, 8].

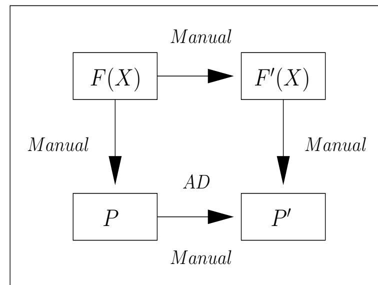

Figure 1.2: Modèle pour implémenter des dérivées.

Sur la figure 1.2, on peut observer le coût de l'implémentation de P', où P' est le programme qui calcule les dérivées nécessaires au modèle $F'(X)$. Il dépend du chemin que l'on suit. Le chemin le moins cher est celui où l'on démarre du modèle mathématique $F(X)$ par l'intermédiaire du programme P implémenté manuellement pour utiliser finalement DA pour calculer le programme differencié P'. D'autre part, l'implémentation manuelle du programme P' demande beaucoup de travail à cause de la taille des codes et de la complexité de la tâche. De plus l'implémentation manuelle de la tâche n'est pas fiable parce qu'elle peut introduire des bugs.

DA est extrêmement rapide et les dérivées générées sont fiables. Ceci est extrêment rapide parce que les modèles DA sont basés sur une application efficace de la règle de dérivation en chaîne. Par exemple, en utilisant l'outil DA TAPENADE [11], on a besoin de 10 minutes pour générer les dérivées de 50.000 lignes de code (LDC). Par contre, en le faisant ̀la main, cela nécessiterait de nombreuses heures de travail humain. Les dérivées obtenues en utilisant DA sont fiables car elles sont le résultat d'une application systématique et mécanique de la règle de dérivation en chaîne, de plus ces dérivées sont exactes et aussi précises que la précision machine. Les programmes différentiés sont également aussi efficaces que le programme original.

DA possède une longue histoire comme moyen de différencier les programmes, la première proposition a été faite en 1964 [21]. Bien que les modèles et les outils DA aient évolué vers de hauts niveaux de qualité, DA possède ses problèmes : une utilisation inconsidérée de DA peut mener à des dérivées peu fiables, comme nous le montrons plus tard dans 1.1. De plus, lorsque DA est utilisée pour générer certains

types de dérivées, par exemple pour le calcul de gradients, ceci peut mener à des dépassements de mémoire pour les gros programmes. Nous discuterons ceci en détail dans la section 1.2.

L'objectif de cette thèse est de traiter des deux problèmes nommés ci-dessus. Dans le reste de ce chapitre, nous donnons une vue d'ensemble de chacun de ces problèmes ainsi que les solutions que nous proposons.

1.1 Utilisation abusive des dérivées

Le comportement de la plupart des programmes dépend des valeurs d'entrées. Ceci est implémenté par instructions conditionnelles, chaque instruction conditionnelle décide quel est le prochain segment de code à exécuter parmi un ensemble de segments. Les modèles DA respectent la structure logique des programmes, par conséquent quand DA est appliquée, chaque instruction conditionnelle choisit parmi un ensemble de segments qui implèmentent les dérivées. En donnant deux entrées légèrement différentes, l'instruction conditionnelle peut choisir deux segments à exécuter complétement différents. En conséquent, le programme différencié retournera deux dérivées complétement différentes pour deux entrées légèrement différentes. Ceci signifie que les dérivées générées par les outils DA peuvent être inconsistantes ou même incorrectes. Par incorrect, nous voulons dire que les outils DA implémentent des dérivées même si les fonctions ne sont pas différentiables pour cette entrée, en particulier quand les fonctions sont implémentées en utilisant des instructions conditionnelles.

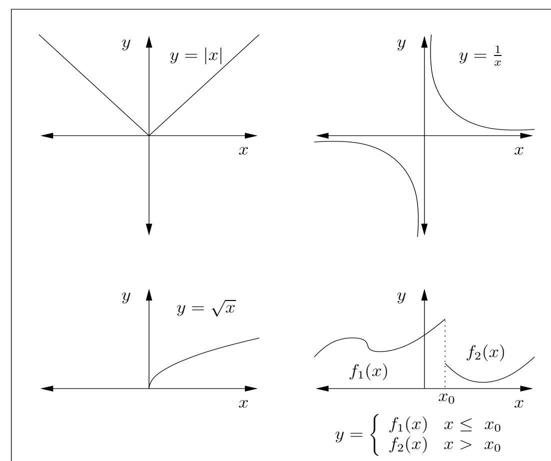

Figure 1.3: L'implémentation de ces formules mènent à des problèmes, à cause de la non différentiabilité de certaines entrées.

Sur la figure 1.3, nous pouvons observer certains cas qui peuvent produire des

dérivées non fiables, parce que les outils DA génèrent différents codes et ces codes peuvent être exécutés avec des entrées qui ne sont pas valides. Où les entrées non valides sont celles où les fonctions considérées présentent des non différentiabilités, où les entrées sont trop proches de changement radical dans le définition des fonctions, par exemple la fonction en bas à droite de la figure 1.3.

Les outils DA ne peuvent pas modifier le programme original parce les détails de fonctionnement du modèle implémenté ne sont pas connus. Par conséquent, nous pouvons seulement notifier à l'utilisateur que pour certaines entrées des dérivées ne sont pas fiables.

Notre intention est de rendre les dérivées utiles même dans les conditions décrites ci-dessus, pour cela nous voulons fournir le plus grande voisinage sûr. Ce voisinage sûr est défini comme la région dans l'intervalle des entrées pour lequel nous pouvons obtenir des dérivées consistantes. Afin de fournir ce voisinage, nous calculons la plus grande variation possible autour de l'entrée donnée qui ne change pas les segments de code à exécuter sélectionnés par les instructions conditionnelles.

Nous avons développé deux stratégies pour calculer le voisinage le plus sûr. La première stratégie, appelée PADE, fournit une description complète et précise (au premier ordre) du voisinage, mais a un coût de calcul prohibitif. La deuxième stratégie, appelée PAD, fournit une description moins complète du voisinage, au même niveau que PADE pour la précision, mais à un coût de calcul plus bas.

Nous avons conduit un certain nombre d'expériences pour valider la stratégie PAD. Nous avons découvert que la stratégie nous permet en effet de calculer le voisinage le plus sûr. Ceci sert à prévenir contre les dérivées inconsistantes ou les non différentiabilités dans le programme.

1.2 Dépassement de mémoire dans le mode inverse

Les gradients sont le type de dérivées le plus populaire utilisé par les méthodes mathématiques. La façon la plus efficace, en temps d'exécution, pour générer les gradients est d'utiliser ce qui s'appelle le mode inverse de DA [8]. Malheureusement, durant l'exécution du mode inverse, il faut rendre un nombre important de valeurs intermédiaires accessibles. Pour gérer ce problème, nous avons deux principales solutions possibles. La première stratégie est appelée Stocker-Tout (ST) [8], qui consiste à sauver les valeurs intermédiaires jusqu'à ce que l'on en ait besoin. Cependant l'efficacité du mode inverse coûte un prix important au niveau de la consommation de mémoire. La deuxième stratégie est appelée Recalculer-Tout (RT) [19], qui consiste en un recalcul de toutes les valeurs intermédaires lorsque l'on en a besoin. Cependant,

RT réduit drastiquement l'efficacité de temps d'éxécution pour le mode inverse.

Dans cette thèse, nous nous intéressons surtout au problème de la consommation de mémoire de la stratégie ST. Nous choisissons cette stratégie parce que dans l'outil (TAPENADE) développé par notre équipe nous avons déjà obtenu une bonne efficacité pour le temps d'exécution, en utilisant la stratégie ST, et notre objectif est de réduire la consommation de mémoire sans perdre trop de rapidité. C'est pourquoi dans cette thèse, nous analysons l'échange entre le temps d'exécution et la consommation de mémoire, en cherchant à fournir les dérivées les plus efficaces, en particulier pour les gradients.

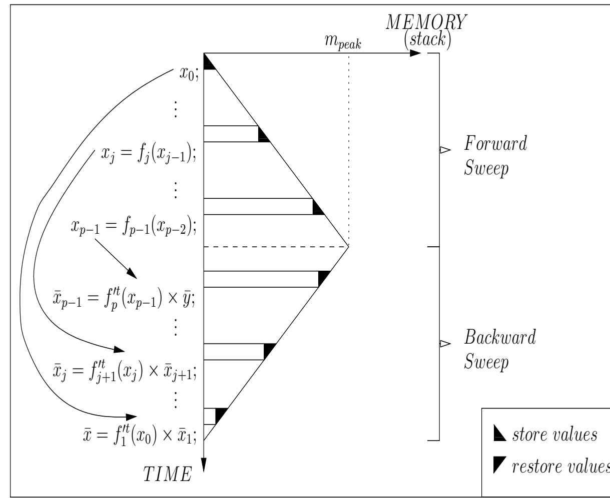

Figure 1.4: Mode inverse de DA avec la stratégie ST.

Dans la figure 1.4, nous pouvons observer les caractéristiques générales du mode DA inverse avec la stratégie de ST. Durant le forward sweep, certaines valeurs intermédiaires sont stockées dans le "stack", par conséquent le pic d'utilisation de mémoire est atteint à la fin de la progression. Après, durant la backward sweep, les valeurs sont déstockées permettant de calculer les dérivées.

Afin d'aborder ce problème, nous avons deux types de stratégie : les stratégies échelle fine et échelle macro. Où la stratégie échelle fine se focalise sur l'optimisation locale, au niveau des instructions. La stratégie échelle macro traite le problème sur les grands segments de code.

Nous menons des recherches pour les deux types de stratégie. Maintenant, nous donnons une vue d'ensemble de nos contributions. Premièrement, nous discutons notre travail sur la stratégie échelle fine. Deuxièmement, nous présentons nos recherches

sur la stratégie échelle macro.

La stratégie échelle fine repose sur une analyse du flot de données. L'analyse du flot de données est un ensemble de techniques qui s'applique au code source du programme. Le principal objectif de ces techniques dans DA est d'aider à générer le meilleur code adjoint possible. En particulier, certaines de ces techniques se concentrent sur le calcul d'un ensemble de valeurs à stocker pour le mode inverse.

Nous améliorons les analyses de flot de données existantes et nous en ajoutons aussi d'autres. Par conséquent, nous sommes capables d'obtenir de plus petits ensembles de valeurs à stocker, et moins d'intructions à implémenter. Par conséquent, notre outil TAPENADE génère des codes de hautes qualité et efficacité.

Au contraire de la stratégie échelle fine, la stratégie échelle macro se concentre sur prendre avantage de l'échange entre stocker et recalculer, où l'échange le plus efficace est situé au niveau des segments de code.

Le méchanisme échelle macro le plus important est appelé "checkpointing". Dans le mode inverse de DA et dans la stratégie ST, certaines valeurs des intructions intermédiaires sont stockées par empilement. Quand ces valeurs sont nécessaires pour calculer des dérivées, elles sont restorées. Alternativement, le processus "checkpointing" consiste à désactiver le stockage de certains segments du code. Quand des valeurs intermédaires sont nécessaires pour calculer les dérivées, certains segments de code sont recalculés et cette fois en activant le stockage activée. Cependant, "checkpointing" ne stocke que ce qui est nécessaire pour effectuer le recalcul du segment choisi pour le "checkpointing", cet ensemble de valeur est appelée "snapshot". En général, cette quantité de données est plus petite que les données stockées durant le calcul du segment, nous gagnons donc de la mémoire. La contrepartie ici est que parce que nous avons besoin de recalculer certains segments, nous avons perdu du temps d'éxecution.

En utilisant une analyse améliorée des stratégies échelle fine, nous pouvons calculer de plus petit "snapshots". De plus, nous introduisons de la flexibilité dans les emplacements des "checkpointing". Auparavant, TAPENADE plaçait les appels au "checkpointing" systématiquement avant chaque appel de "subroutine", mais cette stratégie n'est pas optimale. Nous avons fourni à l'utilisateur la possiblité de sélectionner les emplacements du "checkpointing".

Nous avons mené des expériences sur plusieurs programmes scientifiques afin de valider notre approche. Les résultats sont prometteurs pour les deux niveaux d'optimisation. Les résultats confirment aussi le niveau attendu d'amélioration de ces stratégies.

Chapitre 2

Différentiation Automatique

Ce chapitre présente des concepts et des principes fondamentaux de Différentiation Automatique (DA). La DA est une approche générale pour produire des programmes qui implémentent des dérivées, elle est générale parce qu'elle inclut plusieurs stratégies pour obtenir des dérivées d'un programme. Les dérivées peuvent être générées comme des approximations du premier ordre d'une fonction, d'autre part des dérivées exactes peuvent être obtenus en utilisant la règle de dérivation en chaînes et les règles bien connues du calcul différentiel. L'utilisation de la règle de dérivation en chaîne est l'approche généralement admise.

La règle de dérivation en chaîne peut être employée fondamentalement de deux manières : vers l'avant et vers l'arrière. Vers l'avant et vers l'arrière indiquent la direction de l'application de la règle de dérivation en chaînes en ce qui concerne les entrées et les sorties de la fonction. Cette dichotomie soutient les deux modes de DA les plus classiques, *tangent* et *inverse* mode [8]. Nous avons appelé la tangente le premier mode, celui lié à l'application vers l'avant de la règle de dérivation chaînes, afin d'éviter de futures complications avec la notation. Le mode inverse, celui lié à l'application vers l'arrière de la règle de dérivation en chaînes, suscite le plus d'attention de la recherche dans ce domaine, parce qu'il est employé pour calculer les gradients, qui sont fortement exigés en applications scientifiques.

Avant de présenter les formalisations nous présentons la motivation et le cadre dans le domaine de la DA dans lequel cette thèse s'insère.

2.1 DA: Cadre

Il y a principalement deux manières de calculer les dérivées informatiques, l'une des manières s'appelle *numérique* et l'autre s'appelle *symbolique*. Dans cette thèse,

nous nous concentrons seulement sur la première. Nous abandonnons la dernière en raison de la complexité présentée par la manipulation des formules et de la quantité supplémentaire de mémoire et de durée de calcul nécessaire.

Dans la différentiation numérique que nous avons deux approches principales pour calculer les dérivés, un s'appelle *Divided Differences* (DD) et l'autre s'appelle DA.

Afin d'obtenir des dérivées par l'approche DD, on applique l'équation suivante 2.1.

$$f'(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon} \tag{2.1}$$

Nous abandonnons l'approche DD pour les trois raisons suivantes. D'abord, à la différence de la DA où la précision est liée à la précision de machine, cette approche présente des erreurs de troncature. Par conséquent la précision est limitée par la manipulation de la virgule flottante. Deuxièmement, le choix du ϵ correct dans la formule 2.1 est un processus sans fin d'essais et d'erreur, qui affecte par la suite la précision de dérivées. Troisièmement, au premier abord, cette approche a l'air bon marché du point de vue informatique, elle exige juste deux évaluations du programme original plus quelques opérations élémentaires. Cependant, si les dérivées pour un grand nombre de variables d'entrée sont nécessaires, la performance en termes de temps d'exécution diminue en proportion directe avec le nombre de variables, ce qui est inacceptable dans des applications réelles.

Compte tenu de la discussion ci-dessus, cette thèse est consacrée à l'approche numérique-analytique de la différentiation automatique.

Dans les sections suivantes nous présenterons les principes fondamentaux sur lesquels la DA se base. Nous commençons par la relation entre les programmes informatiques et les fonctions mathématiques. Le reste du chapitre est organisé comme suit. Dans La Section 2.3 nous présentons la règle de dérivation en chaînes. Dans La Section 2.4 nous présentons le mode tangent de la DA. Dans La Section 2.5 nous présentons le mode inverse de DA.

2.2 Programmes et fonctions mathématiques

2.2.1 Programmes et instructions

Nous considérons des programmes en tant que série d'instructions concaténées, où des instructions sont représentées par le symbole I_j, avec $j \in \{1, q\}$. Ainsi, un programme arbitraire P a la forme suivante :

$$P = I_1 \ ; \ I_2 \ ; \ldots ; \ I_j \ ; \ldots ; \ I_{n-1} \ ; \ I_q \tag{2.2}$$

où le point-virgule représente l'opérateur de concaténation.

Les instructions qui implémentent des expressions numériques sont composées d'opérations élémentaires et de fonctions mathématiques données par le langage de programmation (*intrinsic* functions). Dans le meilleur des cas, ceci devrait être tout ce qui est nécessaire pour représenter n'importe quel modèle mathématique, mais ce qui n'est pas étant donné vrai que les programmes ont une complexité élevée et une grande taille. Par conséquent, pour faire face à la complexité, les programmes se servent des structures de commande (par exemple, le rapport conditionnel *if... then... else*, ce que nous appelons un *test*) et les boucles, deux composantes du quatrième groupe d'instructions. En conclusion, pour faire face aux grands programmes, qui peuvent inclure l'utilisation multiple de segments non séquentiels de code, les programmes peuvent être divisés en des morceaux maniables (sous-programmes), ce qui introduit l'utilisation des appels de sous-programme.

Contrôle de flot et graph de flot

Le contrôle de flot détermine le comportement dynamique des programmes, et indique l'existence de plusieurs séquences secondaires d'instructions qui peuvent mener à bien l'exécution du programme. Une séquence secondaire d'instructions s_i, où $s_i \subset P$, s'appelle un *chemin d'exécution* si les instructions dans s_i calculent un résultat valide du programme P pour un certain ensemble d'entrée. Ainsi, un programme avec une structure de contrôle de flot est un ensemble de chemins possibles d'exécution, mais quand un programme est exécuté, seulement un chemin d'exécution est effectué.

Si une instruction I_j d'un programme P implémente un test, l'expression 2.2 ne changera pas, parce que c'est une représentation statique qui cache le flot de l'information dans la structure de contrôle de programme. Ainsi, afin de représenter l'ensemble de chemins d'exécution nous nous servons du *graph de flot* [1, 17] (Figure 2.1).

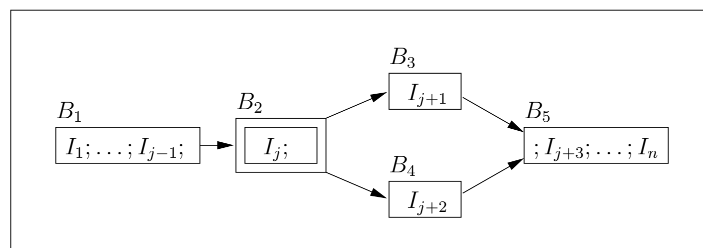

Figure 2.1: Two valid execution paths.

Sur la Figure 2.1, la structure conditionnelle dedans I_j présente deux branches possibles (I_{j+1}, I_{j+2}). Par conséquent, le programme P a les chemins suivants d'exécution : $s_1 = I_1 ; \ldots ; I_j ; I_{j+1} ; \ldots ; I_q$ et $s_2 = I_1 ; \ldots ; I_j ; I_{j+2} ; \ldots ; I_q$.

Afin de simplifier la notation, une séquence d'instructions sans structure de contrôle s'appelle un *bloc de base* B_i, par exemple sur la Figure 2.1 nous avons le bloc $B_1 = I_1 ; \ldots ; I_{j-1}$. En outre, un bloc qui contient seulement un rapport conditionnel (test T_i) nous l'appelons un *bloc d'en-tête de*, par exemple nous avons le bloc B_2 sur la Figure 2.1.

Appels de sous-programmed et the call-graph

Maintenant nous présentons la notation pour manipuler des appels de sous-programme, redéfinissons le programme P comme suit :

$$\begin{aligned} P &= B_1 \ ; \ Call\ B \ ; \ B_2 \ ; \ Call\ C \ ; \ B_3 \\ C &= B_1 \ ; \ Call\ D \ ; \ B_2 \end{aligned} \tag{2.3}$$

où P a un corps principal d'où deux sous-programmes (B et C) sont appelées, et il a trois autres blocs de base, le sous-programme C a juste deux blocs de base, et un appel au sous-programme D.

Les sous-programmes appelés par le programme P peuvent appeler d'autres sous-programmes. Par exemple, le sous-programme C appelle le sous-programme D (figure 2.2). Ce mécanisme s'appelle l'emboîtement. Le niveau de l'emboîtement appelé *profondeur* est arbitraire, et la profondeur est compté à partir du niveau zéro qui est le corps principal du programme au dernier niveau des sous-programmes appelés. Ce processus de décompte est possible à mettre en place parce que la structure d'appel ressemble, en ce qui concerne le temps d'exécution, à la structure arborescente.

Afin de représenter graphiquement la structure d'appel de sous-programmes des programmes, nous employons le call-graph [1, 17] (cf. figure 2.2).

Sur la figure 2.2, nous représentons les sous-programmes avec les boîtes désignées par les noms donnés en forme algébrique (formule 2.3), les appels sont représentés par des flèches. La direction de la flèche indique la direction d'appel, par exemple, le programme P appelle le sous-programme B .

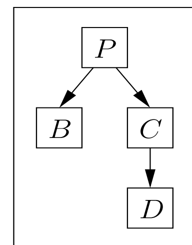

Figure 2.2: Call-graph: nested calls.

Notez que la structure de commande est étroitement liée à la structure d'appel du programme, par exemple, un sous-programme pourrait être appelé par une boucle ou par une branche d'un test. Malheureusement, la représentation d'appel graphique n'illustre pas ce comportement dynamique dans des sous-programmes. Par conséquent, nous emploierons l'appel graphique pour montrer seulement la structure statique des appels dans un programme, sans appels réitérés ou récursifs, et en supposant que tous les sous-programmes s'appellent au moins une fois s'ils sont dans des instructions conditionnelles.

2.2.2 Fonctions mathématiques implémentées par programmes

Les modèles ou les fonctions mathématiques se composent des fonctions mathématiques élémentaires. Nous dénotons ces fonctions élémentaires par $f_i()$, avec $i \in \{1, p\} \subset I\!N$. Par exemple, nous définissons la fonction F, qui reçoit un vecteur X comme entrée, ce vecteur se compose de variables d'un espace d'entrée de dimension n. La fonction F retourne un vecteur Y qui appartient à un espace de dimension m. Ainsi, F est exprimé comme suivant :

$$Y = F(X) = f_p(\dots f_i(\dots f_1(X))) \tag{2.4}$$
$$F : X \in I\!R^n \to Y \in I\!R^m$$

Après le calcul de $f_1(X)$ le vecteur des variables de sortie devient le vecteur d'entrée, X_2, pour la fonctionner $f_2()$. Ce processus est répété jusqu'au calcul de la dernière fonction $f_p(X_p)$. Ce mécanisme pourrait être reprśent'e par la formule 2.5, où le symbole $\circ$ est l'opérateur de composition.

$$Y = F(X) = f_p(X_p) \circ \dots \circ f_i(X_i) \circ \dots \circ f_1(X) \tag{2.5}$$

où le vecteur des variables intermédiaire $X_i = f_{i-1}(X_{i-1}) \circ \cdots \circ f_1(X)$, pour chaque $i \in \{1, p\}$, est obtenu par l'évaluation de la fonction élémentaire $f_{i-1}()$. Ce vecteur sert comme entrée pour la fonction $f_i()$.

2.3 La règle de dérivation en chaînes

La règle de dérivation en chaînes peut être appliquée seulement si les fonctions impliquées sont différentiables. Par exemple, pour une fonction $F(X) = f_2(f_1(X))$, avec $f_2()$ et $f_1()$ sont différentiables, cette règle s'ecrit comme suit:

$$Y' = F'(X) = (f_2'(X_2) \circ f_1(X)) \times f_1'(X) \tag{2.6}$$

Dans le contexte de ce travail, où la fonction F est un modèle mathématique implémenté par un programme, nous supposons que chaque fonction mathématique élémentaire $f_i()$ de la fonction F est différentiable. Par conséquent, F est aussi différentiable. De plus sa forme différenciée s'exprime comme suit :

$$Y' = F'(X) = f'_p(X_p) \times \ldots \times f'_i(X_i) \times \ldots \times f'_1(X) \tag{2.7}$$
$$F' : X \in I\!R^n \to Y' \in I\!R^{m \times n}$$

où $f'_i()$ (resp. F') est la matrice Jacobienne de f'_i (resp. F). $X_i = f_{i-1}(X_{i-1}) \circ \cdots \circ f_1(X)$ $\forall i \in \{1, p\}$ est l'ensemble de variables intermédiaires calculées par la composition des fonctions originales.

La règle de dérivation en chaînes peut également être appliquée aux fonctions définies par morceaux, mais dans ce cas la fonction doit être continu aux points de raccordement pour asssurer des conditions minimales pour la différentiation. Néanmoins, le calcul des dérivées autour ou pour ces points de raccordement pourrait être incorrecte. Ce problème est la motivation pour le troisième chapitre de cette thèse.

2.4 Le mode tangent de DA

2.4.1 Définition et modèle

Le mode *tangent* est la première spécialisation du cadre décrit dans la section précédente. La motivation pour ce modèle provient du fait que le calcul du Jacobian est trop couteuse, et souvent seulement quelques directions dans l'espace d'entrée sont nécessaires. Le mode tangent produit un code permettant le calcul des dérivés directionnelles. Par consequent, afin de produire le code différencié en mode tangent, la formule 2.7 est modifiée pour présenter l'effet directionnel. Ainsi nous obtenons la forme suivante :

$$dY = F'(X) \times dX = f'_p(X_p) \times \cdots \times f'_i(X_i) \times \cdots \times f'_1(X) \times dX \tag{2.8}$$
$$F' \times dX : X, dX \in I\!R^n \to dY \in I\!R^m$$

où X représente les variables d'entrée, et dX un vecteur colonne qui tient une direction dans l'espace d'entrée, toutes les deux sont de dimension n; également dY est un vecteur ligne de dimension m représentant les variables de sortie. Dans la Formule 2.8 nous avons employé la propriété d'associativité de la multiplication des matrices, ainsi son calcul prend $O(pn^2)$ au lieu de $O(pn^3)$ que prend un algorithme linéaire classique d'algèbre.

Le modèle de DA qui met en application la formule 2.8 est de la forme suivante :

$$P' = I'_1 \ ; \ I_1 \ ; \ldots ; \ I'_j \ ; \ I_j \ ; \ldots ; \ I'_q \tag{2.9}$$

Dans la Formule 2.9 les produits matrice-vecteur sont notées par I'_j avec $j \in \{1, q\}$.

2.4.2 Example

Le Tableau 2.1 illustre la différentiation automatique en mode tangent d'un programme exemple. La liste d'évaluation d'instructions originale est donnée dans la partie gauche de du Tableau 2.1, la liste différenciée d'évaluation est donnée dans la partie droit du Tableau 2.1.

Original Code	Tangent Differentiated Code
`subroutine F(x1,x2,y)`	`subroutine F_d(x1,x1d,x2,x2d,y,yd)`
	`i1d = x1d * x1 + x1 * x1d`
`i1 = x1 * x1`	`i1 = x1 * x1`
	`i2d = x2d * x2 + x2 * x2d`
`i2 = x2 * x2`	`i2 = x2 * x2`
	`i3d = - x2d`
`i3 = 1 - x2`	`i3 = 1 - x2`
	`i4d = i1d + i2d`
`i4 = i1 + i2`	`i4 = i1 + i2`
	`i5d = i4d * COS(i4)`
`i5 = sin(i4)`	`i5 = SIN(i4)`
	`i6d = i5d * i3 + i5 * i3d`
`i6 = i5 * i3`	`i6 = i5 * i3`
	`yd = (i6d * i4 - i6 * i4d) / i4**2`
`y = i6 / i4`	`y = i6 / i4`

Tableau 2.1: Code différencié tangent produit automatique.

Dans le code différencié, yd présente la dérivée directionnelle (dans le direction $(x1d, x2d)$) de y (variable de sortie) par rapport á $(x1, x2)$ (variable d'entrée pour F). . Les instructions avec le suffixe d implémente le dérivé directionnel de l'instruction correspondante. Notez que pour le code a présenté dans ce travail, les fonctions intrinsèques ($sin()$, $cos()$) sont écris en majuscules, et les declarations des variables sont omis.

2.5 Le mode inverse de DA

2.5.1 Définition et modèle

Le but principal du mode inverse est de calculer les combinaisons linéaires des colonnes du Jacobian, et particuliérement le gradient. Ce dernier defini comme etant le vecteur colonne dont les components sont les dérivées partielles, constitue un outil principal permettant de caractériser les solutions et de produire les algorithmes de resolution des problèmes scientifique. Considérons $\overline{Y}$ un vecteur poid dans l'espace de sortie de F. Le produits scalaire de $\overline{Y}$ avec $Y = F(X)$ définit un résultat scalaire,

$$\bar{Y}^t \times Y = Y^t \times \bar{Y} \tag{2.10}$$

remplacement $Y = F(X)$ dans 2.10 nous obtenons

$$\bar{Y}^t \times Y = F^t(X) \times \bar{Y} \tag{2.11}$$

le gradient $\bar{Y}^t \times Y$ s'écrit comme suit (employant la règle de dérivation en chaînes)

$$\bar{X} = F'^t(X) \times \bar{Y} = f_1'^t(X) \times \ldots \times f_i'^t(X_i) \times \ldots \times f_p'^t(X_p) \times \bar{Y} \tag{2.12}$$

$$F' \times \bar{Y} : \bar{Y} \in I\!R^m, X \in I\!R^n \to \bar{X} \in I\!R^n$$

où chaque $f_j'^t()$ et $F'^t(X)$ sont les transposés des Jacobians. X_i sont calculés comme ils sont calculés comme dans la règle de dérivation en chaînes, cela est par l'évaluation de la fonction de l'entrée originale. Cependent que dans la formule 2.12 l'ordre du calcul est inversé, ainsi la première fonction à calculer est $f_p'^t(X_p)$ dont l'entrée est X_p qui est obtenu (voir section 2.3) après le calcul de toutes les fonctions originales $f_i()$ excepté $f_p()$. Par conséquent, pour le calcule du mode inverse, nous devons calculer d'abord le programme original et ensuite la formule 2.12.

Le modèle de DA qui met en application la formule 2.12 est de la forme suivante :

$$\bar{P} = \overrightarrow{P}; \overleftarrow{P} = I_1 \;; \ldots ;\; I_j \;; \ldots ;\; I_{q-1} \;;\; I_q' \;; \ldots ;\; I_j' \;; \ldots ;\; I_1' \tag{2.13}$$

où le programme différencié en mode inverse $\bar{P}$ se compose de deux parties : la première s'appelle le *forward sweep* $\overrightarrow{P}$, qui est fondamentalement la suite des instructions originales. La deuxième partie s'appelle *backward sweep* $\overleftarrow{P}$, et il se compose d'instructions qui mettent en application les produit entre le transposé de Jacobian et un vecteur. Remarquablement, la dernière instruction de forward sweep n'est jamais exigée dans le backward sweep. C'est parce que l'ensemble d'entrée de variables pour la première instruction de backward sweep est la sortie de forward sweep sans la dernière instruction. Par conséquent ce que cette dernière instruction calcule n'est pas exigé (à moins que si la dernière instruction est définie $I_n = \sqrt{(I_{n-1})}$).

2.5.2 Example

Dans Le Tableau 2.2 la version différenciée en mode inverse du sous-programme F (F_b), reçoit comme variables d'entrées $x1$ et $x2$, et le poids du sortie représenté par yb. Les sorties sont les composantes du gradient (dans cet exemple) $x1b$ et $x2b$. Les instructions avec le suffixe b implémentent les dérivés correspondantes aux instructions originales.

Original Code	Sweep	Reverse Differentiated Code
`subroutine F(x1,x2,y)`		`subroutine F_b(x1,x1b,x2,x2b,y,yb)`
	f	
`i1 = x1 * x1`	o	`i1 = x1 * x1`
`i2 = x2 * x2`	r	`i2 = x2 * x2`
`i3 = 1 - x2`	w	`i3 = 1 - x2`
`i4 = i1 + i2`	a	`i4 = i1 + i2`
`i5 = sin(i4)`	r	`i5 = SIN(i4)`
`i6 = i5 * i3`	d	`i6 = i5 * i3`
`y = i6 / i4`		`y = i6 / i4`
	b	
	a	`i6b = yb / i4`
	c	`i5b = i3 * i6b`
	k	`i4b = COS(i4) * i5b - i6`
	w	`* yb/i4**2`
	a	`i3b = i5 * i6b`
	r	`i1b = i4b`
	w	`i2b = i4b`
	d	`x2b = x2b + 2 * x2 * i2b - i3b`
		`x1b = x1b + 2 * x1 * i1b`

Tableau 2.2: Automatic generated reverse differentiated code.

Notons que les variables exigées dans le backward sweep peuvent éventuellement être redéfinies dans le forward sweep, et par conséquent les valeurs qu'elles tenaient ne sont plus disponibles pour le backward sweep. Ceci contitue la difficulté majeure dans la DA en mode inverse. Pour y faire face nous avons une stratégie qui s'appelle *Stocker-Tout* (ST). ST consiste en stockant toutes les valeurs, pendant le forward sweep, les rendant ainsi accessibles dans le backward sweep. Le probléme de cette stratégie est qu'elle consomme beaucoup d'espace mémoire. Cette discussion sert d'introduction au quatrième chapitre de cette thèse.

Chapitre 3

Le domaine de la validité des dérivés

Les outils automatiques de différentiation (DA) supposent la differentiabilité de la fonction implémentée par le programme donné. Cette hypothèse est fondamentale, parce que le mécanisme sous-jacent de la DA est l'application systématique de la règle de dérivation en chaînes, qui suppose la differentiabilité de chaque fonction composante. Cependant, parfois des fonctions sont composée avec des fonctions élémentaires non-lisses, qui peuvent mener à la perte la differentiabilité globale. Une autre source des dérivées fausses sont les commutateurs dans le flot de commande, venant principalement de rapports conditionnels. Ces commutateurs rendent la plupart des programmes seulement différentiables par morceaux. Dans ces cas, les dérivées près des commutateurs sont parfois inconsistantes, parce qu'elles sont calculées par différents ensembles d'instructions. En outre, les programmes différenciés peuvent renvoyer des dérivées où la fonction implémentée n'est pas différentiable [22]. Malheureusement, l'utilisation journalière de la DA ignore ces problèmes, ainsi les problèmes qui devraient être fondamentaux deviennent objet d'attention seulement quand les résultats ne sont pas ceux qui étaient prévus.

Pour aborder ce problème, dans ce chapitre nous proposons un modèle général pour évaluer, avec la dérivée, la taille du voisinage différentiable autour de l'entrée courante où les dérivées retournées par la DA ne souffrent pas du non-differentiabilités. Ce " voisinage fiable "est essentiel pour employer les dérivées de façon consistante. Nous étudions plusieurs méthode pour calculer ce voisinage et pour étudier leur complexité. Avec ce modèle général, le programme différencié calcule, avec les dérivées habituelles, quelques informations d'exécution supplémentaires sur la fiabilité des dérivées, sans diminuer le taux d'efficacité du code différencié. En conclusion, nous présentons une implémentation et des expériences faites avec notre outil DA TAPENADE.

Dans la section suivante, nous présentons le problème de la validité, en se concentrant particulièrement sur les commutateurs de flot de commande, mais n'omettant pas les problèmes inhérents à l'implémentation des fonctions mathématiques fondamentales par les langages de programmation. Le reste du chapitre est organisé comme suit. Dans la Section **??** nous présentons l'analyse en profondeur du problème de validité. Dans la section 3.2 nous définissons l'information de validité et sa propagation. Dans la section 3.3 nous proposons la première méthode pour faire face au problème de validité. Dans la section 3.4 nous proposons la deuxième méthode. Dans la section 3.5 nous présentons les résultats expérimentaux. Enfin dans la section 3.6 nous discutons les résultats et donnons quelques perspectives au sujet des méthodes proposées.

3.1 Le Problème de Validité

Fournir des dérivées précises et fiables est l'un des objectifs de la communauté de DA, ainsi les outils de DA devraient fournir des dérivées valides dans le domaine d'entrée des applications. Malheureusement, les modèles courants de DA n'incluent pas la vérification de la differentiabilité des fonctions. En outre, l'expérience prouve que le differentiabilité des fonctions implémentées par des programmes peut être assez facilement corrompu, et pour empirer le problème, les moyens de cette corruption sont habituellement négligés, en particulier à l'étape de développement.

Nous avons identifié deux sources principales de non-differentiabilité dans les programmes : d'abord, l'utilisation des fonctions intrinsèques d'un langage de programmation qui ne sont pas différentiables pour leurs entrées courantes. En second lieu, les changements du flot de commande qui cassent la continuité des fonctions.

3.1.1 La Differentiabilité des fonctions intrinsèques

Certaines des fonctions intrinsèques des langages de programmation cachent des discontinuités et des non-differentiabilit'es. Un liste courte et incomplète de fonctions intrinsèques dangereuses est : $abs(x) = |x|$, $sqrt(x) = \sqrt{x}$, $|x, y| = \sqrt{x^2 + y^2}$, max(x,y), min(x,y), sign(x), heav(x).

Ces fonctions sont généralement employées, parce qu'elles sont fournies par la langage de programmation, mais leurs implémentations et les définitions de leurs dérivées exigent une attention particulière . Ceci vient du fait que toutes incluent des discontinuités, ou que leurs dérivées sont bien connues pour leurs problèmes de differentiabilité.

Une façon de traiter ces fonctions problématiques est de changer leur implémentation mais non leur signification mathématique. Par conséquent l'implémentation de la

<table>
<tr><th>Intrinsic</th><th>Tangent Differentiated Code</th></tr>
<tr><td><pre>C abs()
  i1 = ABS(x1)</pre></td><td><pre>C derivative of abs()
  IF (x1 .GE. 0.) THEN
    i1d = x1d
    i1 = x1
  ELSE
    i1d = -x1d
    i1 = -x1
  END IF</pre></td></tr>
<tr><td><pre>C sqrt()
  i1 = SQRT(i1)</pre></td><td><pre>C derivative of sqrt()
  IF (i1d .EQ. 0.0 .OR. i1 .EQ. 0.0) THEN
    i1d = 0.0
  ELSE
    i1d = i1d/(2.0*SQRT(i1))
  END IF
  i1 = SQRT(i1)</pre></td></tr>
<tr><td><pre>C max(,)
  i2 = MAX(i1, x1)</pre></td><td><pre>C derivative of max(,)
  IF (i1 .LT. x1) THEN
    i2_d = x1d
    i2 = x1
  ELSE
    i2_d = i1d
    i2 = i1
  END IF</pre></td></tr>
</table>

Tableau 3.1: Remplacements de fonctions intrinsèques et de leurs dérivées pour la DA.

fonction peut par sa forme intrinsèque (originellement implémenté par des bibliothèques du langage de programmation) peut être remplacée par sa forme *conditionnelle*. Cette dernière s'appelle la forme conditionnelle parce que dans la plupart des cas la nouvelle exécution inclut des rapports conditionnels pour surmonter les problèmes inhérents des fonctions, ainsi le ou les point(s) de non-differentiabilité sont pris en considération.

Le mécanisme précédent permet d'implémenter des dérivées pour ces fonctions différentielles problématiques, et il préserve les résultats originaux de calcul sans perte d'efficacité. En outre, le mécanisme peut être niché, et ainsi permettre toutes les combinaisons possibles de ces fonctions. Le Tableau 3.1 montre l'implémentation de certaines des fonctions de remplacement des fonctions problématiques, avec leurs

dérivés.

Une fois que le mécanisme ci-dessus est appliqué, la première source des problèmes de differentiabilité devient la deuxième source des problèmes, parce que tous les remplacements incluent des rapports conditionnels. Par conséquent, nous supportons l'idée que la deuxième source des problèmes est le seul fondamental, et c'est le problème que nous étudierons, c'està-dire les dérivées calculées par des changements du flot de commande.

3.1.2 Differentiabilité des rapports conditionnels

Parfois les dérivées dépendent des tests. Dans ces cas nous avons des dérivées différentes selon la commutation du test. Si le test a la forme
"*if ... then ... else*" alors nous avons deux ensembles d'instructions de dérivées, chacune correspondant aux branches du test. Rappelons la notation utilisée pour les programmes : les programmes sont composés de blocs d'instructions B_i et de tests T_i.

Après différentiation, nous obtenons les blocs B'_i, sui sont les blocs différenciés correspondant aux B_i, qui contiennent les instructions originales plus les instructions de dérivation. Les instructions conditionnels demeurent les mêmes, c.-à-d., qu'elles sont représentées par T_i dans le code différencié.

Par exemple, dans la figure 3.1 le flot de commande suivra les instructions B'_3 ou B'_4 selon le signe de l'essai.

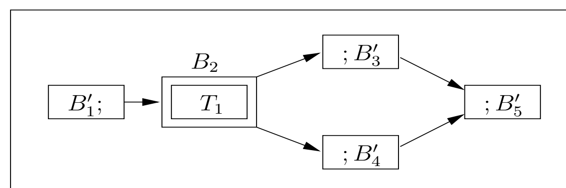

Figure 3.1: Differentiated flow-graph for example.

Le problème survient lorsque pour une certaine entrée, le programme évalue la séquence des blocs $B'_1; T_1; B'_3; B'_5$, et pour une autre valeur légèrement différente d'entrée, le programme évalue $B'_1; T_1; B'_4; B'_5$. La différence entre la première et deuxième valeur d'entrée peut être très petit, mais assez pour commuter le test T_i. Les instructions dérivées dans B'_3 et B'_4 peuvent être totalement différentes. Ainsi, les petits changements des valeurs d'entrée peuvent commuter le test en renvoyant des

dérivés complètement différents. Notoms qu'il y a un cas spécial quand les instructions dans les deux branches sont identiques.

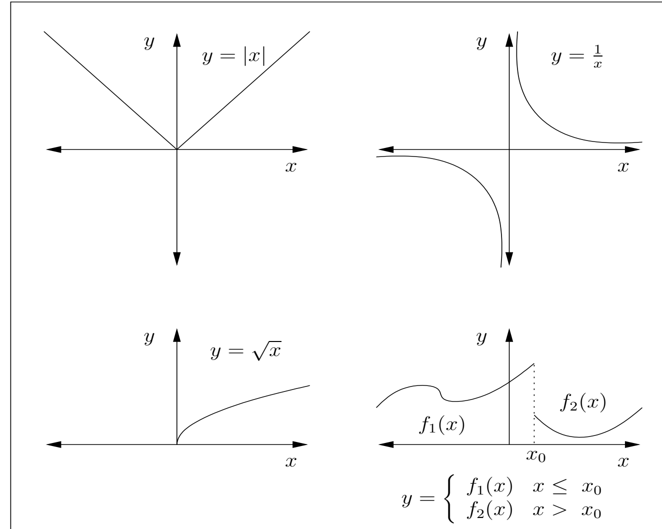

Figure 3.2: Examples of problematic functions.

En général, nous pouvons produire toutes formes d'instructions conditionnels basées sur l'ensemble minimal de formes présentées dans la figure 3.2. Sur cette figure la figure inférieure droite représente par une fonction définie par morceaux, et les autres formes peuvent être modifiées en quelque chose de similaire à cette dernière. La ramification peut se développer arbitrairement, et les branches peuvent avoir des branches elles-mêmes, donc les ensembles de dérivés retournés par DA peuvent être comptés comme chemins d'exécution d'un organigramme. La propagation d'information est le facteur principal dans ce problème, étant donné que les codes dans la réalité peuvent avoir des centaines de test, et beaucoup d'entre eux sont reliée aux dérivées.

Il est important de remarquer le fait que le problème des dérivées inconsitente est provoqué par l'exécution répétée du code differencié, qui renvoie différents dérivées pour une entrée plutôt semblable. Pour une valeur particulière d'entrée, l'exécution suit un chemin d'instructions. Mais dans une deuxième exécution du code differencié, avec une valeur semblable d'entrée, le chemin a suivi des changements en raison d'une commutation de test. Par conséquent les dérivées retournées par DA sont differentes ou radicalement différentes de ce qui a été obtenu à la première exécution. Le problème est inhérent au comportement en cours d'exécution des programmes, donc il n'est pas possible de prévoir ce problème, car il dépend de l'entrée donnée pour chaque exécution.

Nous croyons que l'information requise pour assurer la validité des dérivées pourrait être employée pour améliorer quelques algorithmes, par exemple les algorithmes qui se fondent sur un critère qui dépend des dérivées pour diriger le processus itératif.

Dans la section suivante, nous présentons notre approche du problème de validité des dérivées. Cette approche vise à introduire un nouveau modèle de la DA qui ne change pas la structure du programme, mais ajoute pourtant des informations valables sur les dérivées concernant la durée d'exécution sans diminuer la performance.

3.2 Définir et propager l'information de validité

Dans cette section, nous proposons un nouveau modèle de DA qui inclut une méthode pour valider les dérivées. Notre approche n'essaie pas d'établir la non differentiabilité près d'un test, mais d'accepter plutôt que n'importe quel test peut causer une non-differentiabilité. Par conséquent, notre approche est juste d'évaluer juste à quels points les test sont proches de leurs valeurs de commutation.

Pour valider les dérivées, nous évaluons l'intervalle autour des données d'entrée où aucun problème de non-differentiabilité ne survient. En pratiqueme, ceci nécessite l'analyse de chaque instruction conditionnelle en cours d'exécution, afin de trouver pour quelles données il commutera, et propager cette information comme contrainte sur les données d'entrée. Nous discutons également la complexité de ce mode et de quelques solutions de rechange. En conclusion, nous développons un mode qui étudie la validité des dérivées ; l'exécution de cette méthode présente une prolongation du mode tangente de DA.

Nous concevons une méthode qui renvoie un certain intervalle des solutions où les dérivées ne sont pas mises en danger par la commutation conditionnelle. Pour faire que, nous développons une formalisation qui relie les tests, les valeurs d'entrée et les variations des valeurs d'entrée. Notre idée est d'évaluer le plus grand intervalle autour des données d'entrée courantes, de sorte qu'il n'y ait aucun problème de differentiabilité si l'entrée demeure dans cet intervalle. Dans le cas où cet intervalle est notamment trop petit, ce sera un avertissement à l'utilisateur contre une utilisation non valide de ces dérivées.

3.2.1 Définition de l'Information de Validité

Comme nous l'avons présenté dans la section précédente, les programmes peuvent être vus comme concaténation des blocs B_i et des essais T_i. La notation pour des essais est étendue à $T_i(X_i)$, soulignant le fait que les tests sont construits à partir de variables intermédiaires X_i dans le bloc (B_i) avant le test. Par conséquent, un

programme arbitraire P a la forme suivante :

$$P = (B_1(X), X_1) \ ; \ T_1(X_1) \ ; \ldots ; \ (B_n(X_{n-1}), X_n) \ ; \ T_n(X_n) \ ; \ (B_{n+1}(X_n), Y) \quad (3.1)$$

où la paire $(B_i(X_{i-1}), X_i)$ décrit l'entrée (X_{i-1}) pour le bloc B_i et sa sortie est l'ensemble mis à jour des variables intermédiaires X_i.

Lorsque l'on applique la DA, le nouveau programme augmenté P' inclut les instructions de dérivées et la structure originale des commandes du programme, pour ce cas-ci la structure est donnée dans la formule 3.1.
$P' = (B'_1(X, dX), X_1, dX_1); \ T_1(X_1) \ ; \ldots$

$$\ldots ; \ (B'_n(X_{n-1}, dX_{n-1}), X_n, dX_n) \ ; \ T_n(X_n) \ ; \ (B'_{n+1}(X_n, dX_n), Y, dY) \quad (3.2)$$

où l'expression $(B'_i(X_{i-1}, dX_{i-1}), X_i, dX_i)$ a comme entrée les variables intermédiaires X_{i-1}, dX_{i-1} avant le bloc B_i, et comme sortie les variables mises à jour X_i, dX_i. L'expression décrit la relation différentielle entre dX_{i-1} et dX_i utilisant l'approximation du premier ordre suivante,

$$dX_i = J(B_i(X_i)) \times dX_{i-1} \quad (3.3)$$

En outre, la formule 3.3 montre la propagation des variables intermédiaires X_i par l'évaluation du jacobien du bloc $B_i()$.

Afin de décrire comment la propagation des variables intermédiaires est liée aux tests de commutation, considèrons un test $T_i(X_i)$ isolé. Il emploie seulement des variables de X_i, et nous pouvons admettre sans perte de généralité, que T_i est positif, qui donne

$$T_i(X_i) \geq 0 \quad (3.4)$$

L'analyse que nous avons développée cherche à savoirdans quelle mesure les variables intermédiaires X_i peuvent changer sans commuter le test, où ce changement est représenté par dX_i. En ajoutant à la formule 3.4 la variation dX_i, nous obtenons :

$$T_i(X_i + dX_i) \geq 0 \quad (3.5)$$

En développant la formule 3.5 comme une approximation du premier ordre, nous obtenons :

$$\begin{aligned} T_i(X_i) + < T'_i(X_i), dX_i > \geq 0 \\ < T'_i(X_i), dX_i > \geq -T_i(X_i) \end{aligned} \quad (3.6)$$

où l'opérateur $<, >$ est le produit point.

Le développement récursif de la formule 3.3 décrit comment dX_i dépend différentiellement (au premier ordre) de l'entrée X et dX :

$$dX_i = J(B_i(X_i)\ ;\ldots;\ B_1(X)) \times dX = J(B_i(X_i)) \times \ldots \times J(B_1(X)) \times dX \quad (3.7)$$

En utilisant la formule 3.6 et la formule 3.7, nous pouvons affirmer que la contrainte sur dX pour laquelle le test T_i ne commute pas, est :

$$< J(T_i(X_i)),\ J(B_i(X_i) \times \ldots \times J(B_1(X)) \times dX > \geq\ -T_i(X_i) \quad (3.8)$$

où la dérivée du test T_i s'exprime comme $T_i'(X_i) = J(T_i(X_i))$, ceci est rendu possible par lefait qu'un test est implémenté comme instruction.

La formule 3.8 est une contrainte sur dX, qui représente un demi-espace dans l'espace d'entrée. Si les entrées demeurent dans le demi-espace qui satisfait la contrainte, la variation dX est valide. En termes informatiques, pour un X donné dans le demi-espace qui satisfait les contraintes, le programme suit le même chemin d'exécution, ou demeure en d'autres termes dans la même branche de test, donc les variations dX sont consistantes.

3.2.2 Propagation se l'Information de Validité

Pour le programme entier, les dérivées calculées seront valides si la variation dX de l'entrée X satisfait toutes les contraintes (formule 3.8) pour chaque test T_i. Ceci fournit un ensemble de contraintes sur dX, ou un ensemble de demi-espaces. L'intersection des demi-espaces compose le voisinage autour des valeurs d'entrée qui renvoie des dérivées valides ; ce voisinage est ce que nous appelons *l'information de validité*.

Pour calculer l'information de validité pour le programme entier, chaque test est analysé systématiquement ; une fois que l'information de validité est calculée pour un test, l'information doit être combinée avec l'information des tests précédents, et ainsi de suite jusqu'à la fin du programme. Ce mécanisme peut être mis en application de plusieurs manières. Dans la prochaine section, nous discuterons une première approche basée sur mode inverse de la DA.

3.3 Propagation Arrière des Demi-Espaces (PADE)

3.3.1 Définition

Pour appliquer la méthode précédente, nous devons calculer plusieurs jacobiens, mais le calcul des jacobiens complets est cher, donc nous avons étudié des manières meilleur marché pour appliquer la méthode. Par exemple, il est possible de modifier la formule 3.8 afin de permettre le calcul des jacobiens en utilisant le mode inverse

de la DA. Alternativement, le facteur de gauche du produit scalaire dans le terme de gauche (lhs) de la formule 3.8 peut être calculé en utilisant le mode vers l'avant de la DA. Rappelons ce que nous avons présenté dans **??**, les coûts informatiques du mode vers l'avant sont proportionnels à la dimension de l'espace d'entrée.

Observons la formule 3.8, rappelons que nous devons la résoudre pour dX. Ceci signifie que nous devons isoler dX. Une manière efficace de le faire est de transposer les jacobiens dans le produit scalaire correspondant à l'équation équivalente :

$$dX^t \times J(B_1(X))^t \times \ldots \times J(B_i(X_i))^t \times J(T_i(X_i)) \geq -T_i(X_i) \tag{3.9}$$

$$dX_i^t \times J(T_i(X_i)) \geq -T_i(X_i) \tag{3.10}$$

loù dans la formule 3.10, le calcul de la relation différentielle entre dX et dX_i par X (Formule 3.7) est effectué vers l'arrière. Ceci s'écrit comme suit :

$$dX_i^t = (J(B_i(X_i)\ ;\ldots;\ B_1(X)) \times dX)^t = dX^t \times J(B_1(X))^t \times \ldots \times J(B_i(X_i))^t \tag{3.11}$$

où en arrière signifie que le premier jacobien à calculer est le jacobien du bloc juste avant le test. L'expression $J(B_1(X))^t \times \ldots \times J(B_i(X_i))^t \times J(T_i(X_i))$ dans la formule 3.9 est directement calculée par le mode inverse de la DA.

Dans cette approche, la contrainte sur dX est calculée de manière efficace concernant le durée d'exécution, aussi, l'espace des solutions est une représentation exacte (au premier ordre), un demi-espace, donc une fois que toutes les contraintes sont combinées l'espace global des solutions sera précis. Malheureusement, le mode inverse est cher en termes de consommation de mémoire, parce qu'il exige le stockage d'un grand nombre de variables intermédiaires. Par conséquent, avant d'implémenter le modèle, nous discuterons des solutions de rechange pour réduire les calculs de la formule 3.9.

3.3.2 Problèmes avec PADE

Nous considérons le modèle de la section 3.3.1 complet dans le sens qu'il renvoie une contrainte sur dX pour chaque test produit pendant l'exécution du programme. Cependant, dans de vraies situations, le nombre de tests est si grand que ce modèle complet n'est pas pratique, parce que l'analyse de chaque test mène au calcul de la formule 3.9, qui requiert de l'espace mémoire.

3.4 Propagation vers l'Avant Directionnelle (PAD)

La taille de l'ensemble de contraintes demeure un grand sujet de préoccupation, mais parmi les manières de réduire la taille de problème, il en existe est une qui mène à l'utilisation de la PAD. C'est-à-dire, qu'il est possible de choisir certaines directions

de dérivation, ou même mieux de laisser l'utilisateur identifier et choisir certaines directions de l'entrée. Ceci permet de simplifier les contraintes, et de se concentrer sur le domaine approprié de la validité. En outre, nous pouvons employer le mode vers l'avant de la DA parce que les dérivées nécessaires seront les dérivées directionnelles.

3.4.1 Définition

Le modèle que nous avons présenté dans la section 3.3 est cher en temps de consommation et d'exécution de mémoire. Pour parer à ceci, nous proposons ici une stratégie qui se concentre sur les dérivées directionnelles. Le but est de fournir à l'utilisateur des informations sur la validité des dérivées concernant des directions spécifiques dans l'espace d'entrée. Cette stratégie retourne un espace exact de solution, mais il est restraint à une certaine direction spatiale.

Nous rappelons la formule 3.8, et développons le produit scalaire :

$$J(T_i(X_i))^t \times J(B_i(X_i) \times \ldots \times J(B_1(X)) \times dX \ \geq \ -T_i(X_i) \tag{3.12}$$

où dX peut être décomposé comme $dX = d\hat{X} \times \beta_i$, avec $d\hat{X}$ représentant la direction de la variation de l'entrée, et β_i étant la grandeur scalaire qui donne l'amplitude de cette variation, en particulier β_i est lié au test $T_i()$. En développant la formule 3.12 avec la décomposition dX , nous obtenons :

$$J(T_i(X_i))^t \times J(B_i(X_i)) \times \ldots \times J(B_1(X)) \times d\hat{X} \times \beta_i \geq -T_i(X_i) \tag{3.13}$$

en isolant β_i dans la formule 3.13, nous obtenons :

$$\beta_i \geq \frac{-T_i(X_i)}{J(T_i(X_i))^t \times J(B_i(X_i)) \times \ldots \times J(B_1(X)) \times d\hat{X}} \tag{3.14}$$

La formule 3.14 est la contrainte que β_i doit satisfaire pour le test T_i, ainsi β_i contient l'information sur combien la variation dX d'entrée peut augmenter en suivant la direction donnée $\hat{X}$ de l'espace d'entrée.

Pour calculer le domaine directionnel de la validité autour de l'entrée X concernant une certaine direction, β_i doit être calculé pour chaque test dans le programme, et tous les β_i doivent être recueillis et combinés dans une contrainte simple sur β.

Notez que la répétition du procédé pour toute la base cartésienne dans l'espace d'entrée renvoie une information moins précise que le modèle de la section 3.3, comme nous pouvons l'observer sur la figure 3.3.

La figure 3.3 montre les intervalles représentés par des flèches commençant sur l'entrée (point noir), par exemple les directions d'entrée dans cet exemple sont la base

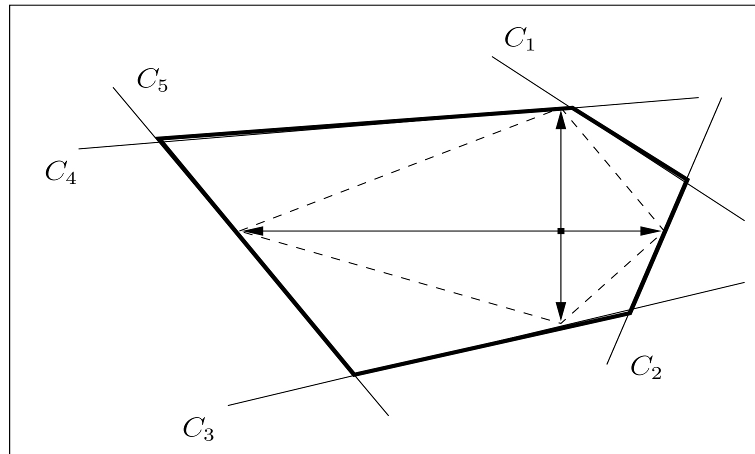

Figure 3.3: Représentation directionnelle pour un exemple arbitraire.

cartésienne. La forme intérieure (lignes pointillées rectangle) est une approximation qui peut être déduite en utilisant la représentation d'intervalle. Cette déduction tire profit de la convexité de l'espace des solutions (lignes noires "bold").

La performance de la méthode est meilleure que la méthode de PADE par deux aspects, mémoires et temps. Ceci parce que la PAD calcule certaines directions dans l'espace d'entrée, et que le modèle de DA derrière le processus est le mode vers l'avant, qui est meilleur marché pour calculer que le mode inverse, qui est derrière PADE. Malheureusement, PAD est moins précis, même si nous projetons un polyèdre des frontières de la solution directionnelle (comme sur la figure 3.3) l'espace de solution sera un sous-ensemble de l'espace exact de solution.

3.4.2 Problèmes avec la PAD

PAD renvoie l'information de la validité de la précision acceptable pour certaines directions dans l'espace d'entrée. Malheureusement, comme cela peut se produire pour un grand code ou un code avec un espace d'entrée avec beaucoup de dimensions, le coût d'application du modèle est encore élevé, et peut être plus que le coût que les utilisateurs peuvent accepter.

Pour faire face à ce problème, nous avons principalement deux options pour réduire le coût du modèle de PAD, le premier doit choisir les tests les plus appropriés, ce qui exige une bonne connaissance du code, et par la suite peut mener à beaucoup de travail manuel, si le nombre de tests est grand. En deuxième option, l'utilisateur peut identifier les directions les plus appropriées dans l'espace d'entrée, ce qui est une manière bon marché de réduire le coût, parce que la décision des directions appropriées

est prise juste une fois, avant l'application du modèle de PAD, qui est automatique.

3.4.3 Implementation de PAD

L'implémentation de PAD est basée sur le mode tangente de la DA, parce que la tangente calcule la variation directionnelle de la sortie en ce qui concerne certaines directions de l'entrée. Ainsi, l'implémentation de la PAD tire profit du calcul de la variation des variables intermédiaires, permettant donc le calcul de chaque contrainte.

Pour un programme arbitraire $P = B_1 \; ; \; T_1 \; ; \ldots ; \; B_n \; ; \; T_n \; ; \; B_{n+1}$, la version différenciée tangente est $P' = B'_1 \; ; \; T_1 \; ; \ldots ; \; B'_n \; ; \; T_n \; ; \; B'_{n+1}$.

Pour mettre en application la PAD, nous insérons une instruction avant chaque essai du programme P'. Cette instruction calcule la valeur β_i pour le test et met à jour la valeur globale β pour le programme.

Pour un programme général$P$, le programme *domaine-validé* $\check{P}$ se définit comme suit :

$$\check{P} = B'_1 \; ; \; V_1 \; ; \; T_1 \; ; \ldots ; \; B'_n \; ; \; V_n \; ; \; T_n \; ; \; B'_{n+1}$$

où V_i est l'instruction qui calcule la valeur de β_i. La valeur de β_i est employée pour mettre à jour la valeur de β. À la fin, nous obtenons la valeur de β qui contient l'information pour le programme entier $\check{P}$. Finalement, l'intervalle de la validité (neighborcapot de "safe") est accumulé dans la valeur de β.

En pratique, l'instruction V_i n'est pas une expression simple, elle est un appel à un sous-programme, qui fait tourner l'algorithme qui met à jour le β global, qui est fondamentalement une intersection d'algorithme d'intervalles.

3.5 Résultats Expérimentaux

Dans cette section, nous montrons comment fonctionne l'exécution de la PAD , et comment s'expriment les résultats . Nous présentons les résultats numériques obtenus par l'application de la PAD à la méthode de newton et nous discutons les résultats.

3.5.1 Expériences avec la méthode de Newton

Le but de cette expérience est de montrer comment l'information de validité peut aider une méthode générale, en particulier quand la méthode inclut des fonctions non-lisses définies par morceaux [7], qui au niveau de l'exécution sont implémentées

comme des instructions conditionnelles.

La méthode de newton est un algorithme itératif pour trouver un maximum/minimum local, la fonction $f()$ doit être deux fois différentiable , et ce que la méthode résout peut être vu comme la recherche de la racine de $f'()$. La méthode pour rechercher un minimum local a la forme suivante :

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{3.15}$$

$$f \in C^2 : [a,b]- > I\!R$$

La méthode peut être étendue pour des dimensions arbitraires en remplaçant $f'()$ par le gradient $\nabla f()$, et la dérivée seconde $f''()$ par la matrice hessienne $Hf()$ de $f()$.

Nous implémentons la formule 3.15, ainsi nous pouvons employer la DA pour calculer $f'()$ et $f''()$ pour l'implémentation d'une fonction arbitraire $f()$. Afin d'effectuer l'expérience, nous utilisons la fonction $f()$ suivante :

Original Function Code	**Second Order Differentiated Code**

`REAL FUNCTION F(x)`
 	
`f = x**3 + x**2 - 3*x - 3`
	`REAL FUNCTION F_D_D(x,xd0,xd,f,f_d)`
	
	`f_d_d = 3*xd*2*x*xd0 + 2*xd*xd0`
	`f_d = 3*x**2*xd + 2*x*xd - 3*xd`
	`f = x**3 + x**2 - 3*x - 3`

Tableau 3.2: Fonction $f()$ et sa version différentielle tangente.

Le tableau 3.2 montre que l'implémentation de la fonction $f()$, et sa dérivé seconde $f''()$, toutes deux nécessaires pour effectuer la méthode itérative. Les dérivées premières et secondes sont implémentées dans la fonction `FUNCTION F_D_D()` du secteur droit du tableau 3.2, cette implémentation est obtenue en appliquant le mode tangente de la DA sur le code qui implémente la fonction $f()$ deux fois.

Sur la figure 3.4 nous pouvons observer les résultats de la méthode sur $f()$. La méthode obtient la bonne solution, et elle nécessite le même nombre d'étapes qu'une version écrite à la main , donc l'exécution de la méthode employant des dérivées de DA est viable et efficace, au moins pour ce genre de fonctions.

Nous implémentons une fonction $f_piecewise()$ définie par morceaux qui montre un comportement semblable comme dernier cas (en bas à droite) de la figure 3.5.

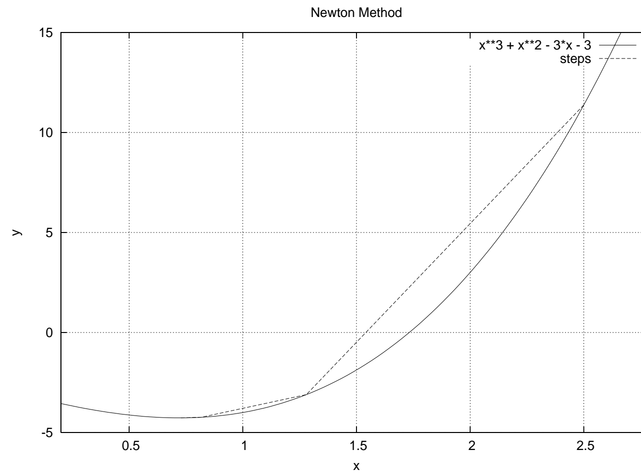

Figure 3.4: Méthode de Newton utilisant des dérivées générées par AD.

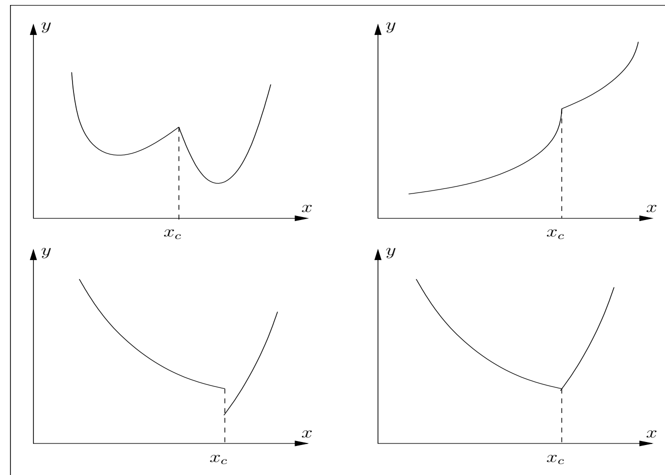

Figure 3.5: Exemples de fonctions définies par morceaux pour la méthode de Newton.

Cette décision est basée sur l'idée que l'information de validité pourrait aider plus qu'à avertir juste de l'inconsistance de certaines dérivées.

Le tableau 3.3 montre l'implémentation de la fonction $f_piecewise()$ et de ses dérivées, les dérivées du premier et du second ordre exigées par la méthode de Newton. La première branche de la fonction $f_piecewise()$ est semblable à la fonction $f()$ de l'expérience précédente. La méthode de Newton est appliquée à la fonction définie par morceaux avec la même installation utilisée pour la fonction $f()$, et nous obtenons les résultats suivants.

Piecewise Function Code	**Code différencié du second ordre**

`REAL FUNCTION F_PIECEWISE(x)`
 	
`IF ( x .GT. 1 ) THEN`
`  f_piecewise = x**3 + x**2`
`    - 3*x - 3`
`ELSE`
`  f_piecewise = x**3 + 2*x**2`
`    - 15*x + 8`
`ENDIF`
	`REAL FUNCTION F_PIECEWISE_D_D`
	` (x,xd0,xd,f_piecewise,f_piecewise_d)`
	
	`IF (x .GT. 1) THEN`
	`  f_piecewise_d_d = 3*xd*2*x*xd0 +`
	`     2*xd*xd0`
	`  f_piecewise_d = 3*x**2*xd +`
	`     2*x*xd - 3*xd`
	`  f_piecewise = x**3 + x**2 -`
	`     3*x - 3`
	`ELSE`
	`  f_piecewise_d_d = 3*xd*2*x*xd0 +`
	`     2*2*xd*xd0`
	`  f_piecewise_d = 3*x**2*xd +`
	`     2*2*x*xd - 15*xd`
	`  f_piecewise = x**3 + 2*x**2 -`
	`     15*x + 8`
	`END IF`

Tableau 3.3: Fonction *f_piecewise*() et sa version dérivée tangente.

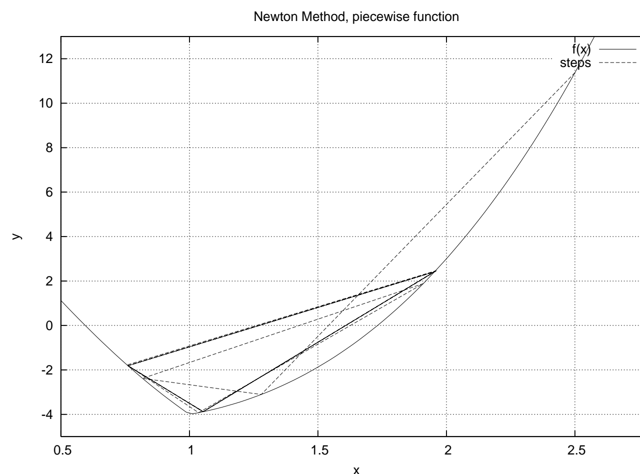

Figure 3.6: Fonctions définies par morceaux.

Sur la figure 3.6 nous pouvons observer que la convergence n'est pas atteinte, et dans le tableau 3.4, qui est tronqué à l'itération 10, nous pouvons l'observer aussi. Ceci parce que le processus itératif tombe dans une boucle. Cette boucle se compose de trois étapes, par exemple, la première boucle va de l'itération 5 à l'itération 7.

Steps	**x**	**y**	**y'**	**y"**
0	2.50000	11.37500	20.75000	17.00000
1	1.27941	-3.10708	4.46951	9.67647
2	0.81752	-2.37972	-9.72493	8.90510
3	1.90958	1.88102	11.75864	13.45748
4	1.03582	-3.92319	2.29038	8.21490
5	0.75701	-1.77519	-10.25278	8.54205
6	1.95728	2.45734	12.40740	13.74368
7	1.05451	-3.87894	2.44499	8.32705
8	0.76089	-1.81492	-10.21958	8.56534
9	1.95402	2.41699	12.36265	13.72413
10	1.05323	-3.88207	2.43430	8.31935

Tableau 3.4: Résultats numériques de la méthode de Newton pour une fonction définie par morceaux.

Nous avons appliqué la méthode PAD à la fonction définie par morceaux afin de calculer l'information de validité le long de la méthode de newton. Nous obtenons les résultats présentés dans le tableau 3.5.

Steps	**x**	**y**	**y'**	**gmin**	**gmax**
0	2.50000	11.37500	20.75000	-1.50000	∞
1	1.27941	-3.10708	4.46951	-0.27941	∞
2	0.81752	-2.37972	-9.72493	$-\infty$	0.18248
3	1.90958	1.88102	11.75864	-0.90958	∞
4	1.03582	-3.92319	2.29038	-0.03582	∞
5	0.75701	-1.77519	-10.25278	$-\infty$	0.24299
6	1.95728	2.45734	12.40740	-0.95728	∞
7	1.05451	-3.87894	2.44499	-0.05451	∞
8	0.76089	-1.81492	-10.21958	$-\infty$	0.23911
9	1.95402	2.41699	12.36265	-0.95402	∞
10	1.05323	-3.88207	2.43430	-0.05323	∞

Tableau 3.5: Résultats numériques de la méthode de Newton et la méthode PAD pour une fonction définie par morceaux.

Cette expérience est spécifique parce que le minimum local est en même temps le point de raccordement entre les deux fonctions composant $f_piecewise()$. Pour cette raison, nous pouvons observe5 dans le Tableau 3.4, sur la digure 3.6 et tableau 3.5 que la convergence n'est pas réalisée, et la méthode réitérera ind'efiniment sans satisfaire

le critère d'arrêt.

Une fois que l'information de validité est calculée pour chaque itération, l'intervalle de la validité avertit de la proximité du test, dans ce cas-ci le minimum local. Par exemple, pour l'étape 1 l'intervalle de la validité est $[-0.27941, \infty]$, qui est dans la direction donnée ($xd >= 1$) et il n'y a pas de problèmes de differentiabilité, d'autre part, le point est très près du test et d'un changement drastique de la valeur de la dérivée première. Comme nous pouvons l'observer dans le tableau 3.5, l'étape 2 est même plus près du minimum local, mais de toute façon le critère d'arrêt n'est pas satisfait.

Nous avons passé en revue le comportement de la méthode de Newton quand elle est appliquée à un certain type de fonctions, et avons tenu compte de l'information utile produite par PAD. D'abord, les résultats ci-dessus prouvent que la PAD informe avec succès de l'information de validité pour cette méthode itérative. En second lieu, l'information de validité peut être utile pour un utilisateur de la méthode de Newton quand la fonction à optimiser n'est pas lisse.

Nous proposons que l'information de validité puisse être employée non seulement pour avertir l'utilisateur de la non-differentiabilité, mais également comme compte-rendu rétroaction pour les algorithmes, particulièrement quand ces algorithmes traitent des fonctions non-lisses. Par exemple, dans le cas de notre exemple la méthode de Newton peut être modifiée, en améliorant le critère d'arrêt ou le calcul de la prochaine étape concernant l'information de validité.

3.6 Conclusions

Nous avons proposé deux méthodes pour aborder le problème de non-differentiabilité dans les programmes différenciés avec la différentiation automatique. Tous les deux reposent sur une analyse dedans de chaque instruction conditionnelle, afin de déterminer quel est le voisinage dans lequel nous pouvons obtenir des dérivés fiables. Ce voisinage sûr est appelé le domaine de validité.

La première méthode que nous avons proposé PADE nous permet de calculer le domaine de la validité pour une entrée donnée avec une précision acceptable. La méthode calcule l'information de validité pour chaque instruction conditionnelle (contrainte) et puis propage l'information, pour ce faire elle emploie le mode inverse de la DA.

Malheureusement, même si PADE nous fournit une description complète du domaine de la validité, le coût de cette méthode est prohibitivement élevé, pour la durée

d'exécution et la consommation de mémoire. Le problème principal pour appliquer cette méthode est la taille de l'ensemble de contraintes.

La deuxième méthode proposée PAD calcule l'information de validité suivant une direction indiquée dans l'espace d'entrée. Ainsi la représentation de l'information de validité devient un intervalle. Si l'entrée demeure dans cet intervalle, les dérivées retournées par la DA n'ont aucun problème de differentiabilité. L'exécution de cette méthode est basée sur le mode tangente de la DA. Le coût informatique de PAD est marginal (4%) en comparaison du coût informatique du mode tangente. Ces petits coûts sont ajoutés au calcul des dérivées quand PAD analyse chaque test dans le code, mais seulement pour certaines directions dans l'espace d'entrée.

Nous avons présenté la méthode PAD, qui s'avère utile, ainsi elle est une manière possible pour avertir l'utilisateur de l'utilisation abusive des dérivées.

Chapitre 4

Analyses de flot et Checkpointing pour le mode renversé de DA

Dans ce chapitre nous étudions des méthodes pour faire face au problème principal du mode inverse de DA. Malgré l'efficacité élevée pour la durée d'exécution de la structure inverse du programme différencié, cette structure est également la source du problème principal du mode inverse de DA. Rappelons succintement la section 2.5.1, la structure différenciée inverse d'un programme se compose de deux parties principales, la première partie s'appelle la progression vers l'avant et la deuxième partie s'appelle le progression vers l'arrière. La progression vers l'avant a fondamentalement les mêmes instructions que le programme original. Le but de la progression vers l'avant est de calculer des valeurs intermédiaires requises par la progression vers l'arrière. La progression vers l'arrière implémente les dérivées. Pendant l'exécution de la progression vers l'avant les variables des valeurs d'intermédiaires sont redéfinies, et de ce fait modifiées. Le problème survient quand les valeurs d'intermédiaires sont requises par les dérivées, mais elles ne sont plus accessibles parce qu'elles ont été écrasées.

Dans cette thèse, nous nous concentrons sur une stratégie pour faire face au problème ci-dessus. Cette stratégie s'appelle Stocker-Tout (ST). La stratégie ST consiste à stocker les valeurs intermédiaires de variables dans la progression vers l'avant, puis de les restaurer pour la progression arrière, rendant de ce fait les valeurs intermédiaires accessibles aux instructions qui implémentent les dérivées. Le problème avec cette approche est la consommation de mémoire élevée possiblement inacceptable [5]. Ceci motive notre recherche. Nous avons deux types d'optimisation à explorer afin de résoudre le problème de mémoire. Le premier type d'optimisation améliore le code différencié au niveau des instructions, et est basé sur des analyses de flot de données. Le deuxième type d'optimisation travaille sur les segments du code de taille arbitraire.

Ce chapitre est organisé comme suit. Dans la section 4.1 nous présentons le problème principal du mode inverse de DA. Dans la section 4.2 nous présentons les

stratégies classiques pour faire face au problème mentionné. Dans la section 4.3 nous présentons un modèle formel initial pour la différentiation inverse, qui tient compte de la stratégie ST. Dans la section 4.4 nous présentons nos contributions à l'optimisation au niveau des instructions . Dans la section 4.5 nous présentons nos contributions à l'optimisation sur les segments de code. Enfin dans la section 4.6 nous concluons et discutons les travaux futurs dans les directions des contributions proposées.

4.1 Le problème de consommation de mémoire du mode inverse

Le mode inverse calcule des codes adjoint, en particulier de gradients. En raison de la structure du mode renverse de la DA, le calcul de ce type de code est très efficace en termes de durée d'exécution, mais le coût est l'exigence d'une grande quantité de valeurs de variables intermédiaires, qui peuvent être perdue par une redéfinition de la variable pendant la progression vers l'avant. Par conséquent, le mode inverse présente un inconvénient. Ceci parce que des variables intermédiaires sont redéfinies pendant la progression vers l'avant, ainsi les valeurs de ces variables intermédiaires tenues ne sont pas accessibles dans la progression vers l'arrière, alors que les dérivées peuvent les exiger. Par exemple, considèrons l'exemple suivant :

$$P = I_1 \ ; \ I_2 = x \times y \ ; \ I_3 \tag{4.1}$$

$$\overline{P} = I_1 \ ; \ I_2 = x \times y \ ; \ I_3 \ ; \ I_3' \ ; \ xb = y \times I_2 b \ ; \ yb = x \times I_2 b \ ; \ I_1' \tag{4.2}$$

où le programme $\overline{p}$ est la version différenciée inverse du programme P, et les instructions I_i' implémentent la dérivée de l'instruction correspondante I_i.

Dans la formule 4.2, la dérivée inversée de l'instruction I_2 se décompose en deux limites, $I_2' = (xb = y \times I_2 b \ ; \ yb = x \times I_2 b)$. Si nous supposons que les variables x et/ou y sont redéfinies dans I_3, alors les valeurs de ces variables au moment du calcul de I_2 sont hors de portée quand elles sont exigées par l'instruction I_2'.

En fait, les programmes incluent plusieurs redéfinitions des variables, il suffit juste d'imaginer le cas des variables dans des boucles, ainsi le scénario présenté par la formule 4.2 est usuel dans les programmes scientifiques et industriels. Afin d'éviter ce problème, nous pouvons demander aux utilisateurs de modifier leurs codes, mais ce n'est pas réaliste, ainsi les modèles AD doivent surmonter ce problème d'une manière systématique. Dans la prochaine section, nous explorons les principales solutions de rechange pour faire face au problème mentionné.

4.1.1 Stratégie Stocker-Tout (ST)

La stratégie ST consiste à stocker dans une structure de mémoire d'empilement spécial (*bande enregistreuse*) toutes les valeurs intermédiaires qui seront exigées par le sweep arrière et ensuite à restaurer dans le sweep arrière les valeurs de la bande dans l'ordre inverse de celui de l'ordre du stockage. Ceci a pour résultat une structure inversée des programmes différenciés comme illustré sur la partie gauche de la figure 4.1 (page suivante).

Rule	Program	Reverse Differentiated Program
R_0	$P = ()$	$\overline{P} = ()$
R_1	$P = I$	$\overline{P} = I'$
R_n	$P = I \ ; \ D$	$\overline{P} = \bullet_1 \ ; \ I_1 \ ; \ \overline{D} \ ; \ \circ_1 \ ; \ I'$

Tableau 4.1: les règles de la ST

Le tableau 4.1 présente l'ensemble minimum de formules récursives requises pour établir un programme différencié inverse de type ST, où le D représente la séquence des instructions après certaine instruction jusqu'à la fin du programme, le symbole $\bullet_i$ décrit le stockage des valeurs de variables à écraser par l'instruction I_i, et le symbole $\circ_i$ décrit la restauration des valeurs stockées précédemment.

$$\overline{P_{sa}} = \bullet_1 \ ; \ I_1 \ ; \ \bullet_2 \ ; \ I_2 \ ; \ I_3' \ ; \ \circ_2 \ ; \ I_2' \ ; \ \circ_1 \ ; \ I_1' \tag{4.3}$$

La formule 4.3 est la version différenciée inverse du programme pris en exemple (de formule 4.1), il implémente la stratégie de ST. Par conséquent, avant que les variables soient écrasées leurs valeurs sont stockées, et puis restituées dans le sweep arrière avant l'instruction qui les exige. Par exemple, la dernière instruction du programme $\overline{p_{sa}}$ exige les valeurs qui peuvent être écrasées par I_1, ainsi avant I_1 les valeurs sont stockées dans $\bullet_1$, puis restitué à partir de $\circ_1$ avant I_1'; la procédure est également mis implémentée pour I_2'. Ce n'est pas le cas de I_3' parce que les définitions dans I_3 sont inutiles pour I_3', de ce fait I_3 n'est pas nécessaire du tout, donc il n'y a aucun besoin de stocker des valeurs avant I_3.

La stratégie Stocker-Tout varie linéairement pour la consommation de mémoire et la durée déxécution. Le plus mauvais scénario se produit quand il est nécessaire de stocker un grand nombre des valeurs, la longueur de la bande peut prendre une taille inacceptable.

L'inconvénient de la ST est la consommation élevée de mémoire, comme on peut l'observer sur la figure 4.1, le pic de consommation de mémoire se produit à la fin du sweep avant.

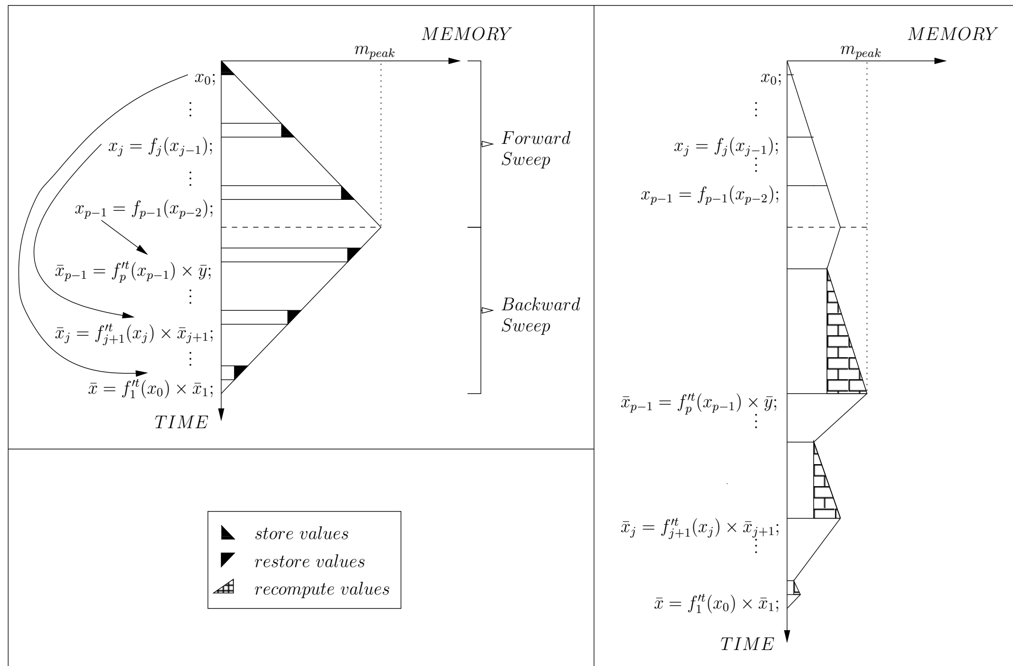

Figure 4.1: L'axe horizontal représente la quantité de valeurs stockée à un moment précis.

4.1.2 Stratégie Stocker-Tout et contrainte de mémoire

Nous décidons d'implémenter et d'étudier la stratégie de ST en détail. Nous croyons que c'est une stratégie très prometteuse pour la durée d'exécution. D'autre part, la consommation de mémoire peut être améliorée et ce sera notre objectif.

Nous croyons que la contrainte de mémoire est cruciale. Par exemple dans l'industrie du matériel informatique, la technologie pour le CPU progresse plus rapidement que la technologie pour la RAM. La première est limitée par la loi du Moore [16] des circuits intégrés (le nombre de transistors doublant tous les 18 mois) et la dernière n'augmente que de 10%. Ainsi, l'espace mémoire est une contrainte importante.

Dans la prochaine section, nous présentons les stratégies classiques au sein de la stratégie ST, qui est le point de départ de notre recherche.

4.2 Stratégies classiques pour l'approche Stocker-Tout

Pour contrôler le problème de mémoire causé par le stockage de valeurs intermédiaires, la stratégie stocker-tout peut être améliorée dans deux directions principales. Premièrement, un rafinement des analyses de flot de données afin de réduire le nombre de valeurs à stocker et de générer les instructions . Ces analyses de flot de données sont des améliorations au niveau des instructions, c'est pourquoi nous les avons appelées des stratégies échelle fine. Deuxièment, la désactivation de la stratégie stocker-tout pour des segments choisis du code, nous permettant d'épargner de l'espace mémoire. Puisque cette stratégie concernent des segments du code, nous l'avons appelé stratégie échelle macro.

Dans la section suivante, nous présentons les stratégies échelle fine classiques employées pour faire face au problème de mémoire du mode inverse.

4.2.1 Stratégies échelle fine

Analyses de flot de données

Les analyses de flot de données sont statiques, parce qu'elles fonctionnent pendant le temps de compilation et sans information d'exécution. Ces analyses sont conservatives en termes de résultats, évitant de ce fait des cas où le résultat de l'analyse est incertain. Un autre risque de l'analyse de flot de données est le problème d'explosion combinatoire. Pour le maîtriser, les analyses de flot de données sont habituellement conçues en tant que modèle hiérarchique. Pour ce genre de modèle deux sweep dans le code sont exécutés, le premier sweep est ascendant et calcule l'information synthétisée localement qui est récursivement combinée niveau par niveau, ainsi cette information est indépendante du reste du programme (contexte libre). Réciproquement, le deuxième sweep est descendant et dépend du contexte, ainsi il propage l'information synthétisée par le programme.

Les caractéristiques ci-dessus et les dispositifs désirés forment le cadre pour tout le flot de données présenté dans cette thèse.

L'analyse traditionnelle du flot de données se concentre sur les codes génériques, des prolongements à l'analyse de flot de données ont été présentés dans [4, 18, 10], où les notions "analyse d'activité" et "être enregistré" (TBR) ont été définies :

- Analyse d'activité [10]:
  Toutes les variables n'ont pas une influence sur la différentiabilité, celle qui ont

cette influence s'appelle les variables actives. Afin de déterminer l'ensemble de variables actives dans un morceau de code nous devons employer le rapport de dépendance, ainsi une variable est en activité si à la fois elle dépend d'une variable indépendante, et qu'en même temps une variable dépendente dépend d'elle.

- TBR [4, 18, 10]:
  L'idée est de déterminer quelles valeurs de variables doivent être stockées pendant le sweep avant parce qu'elles sont exigées par le sweep arrière. Une valeur de variable est stockée sur la bande seulement si elle est exigée dans le sweep arrière et écrasée dans le sweep arrière.

  TBR se compose de deux étapes. La première étape est une analyse ascendante qui a synthétisé deux ensembles de variables, les variables utilisées dans le code adjoint et les variables écrasées par le sweep arrière. La deuxième étape est une analyse descendante utilisant les deux ensembles précédents de variables, elle calcule l'ensemble de variables qui sont exigées dans le code adjoint, ainsi quand une variable exigée est écrasée, elle est marquée comme "être enregistré ". Par conséquent, quand le code est produit, un certain nombre de sous-programmes de gestion d'empilement les `PUSH/POP` est inséré, le `PUSH` est inséré juste avant l'instruction qui écrase la variable, le `POP` est inséré juste après l'instruction qui implémente la dérivée de l'instruction qui écrase la variable.

Recalcul versus stockage

Quelques valeurs exigées par le sweep arrière peuvent recalculées juste en exécutant de nouveau une ou deux instructions originales, dans ce cas, stocker la valeur dans le sweep avant ou la re-calculer dans le sweep arrière sont toutes deux des options valides à suivre. Mais tous les deux exigent une analyse précise afin d'exposer la petite différence.

4.2.2 Stratégies échelle macro

Checkpointing

Le mécanisme qui désactive la stratégie stocker-tout pour certains segments choisis, s'appelle *checkpointing.* Le checkpointing exploite une réciprocité entre le stockage et le re-calcul. Il a deux conséquences sur le comportement du programme différencié inversé :

1. quand le sweep arrière atteint un segment choisi (appelé *checkpointed segment*), un ensemble suffisant de valeurs (appelées *snapshot*) doit être stocké. Le snapshot permet d'exécuter à nouveau le segment concerné par le *checkpointing*

dans le sweep arrière avec le contexte correct. Pendant l'exécution du sweep avant pour un segment concerné par le *checkpointing* la stratégie stocker-tout est désactivée.

2. quand le snapshot est restitué pendant le sweep arrière, le segment concerné par le *checkpointing* est exécuté de nouveau, mais cette fois la stratégie stocker-tout est activée. Par conséquent, le reste du sweep arrière est exécuté comme d'habitude.

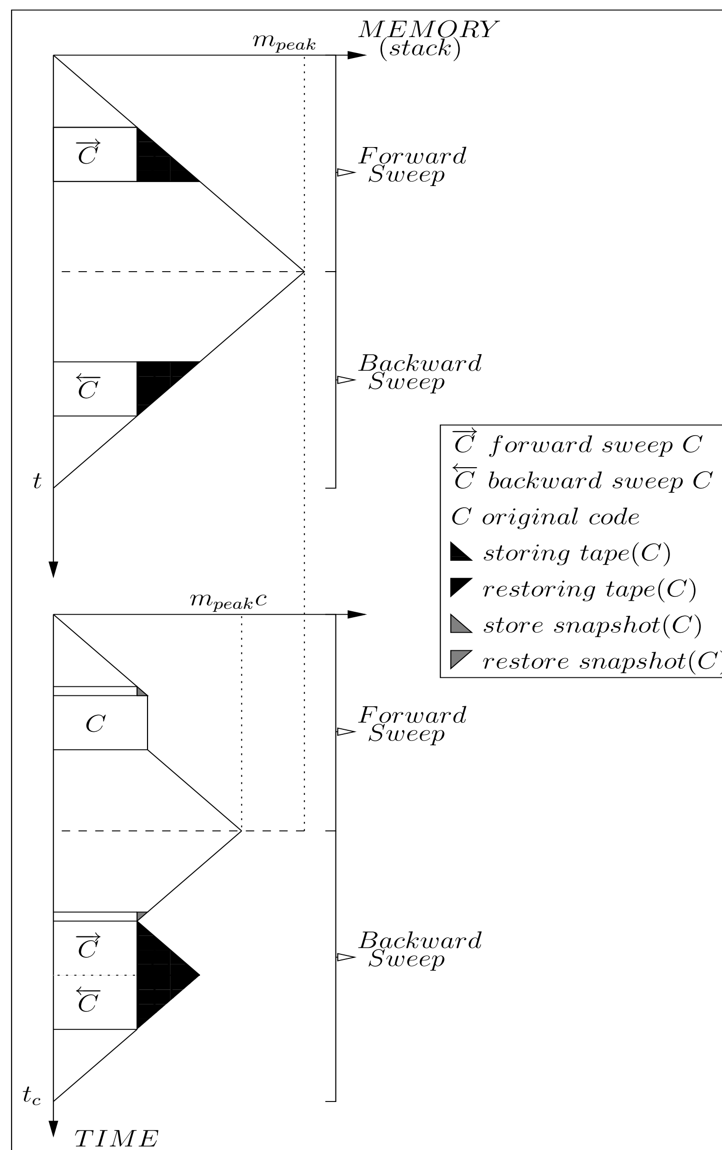

Figure 4.2: Checkpointing sur le mode inverse DA.

L'ensemble des valeurs des variables stockées pour un segment pendant le sweep avant s'appelle *tape*. Le mécanisme checkpointing est profitable si la taille du snapshot est plus petite que la taille de la bande pour n'importe quel segment concerné par le *checkpointing* . Sous cette hypothèse, ce mécanisme est profitable pour l'espace mémoire. Ceci parce que le stockage du snapshot augmente à peine la taille de l'empilement, ainsi le pic de consommation de mémoire du programme différencié inversé avec checkpointing est ($tape - snapshot$) plus petit qu'un programme différencié inversé sans checkpointing. En revanche, le mécanisme n'est pas profitable pour la durée d'exécution, en raison d'une double exécution du segment concerné par le checkpointing.

Afin de définir la stratégie checkpointing deux éléments doivent être spécifiés. Le premier élément est le choix du placement de points de checkpointing. Le deuxième élément est la définition des valeurs de variables appartenant à un snapshot.

Hypothèse concernant la bande et le snapshot

Comme nous l'avons mentionné ci-dessus, nous supposons que la bande est plus grande que le snasphot. L'hypothèse est raisonnable parce que dans la plupart des cas le segment concerné par le checkpointing est aussi grand que la taille de sa bande. Ceci parce que la taille de la bande s'accroît linéairement avec le nombre d'instructions du segment concerné par le checkpointing. D'autre part, le snapshot présente un taux de croissance logarithmique. Par exemple, l'ensemble de valeurs d'entrée pour un grand sous-programme peut être petit. En conclusion, les observations expérimentales confirment l'hypothèse, ces observations font partie des expériences de la fin du chapitre.

Par exemple sur la figure 4.2, nous supposons que $tape(C >)Snp(C)$. Par conséquent, nous voyons que $m_{peak}c$ est plus petit que m_{peak}, parce que dans le cas utilisant le checkpointing, $tape(C)$ n'est pas exigé par le premier sweep arrière du segment C. Inversement, nous voyons que t_c est plus grand que t, parce que dans le cas sans checkpointing les sous-programmes sont exécutés seulement une fois, et comme nous pouvons l'observer dans le cas avec checkpointing, le sous-programme C est exécuté deux fois (C et $\overrightarrow{c}$).

Placements des points de Checkpointing

Les segments concernés par le checkpointing peuvent être placés de manières arbitraires, ou par un arrangement systématique. Dans la littérature, la stratégie de Griewank [6] est bien connue et est optimale mais seulement pour un cas particulier. Dans le cas général le comportement d'exécution du programme est inconnu, par exemple le nombre d'étapes exigées pour accomplir la tâche. Pour ce cas général, la stratégie checkpointing optimale n'a pas encore été trouvée.

Maintenant, nous présentons les stratégies checkpointing classiques :

1. Stratégie optimale de Checkpointing pour un nombre fixe d'étapes :

   Cette stratégie considère la taille des snapshots, ainsi elle se focalise sur l'optimisation de l'endroit où les points de checkpointing doivent être placés, et leur nombre. Afin d'accomplir cela, la stratégie utilise des fonctions récursives qui choisissent quelles étapes de calcul conviennent pour être traîtées par le checkpointing. Cette stratégie est également connue sous le nom de *partition binomiale*. Ceci parce qu'une étape critique de la stratégie est de dédoubler le rang des étapes, permettant de ce fait la récursion sur les rangs secondaires.

   Cette stratégie a été implémentée [9], mais en simplifiant les algorithmes. Ceci parce que la version originale était très chère au niveau informatique (inclut trop de récursions), et très particulière en ce qui concerne le type de problèmes qui peuvent être traîtés.

   Une extension prometteuse de cette stratégie a été présentée par Sternberg [20]. Dans ce travail, le nombre d'étapes est donné mais la taille des snapshots change selon les endroits où les points de checkpointing sont placés. La stratégie emploie deux genres de snapshots, l'un appelé *mince*, l'autre appelé *gras*, les deux genres de snapshots sont reliés, ainsi la taille du snapshot *gras* est trois fois la taille des snapshots *mince*. En outre, ils présentent l'idée de l'emboîtement de checkpointing dans leur cadre d'application. Des heuristiques sont présentés pour améliorer l'exécution pour certains cas. Par conséquence, ils ont réalisé l'exécution suboptimale.

2. Checkpointing sur l'appels de sous-programmes : Une stratégie plus simple et facile à implémenter, bien que non optimale, est de placer systématiquement les points de checkpointing avant chaque appel de sous-programme. Cette stratégie est employée dans l' outil de DA TAPENADE . Il est possible que quelques points de checkpointing ne soient pas nécessaires ou peu utiles, ceci est l'une de raisons pour laquelle la stratégie n'est pas optimale, l'autre raison est que parfois l'hypothèse derrière la stratégie checkpointing n'est pas valide, comme nous le montrons dans les résultats expérimentaux.

   Dans cette stratégie, la taille du snapshot est définie par une équation qui se fonde sur l'analyse de flot de données, ainsi la taille du snapshot est différente pour chaque appel de sous-programme . Ceci parce que cela dépend du contenu du sous-programme concerné par le checkpointoing. En raison de la précision

de l'analyse de flot de données, en général les snapshot sont de petits ensembles de variables.

Définition du Snapshot

Le snapshot est l'ensemble minimal de variables exigées pour permettre au segment concerné par le checkpointing d'être recalculé, produisant de ce fait les valeurs qui sont exigées dans le sweep arrière. Afin de déterminer ce qu'est un snapshot, nous présentons la définition conservative suivante.

Pour un programme arbitraire $P = S \; ; \; D$, où S est un sous-programme, et les points concernés par le checkpointing sont placés avant les appels de sous-programme. La version différenciée du programme P s'écrit comme suit,

$$\overline{P} = \mathbf{PUSH(Snp(}S, D\mathbf{))} \; ; \; S \; ; \; \overline{D} \; ; \; \mathbf{POP(Snp(}S, D\mathbf{))} \; ; \; \overline{S} \tag{4.4}$$

La définition de la formule du snapshot utilisée dans la formule 4.4 est:

$$\mathbf{Snp}(S, D) = \mathbf{Use}(\bar{S}) \cap (\mathbf{Out}(S) \cup \mathbf{Out}(\overline{D})) \tag{4.5}$$

Dans la formule 4.5, l'ensemble $\mathbf{Use}(\overline{S})$ est la limite supérieure de la quantité d'information que nous devons stocker, parce que cet ensemble contient les valeurs nécessaires pour le sweep arrière S. Mais cet ensemble peut inclure trop de valeurs de variables pas vraiment nécessaires. Par conséquent, nous pouvons raffiner la formule du snasphot en détectant les variables qui sont écrasées dans S ($\mathbf{Out}(S)$), détectant également les variables écrasées en aval de $\overline{P}$ ($\mathbf{Out}(\overline{D})$).

4.3 Un modèle formel du mode inverse Stocker-Tout de DA

Pour un programme donné $P = I \; ; \; D$, où I est une instruction, et D représente la séquence des instructions appelées *descendante*, où cette séquence d'instructions va d'après l'instruction I à la fin du programme. Le programme différencié inverse $\overline{P}$ a la forme suivante :

$$\overline{P} = \overline{I; D} = \overrightarrow{I}; \overline{D}; \overleftarrow{I} = \mathbf{PUSH}(\mathbf{Out}(I)) \; ; \; I \; ; \; \overline{D} \; ; \; \mathbf{POP}(\mathbf{Out}(I)) \; ; \; I' \tag{4.6}$$

La structure du programme $\overline{P}$ se compose de deux parties, le sweep avant :

$$\overrightarrow{P} = \overrightarrow{I; D} = \mathbf{PUSH}(\mathbf{Out}(I)) \; ; \; I \; ; \overrightarrow{D}$$

et le sweep arrière :

$$\overleftarrow{P} = \overleftarrow{D; I} = \overleftarrow{D} \; ; \; \mathbf{POP}(\mathbf{Out}(I)) \; ; \; I'$$

Le modèle considère que si une variable est écrasée par I, la valeur de la variable avant d'être écrasée doit être emmagasinée dans l'empilement. Cet ensemble de variables est fourni par l'analyse de **Out**. L'action est effectuée par PUSH. Par conséquent, la valeur peut être restorée par POP dans le sweep arrière avant qu'elle soit employée par I'.

Le modèle peut être amélioré. Dans la prochaine section, nous présentons les premières types d'améliorations. Elles concernent l'analyse améliorée du flot de données.

4.4 Contributions aux stratégies échelle fine

Le modèle décrit dans la formule 4.6 peut être amélioré. La première amélioration est liée au fait que les résultats de l'instruction I dans la formule 4.6 sont seulement utiles pour $\overline{D}$. Ceci parce que si nous avons des instructions avant I, appelée *ascendante* U, le sweep arrière de ces instructions $\overleftarrow{U;I}$, requiert seulement les valeurs intermédiaires de variables créées avant I. Par conséquent, si les résultats de I ne sont pas utiles dans $\overline{D}$, nous pouvons éviter l'instruction I et le PUSH/POP associé. Afin de détecter ce comportement nous présentons le prédicteur *Adj-live*$(I, D) = (\mathbf{Out}(I) \cap \mathbf{live}(\overline{D}) \neq \emptyset)$, donc si le prédicteur est faux nous pouvons enlever l'instruction I. Pour calculer le prédicteur, nous devons d'abord calculer l'analyse **Out**, ce qui a déjà été présenté, et $\mathbf{Live}(\overline{D})$ ce qui est nouveau. Cette nouvelle analyse s'appelle *Adjoint Liveness.*

La deuxième amélioration est liée à l'analyse de TBR. L'idée est de raffiner l'analyse de TBR en utilisant l'information du contexte des instructions différenciées suivantes, ce qui inclut le sweep arrière de U, ainsi nous devons présenter U comme un contexte dans la formule (4.6). Nous employons la notation $\vdash$ pour séparer U de la partie du programme actuellement différenciée. Nous présentons l'ensemble des variables employées par les instructions I' et après, qui est $\mathbf{Use}(I'; \overleftarrow{U})$. Tenant compte de ceci, les seules variables réellement PUSH'ed et POP'ed pour l'instruction I sont maintenant l'ensemble $(\mathbf{Out}(I) \cap \mathbf{Use}(I'; \overleftarrow{U}))$.

La dernière amélioration est liée à l'analyse d'activité. Si l'analyse d'activité était effectuée avant les deux améliorations ci-dessus, les analyses ci-dessus calculeraient de plus petits ensembles de variables intermédiaires. Ceci parce que l'analyse d'activité peut supprimer beaucoup des instructions. Par exemple, il peut s'avérer que quelques variables ont toujours une dérivée nulle en ce qui concerne les entrées indépendantes ou les sorties dépendantes. Quand la variable écrite par la tâche I est inactive, alors I' peut être enlevé. Quand une certaine variable employée par la tâche I est inactive, I' est simplifié.

4.4.1 Modèle amélioré du mode Stocker-Tout inverse de la DA

En raison du nouvel adjoint amélioré et des analyses du flot de données, les modèles formels prennent la forme suivante, un modèle plus précis et plus complexe [12] :

$$U \vdash \overline{I;D} \;=\; \begin{array}{l} [\text{PUSH}(\mathbf{Out}(I) \cap \mathbf{Use}(I';\overleftarrow{U}));I;]\ \textit{if adj-live}(I,D) \\ [U;I] \vdash \overline{D}; \\ [\text{POP}(\mathbf{Out}(I) \cap \mathbf{Use}(I';\overleftarrow{U}));]\ \textit{if adj-live}(I,D) \\ I' \end{array} \tag{4.7}$$

Selon le prédicteur *adj-live*(I,D), on peut également éliminer l'instruction originale I, plus tard le PUSH/POP peut aussi être éliminé, quelque soit le résultat de l'analyse de $\mathbf{Use}(I';\overleftarrow{U})$. C'est un gain important de mémoire, parce que l'empilement n'est pas employé, il représente également un gain dans la durée d'exécution. Ceci parce que le temps d'accès à l'empilement n'est pas vraiment négligeable. Si le prédicteur est vrai, alors le gain dépend de la taille de l'ensemble de valeurs à stocker. Ce gain est la différence entre $(\mathbf{Out}(I) \cap \mathbf{Use}(I';\overleftarrow{U}))$ et $\mathbf{Out}(I)$, ce qui était l'ensemble des variables à stocker dans la formule 4.6.

4.5 Contributions aux stratégies échelle macro

4.5.1 Un modèle formel du mode inverse Stocker-Tout de la DA avec Checkpointing

Nous nous intéressons à l'étude de l'organigramme des codes différenciés inverse. Ceci parce que l'organigramme est la manière la plus pratique pour analyser un programme avec des segments de code concernés par le checkpointing, spécifiquement quand les points de checkpointing sont placés avant les appels de sous-programme. Nous devons prolonger les formules des sections précédentes afin de mesurer l'influence de ces analyses dans l'organigramme.

Maintenant considérons le cas où le programme P contient le segment concerné par le checkpointing. Ainsi, pour un programme arbitraire $P = S\ ;\ D$, où le segment concerné par le checkpointing est un sous-programme. Habituellement, le prédicteur *adj-live*(S,D) est vrai et simplifie de ce fait la notation en perdant aucune généralité. Par conséquent, le modèle inverse de la DA devient :

$$
\begin{aligned}
U \vdash \overline{S;D} \;=\; & \text{PUSH}(\mathbf{Out}(S) \cap \mathbf{Use}(\overleftarrow{U})); \\
& \text{PUSH}(\mathbf{Snp}(U,S,D)); \\
& S; \\
& [U;S] \vdash \overline{D}; \\
& \text{POP}(\mathbf{Snp}(U,S,D)); \\
& [\,] \vdash \overline{S}; \\
& \text{POP}(\mathbf{Out}(S) \cap \mathbf{Use}(\overleftarrow{U}));
\end{aligned} \tag{4.8}
$$

Une différence se présente entre l'ensemble calculé par $\mathbf{Snp}(U,S,D)$ et l'ensemble $\mathbf{Out}(S) \cap \mathbf{Use}(\overleftarrow{U})$, et est fondamentalement due au contexte ajouté par U. Par exemple, mettre U au lieu de $[\,]$ comme contexte pour la génération de $\overline{C}$ coûtera plus de PUSH/POP à l'intérieur de $\overline{C}$, et d'autre part, en stockant $\mathbf{Out}(C) \cap \mathbf{Use}(\overleftarrow{U})$ devient inutile dans 4.8. L'exploration de cette différence est un problème non résolu [3].

Le modèle de la DA avec checkpointing défini dans la formule 4.8 est plus efficace que le modèle de la formule 4.4. Ceci parce qu'il tire profit des analyses de flot de données d'adjoint qui permettent de définir une meilleure formule de snapshot, comme nous le présentons dans la prochaine section.

4.5.2 La Définition Améliorée du Snapshot

En comparaison de la formule 4.5, nous améliorons principalement cette formule en ajoutant l'information du contexte de U qui est employé par les analyses de flot de données d'adjoint. Par exemple, l'ensemble de variables $\mathbf{Live}(\overline{C})$, requis pour faire tourner $\overline{c}$, est plus petit que $\mathbf{Use}(C)$. En second lieu, nous devons reconstituer une variable seulement si elle était modifiée " dans l'intervalle, "c.-à-d." qu'elle est dans **Out** l'ensemble de la séquence du code $C;\overline{D}$. Nous tirerons profit du fait que $\mathbf{Out}(\overline{D})$ est plus petit que $\mathbf{Out}(D)$. Par conséquent, quand le prédicteur *adj-live(*S,D*)* est vrai, nous définissons le snapshot comme :

$$
\mathbf{Snp}(U,S,D) = \mathbf{Live}(\overline{S}) \cap (\mathbf{Out}(S) \cup \mathbf{Out}([U;S] \vdash \overline{D})) \tag{4.9}
$$

L'ensemble des variables à stocker par la définition améliorée du snasphot est plus petit que celui calculé par la formule 4.5. Ceci parce que la définition du snapshot de la formule 4.9 emploie l'analyse *liveness* d'adjoint, qui calcule un plus petit ensemble de valeurs de variables que l'ensemble utilisé dans la formule 4.5. En outre, dans la formule 4.9 l'ensemble **Out** est calculé en tenant compte du contexte $[U;S]$. Ceci implique la possibilité de réduire l'ensemble calculé.

4.5.3 Le Checkpointing Systématique

Des segments concernés par le checkpointing peuvent être choisis de diverses façons. Dans les outils de DA, le checkpointing est appliqué systématiquement, par exemple à chaque appel de sous-programme ou autour des corps des boucles.

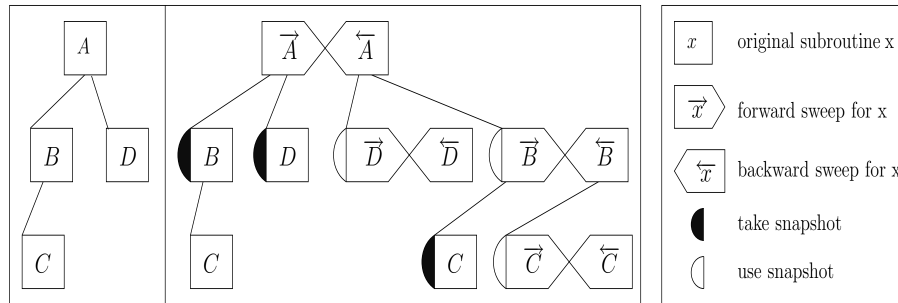

Figure 4.3: Checkpointing pour tous les appels dans le mode inverse de DA (mode joindre-tout).

La figure de la section centrale de 4.3, montre l'organigramme du programme différencié inverse en utilisant le mode joindre-tout, où joindre-tout est la stratégie checkpointing appliquée à chaque appel de sous-programme. L'expérience indique que la stratégie joindre-tout n'est pas optimale, bien qu'il ne soit pas facile de prévoir la situation optimale.

Il est possible de désactiver le mécanisme checkpointing pour certain segment de code, ceci s'appelle le mode split. En mode split, le sweep avant et le sweep arrière sont implémentés séparément, et ne se suivent pas pendant l'exécution, ainsi aucun snapshot n'est exigé, mais ceci impose de stocker plus de valeurs intermédiaires sur la bande. Ceci parce que les valeurs locales de variables du sweep avant du segment sans aucun checkpointing sont inaccessibles pour le sweep arrière correspondant. Par conséquent, elles doivent être stockées. La figure 4.4 montre l'autre alternative classique, qui est aucun point de checkpointing dans chacun des sous-programmes, cette alternative s'appelle la stratégie *Dédoubler-tout.*

L'avantage du mode split est que les sous-programmes sont exécutés juste une fois (figure 4.4), alors l'épargne sur la durée d'exécution est importante.

Dédoubler-tout et joindre-tout sont deux stratégies extrêmes. Cela vaut la peine d'essayer des cas hybrides, nous présentons un couplage des cas dans la figure 4.5. La première stratégie hybrid1, implémente le mode joindre pour tous les sous-programmes excepté le sous-programme D.

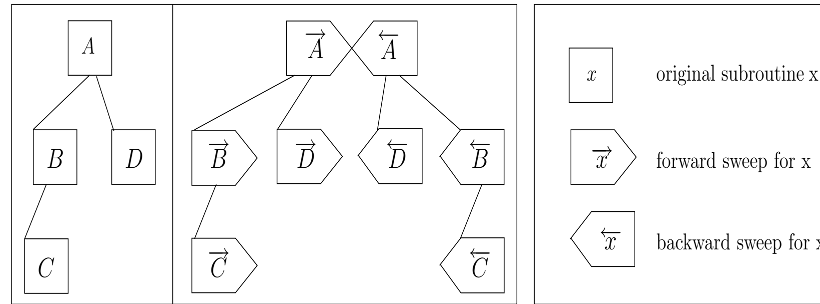

Figure 4.4: Pas Checkpointing dans le mode inverse DA (mode Dédoubler-tout).

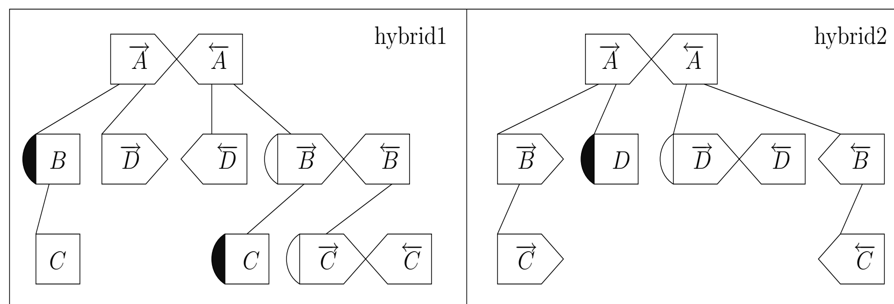

Figure 4.5: Approche hybride (split-joindre)

Réciproquement, la deuxième stratégie : hybrid2, implémente le mode split pour tous les sous-programmes excepté le sous-programme D, qui est traité par checkpointing.

Simulation de stratégies hybrides supposant que $Snapshot < tape$

Afin d'avoir une idée plus précise de la différence mentionnée ci-dessus, nous simulerons les performances des cas mentionnés pour deux scénarios de motivation. Nous supposons que tous les sous-programmes ont la même taille de snapshot, qu' ils ont également la même taille de bande, mais le snapshot et la bande ont des tailles différentes. En outre, nous supposons que chaque sous-programme a le même temps d'exécution. L'organigramme original est celui donné sur la figure gauche **??** et le différencié est donné sur les figures **??**, 4.4, 4.5.

Pour le premier scénario, nous avons placé la capacité de la mémoire du snapshot à 6 et la capacité de la mémoire de la bande à 10. Ces données correspondent à l'hypothèse habituelle que la bande est plus grande que le snapshot pour des sous-programmes.

La figure 4.6 montre le comportement des quatre stratégies de checkpointing

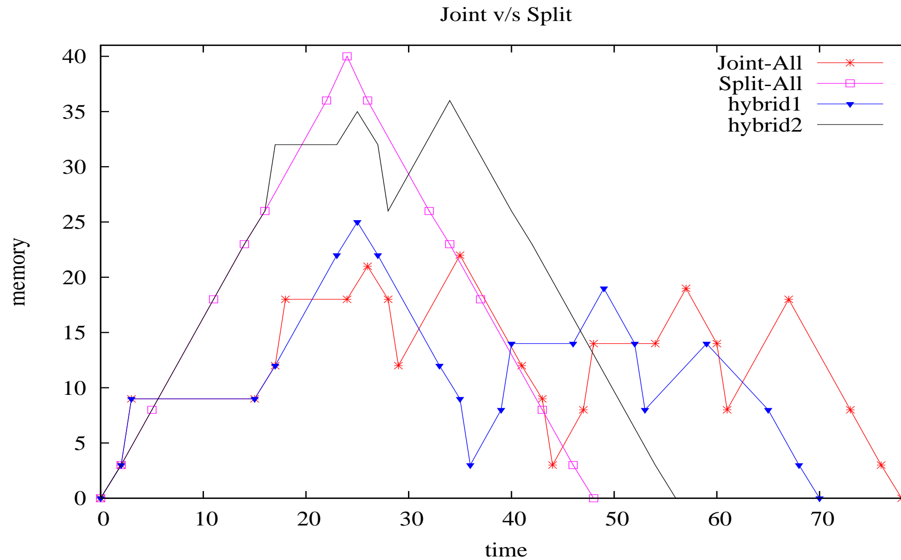

Figure 4.6: Simulation des résultats, tape = 10, snapshot = 6.

mentionées précédemment. Comme nous l'avons prévu, la courbe qui représente la configuration joindre (joint-all dans la légende) montre la plus petite consommation de mémoire mais la plus grande durée d'exécution. Réciproquement, la courbe qui représente le mode split (split-all dans la légende) présente le pic le plus élevé de consommation de mémoire mais le temps d'exécution le plus court. Les stratégies hybrides varient entre ces deux extrémités.

Simulation de stratégies hybrides supposant que $Snapshot > tape$

Le scénario précédent a supposé que la bande est plus grande que le snapshot. Cependant, cette hypothèse n'est pas toujours valide. Par conséquent, nous présentons une deuxième simulation où nous supposons que la bande a coûté 6 dans la mémoire, et la bande coûte en réalité plus, par exemple 10.

La figure 4.7 prouve que les modes joindre et split ne sont plus aux extrémités. En fait, les limites extrêmes dans la consommation de mémoire correspondent aux modes hybrides. Un autre fait intéressant de la deuxième simulation est que la crête maximum de la consommation de mémoire est plus petite que celle de la première simulation, ce qui n'est pas étonnant étant donné que l'instantané est plus grand que la bande mais bien moins utilisé. Dans ce scénario, l'avantage de checkpointing est moins évident en raison des coûts d'instantanés, donc le mode dédoubler-tout est presque le meilleur en tout point.

Nous avons montré dans la figure 4.6 et la figure 4.7 que les stratégies hybrides sont

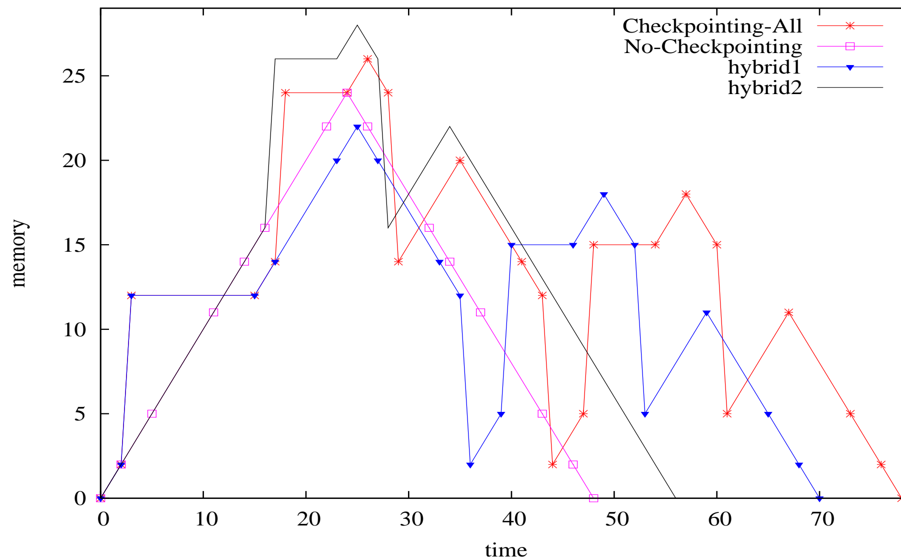

Figure 4.7: Résultats numériques génériques, tape = 6, snapshot = 10.

des solutions de rechange attrayantes par rapport aux stratégies de base. Il serait très utile d'effectuer quelques expériences avec les stratégies de base et hybrides. Ceci nous donnera une idée des meilleures stratégies concernant les codes de taille industrielle. Nous considérons cette expérimentation comme la première étape pour une stratégie checkpointing optimale.

4.5.4 Observation expérimentale des problèmes et des résultats

Codes Exemples

Nous avons appliqué le mode split à certains appels de sous-programmes, recherchant la confirmation expérimentale des intuitions formulées à la section 4.5. En particulier, nous voulons montrer l'intérêt de laisser l'utilisateur mener la stratégie checkpointing.

Les sous-programmes choisis pour être dédoublés étaient ceux qui illustrent le mieux la différence de mémoire et de temps d'exécution. Les critères pour choisir des sous-programmes se fondent sur deux valeurs, qui peuvent être obtenues en étudiant le code inverse généré. Ces valeurs sont : la taille du snapshot et la taille de la bande. L'implémentation des deux valeurs est basée sur des appels de PUSH, ainsi la comparaison entre ces valeurs est directe.

UNS2D

UNS2D est un solveur CFD. Il possède 2.055 LOC. La version différenciée inverse possède 2.200 LOC.

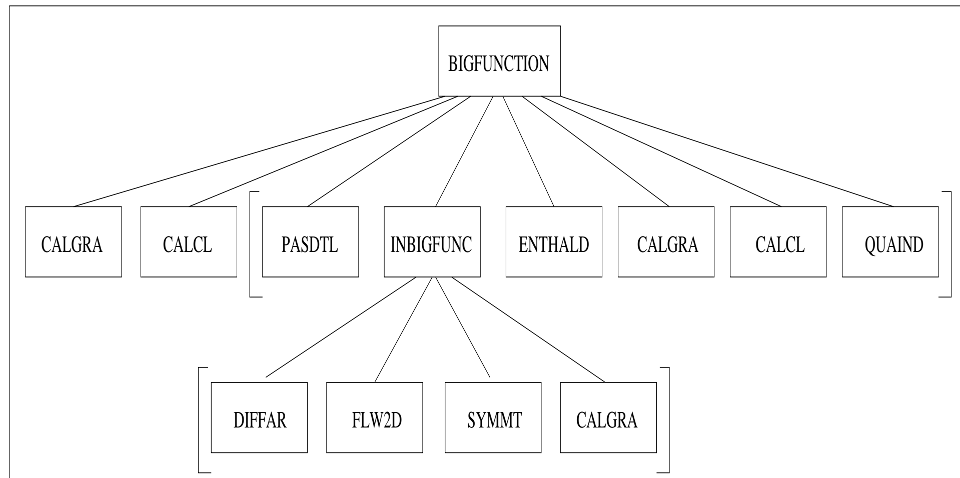

Figure 4.8: Organigramme de UNS2D.

Sur les figures 4.8 et **??**, des boucles sont notées par des crochets. Par exemple, sur la figure 4.8 nous avons deux boucles, une qui couvre du PASDTL DE SOUS-PROGRAMME au QUAIND DE SOUS-PROGRAMME, et un second qui inclut les sous-programmes de tous les INBIGFUNC. Ces boucles sont le segment du programme qui consomme la majeure partie de la mémoire et du temps.

Dans le Tableau 4.2 nous pouvons observer l'information requise pour déterminer les placements des checkpointing. Ceci parce que nous pouvons employer ces informations pour décider pour chaque sous-programme de son statut de checkpointing. La profiling information a été obtenue en utilisant une prolongation du TAPENADE, qui produit du code différencié avec les instructions nécessaires pour effectuer le calcul de profiling information pendant l'exécution.

Les quatre premières expériences 02 - 05 du Tableau 4.3 montrent un gain de temps et de mémoire, nous rappelant le cas où *tape* $<$ *snp* (Figure 4.7). C'est en effet ce que nous observons quand nous mesurons les tailles rélles de la bande et du snapshot pour les sous-programmes en question. Par conséquent, quand chacun des ENTHALD/ CALGRA/CALCL/QUAIND est dédoublé le programme économise de la mémoire pour le snasphot sans en employer autant pour la bande. En même temps, il économise du temps parce que le sous-programme n'est pas exécuté deux fois. Notez que le cas du ENTHALD est quelque peu différent, la recherche est ouverte pour ce cas.

Depth	Subroutine	Snapshot	Tape	#calls
0	BIGFUNC_B	0	184	1
1	PASDTL_B	15	191.4	734
1	INBIGFUNC_B	152	649.7	734
1	ENTHALD_B	30.5	46.9	734
1	CALGRA_B	30.5	21.5	734
1	QUAIND_B	22.8	16.1	13
2	DIFFAR_B	53.4	656.3	1468
2	FLOW2D_B	30.5	78.2	3674
2	SYMMT_B	15.6	0	3674
2	CALGRA_B	45.8	21.5	2940
3	GRDT1_B	0	46.9	1468
3	CONCON_B	0	32.5	3674
3	CONRIE_B	30.5	3.9	3674

Tableau 4.2: Profiling information sur les snapshots, les bandes et le nombre d'appel aux sous-programmes de UNS2D.

Experiences		Temps		Mémoire	
Id	Description	Total [s]	**% gain**	Pic [Mb]	**% gain**
01	Joindre-All stratégie	41.69		184.69	
02	split mode sur tous les appels de placement CALCL	37.66	9.7	167.53	9.3
03	split mode QUAIND	37.54	9.9	162.13	12.2
04	split mode sur tous les appels de placement CALGRA	36.63	12.1	163.92	11.2
05	split mode ENTHALD	34.33	17.6	162.17	12.2
06	split mode INBIGFUNC	31.83	23.6	468.13	-153.5
07	02 et 05	33.95	18.6	163.20	11.6
08	03 et 06	31.75	23.8	446.82	-141.9
09	02, 04 et 05	35.81	14.1	174.45	5.5
10	02, 05 et 06	35.49	14.8	533.23	-188.7
11	02, 03, 04 et 05	38.50	7.6	184.45	0.13
12	02, 04, 05 et 06	30.92	25.8	408.88	-121.4
13	split mode sur tous les programmes précédents	32.67	21.6	443.56	-140.2

Tableau 4.3: Performance en mémoire et en temps pour le code UNS2D.

L'expérience 06 montre un gain de temps au dépend d'une plus grande utilisation

de mémoire. Comme nous l'avons supposés à propos des simulations sur la figure 4.6, ceci correspond au cas où $snp < tape$. C'est aussi ce que les gens avaient à l'esprit quand le checkpointing a été proposé pour la première fois, et dans cette situation le checkpointing est vraiment un compromis temps/mémoire. Par conséquent, le INBIGFUNC CHECKPOINTING (en d'autres termes le mode joint) est un choix judicieux quand la capacité de la mémoire est limitée.

Les expériences 07 - 13 peuvent être séparées en deux parties : si le INBIGFUNC est checkpointed (08, 10, 12 et 13) ou pas (07, 09 et 11). Le critère de séparation souligne le poids relatif du INBIGFUNC DE SOUS-PROGRAMME.

Les expériences 07, 09 et 11 montrent un comportement remarquable sur les performance en temps d'exécution. L'économie de temps d'exécution des sous-programmes combinés de mode slipt devrait s'accumuler, concernant ce que nous observons dans les figures 4.6 et 4.7, mais étonnamment la performance en temps d'exécution pour ces expériences montre un comportement opposé. En particulier, l'économie de temps d'exécution de l'expérience 11's (3.18s) est plus petite que l'économie de temps d'exécution (4.03s, 4.15s, 5.03s et 7.36s) de chacun des sous-programmes qui composent l'expérience elle-même. Ce comportement nous mène à faire davantage d'expérimences et d'analyse afin de le rendre consistant avec le modèle de checkpointing décrit.

En conclusion, nous conseillons d'employer la stratégie de split mode donnée par l'expérience 12 en cas de demande d'économie de temps d'exécution. D'autre part, les expériences 03 et 05 permettent une économie de mémoire jusqu'à 12%.

4.6 Conclusions

Afin de raffiner notre modèle de mode DA inverse pour produire le meilleur code possible en termes de consommation de mémoire, sans diminuer l'efficacité en terme de temps d'exécution, nous avons deux voies d'action principales. La première consiste en une optimisation locale du code généré. Pour réaliser cette optimisation nous améliorons les analyses existantes de flot de données, introduisant ce que nous avons appelé des analyses de flot de données d'adjoint.

Nous avons décrit des analyses de flot de données pour un but particulier, employées pour améliorer les exécutions des codes générés. Les résultats expérimentaux montrent que les améliorations réduisent le pic de consommation de mémoire jusquà 49%. Le temps d'éxécution est aussi réduit en moyenne de 16%.

La seconde voix d'action utilise la relation de compromis entre la sauvegarde et le recalcul, au niveau des segments de code. Cependant, ce type d'optimsation peut

être vu de manière globale. La technique principale correspondant à ce type est le mécanisme appelé checkpointing, qui économise de la mémoire en étant basé sur l'échange entre sauvegarde et recalcul.

Nous sommes partis de l'observation que la stratégie de checkpointing pour chacun des appels de sous-programme, est bien que sûr du point de vue de la mémoire, loin d'être optimal. Les simulations à la fois du point de vue des petits exemples et des expériences avec des programmes réels montrent que certains sous-programmes ne devraient jamais être checkpointed et que d'autres doivent l'être selon la mémoire disponible.

La grande variété de situations possibles rend l'objectif d'obtenir une sélection automatique des emplacements du checkpointing encore éloigné. Il semble cependant raisonnable de laisser ce choix à l'utilisateur à travers une interface adaptée. Nous avons discuté les développements que nous avons effectués dans l'outil de DA TAPENADE pour ajouter cette fonctionnalité. Cette nouvelle fonctionalité nous a permis d'entreprendre des expériences étendues sur de vrais codes, ce qui a justifié a posteriori nos hypothèses sur ce problème de checkpointing optimal et suggèré les critères appropriés pour un futur outil d'aide notamment concernant, pour chaque sous-programme, son temps d'exécution, ses tailles de bande et de snapshot, et le temps requis par le trafic de PUSH et de POP de bande.

Les résultats sont vraiment très encourageant, ainsi la prochaine étape est de rechercher une manière automatique pour découvrir la stratégie checkpointing optimale. Une stratégie automatique pour placer les chekcpoints a pu être basée sur le profiling de temps d'exécution du programme original ou même du code différencié lui-même. Ceci suggère un processus d'améliorations itératives des codes différenciés inverses, basé sur des exécutions précédentes, assez comme ce qui est fait en compilation itérative [15].

Chapitre 5

Conclusions et perspectives de recherche

Cette thèse est composé de deux parties principales. La première partie est consacrée à informer l'utilisateur sur la possible utilisation incorrecte possible des programmes généres produits par la DA. Nous appelons ce problème le problème de validité de DA. La deuxième partie est concentrée sur réduire la consommation de mémoire du mode inverse de DA. Nous appelons ce problème le problème de mémoire du mode inverse de DA.

Dans la suite nous détaillons d'abord le travail que nous avons fait pour adresser le premier problème. Ceci sera suivi du même exercice pour le deuxième problème.

5.1 Sommaire et conclusions

5.1.1 Le Problème De Validité

Les outils de la DA supposent la différentiabilité des fonctions implémentées par les programme donné. Cette hypothése est fondamentale, parce que le mécanisme de la DA est basé sur l'application systématique de la règle de dérivation en chaînes pour chaque fonction élementaire á differencíer composant de la fonction. Mais, parfois ces fonctions élementaires ne sont pas rǵulieres ce qui met en cause la différentiabilité globale. Autres sources de fausse dérivation sont les commutateurs de contrôle principalament provenant de instructiones conditionnelles. Ces commutateurs rendent la plupart des programmes seulement différentiables por morceaux. Les dérivées, calculées dans ces cas par des ensembles d'instructions peuvent être parfois contradictoires prês des commutateurs. En outre, les programmes différenciés peuvent renvoyer des dérivés même si la fonction implémenté n'est pas différentiable. Malheureusement, a l'utilisation quotidienne de DA on ne tient pas compte de ce type de probléme dont

on se rend compte seulement quand on obtient des faux résultats.

Nous avons proposé deux méthodes pour aborder le problème de la non-différentiabilité dans les programmes différenciés avec la différentiation automatique. La première méthode que nous avons proposée PADE nous permet de calculer le domaine de validité pour une entrée donnée avec une précision acceptable. Malheureusement, bien que PADE nous fournis une description complète du domaine de la validité, son coût est prohibitivement haut, dans le temps d'exécution et en consommation de mémoire.

Le problème principal pour appliquer cette méthode est la taille de l'ensemble de contraintes. Pour faire face à ce problème nous avons proposé plusieures alternatives. Ces alternatives incluent : la baisse automatic/manual des contraintes et le changement de la représentation l'espace de solution. Malheureusement, aucune de ces alternatives permet de réduire de manière significative le coût de la méthode. Par conséquent, nous avons constaté qui'il assez difficile de réduire le coût pour rendre cette méthode pratique.

La deuxième méthode que nous avons proposée PAD, calcule les intervalles de suivant une direction donnée des d'entrées. Dans ces intervalles, les dérivées retournés n'ont aucun problème de différentiabilité. Le coût informatique du nouveau mode est marginal par rapport au le coût informatique de PADE.

Le seul inconvénient de PAD est que l'information retournée par la méthode est partielle, parce que cette information est calculée suivant une certaine direction dans l'espace d'entrée. Il est possible d'obtenir une approximation de l'information complète, mais cela ajoutent un surcoût à la méthode.

La deuxième méthode peut être vue comme autre extrémité du spectre. le temps d'éxecution et la conommation en mémoire sont basse, mais le domaine calculé informe juste au sujet d'une direction spécifique de dérivation dans l'espace d'entrée. Néanmoins, la méthode s'est avérée utile comme nous avons montré dans nos expériences.

PAD est facile à employer et elle est entièrement implémentée dans le outil DA TAPENADE. En raison de son bas coût, nous croyons qui est une manière efficace de traiter les problèmes de validité des codes différenciés.

5.1.2 Le problème de mémoire du mode inverse

Le mode inverse calcule les codes d'adjoint, en particulier les gradients. En raison de la structure du mode inverse de DA, le calcul de ce type de programme est très efficace en termes de temps d'exécution, mais a un inconvénient. C'est parce que certaines variables intermédiaires sont redéfinies pendant le forward sweep, les valeurs

tenues por ces variables ne sont pas accessibles au calcul les dérivés dans le backward sweep.

Afin de fournir les valeurs intermédiaires exigées par le mode inverse, nous avons adopte la stratégie Stocker-Tout (ST) [8]. Nous avons étudié le deux optimisations pour la stratégie ST. La première l'optimisation apellée échelle fine, est basée sur l'analyse de flot de données. Cette analyse fonctionne sur une représentation statique du programme complet, et les améliorations sont visibles aux niveau des instructions. La seconde est l'optimisation de échelle macro, cette optimisation consiste principalement en mécanisme checkpointing, qui implique de grands segments de code.

Afin de calculer les plus petits ensembles de variable pour stocke/recupérer, et les quelques instructions à génerer, nous avons amélioré le analyse existant de flot de données adjoint . Nous avons ajouté également une nouvelle analyse, qui permet de calculer l'ensemble précis d'instructions qui sont vraiment exigées pour calculer les dérivés en mode inverse. En employant ces analyses nous avons pu présenter un modèle raffiné pour le mode inverse. Ces améliorations sont actuellement implementées dans notre outil de DA.

Les analyses des flot de données sont fondamental dans la génération des programmes differenciés efficaces. En particulier, Les analyses des flot de données adjoint sont les outiles principaux pour générer le meilleure backward sweep code. En outre, les analyses permet de calculer les plus petits ensembles de valeurs pour stocker pendant le forward sweep, et aussi permet de réduire le nombre d'instructions de forward sweep en indiquant quelles instructions originales ne sont pas nécessaires pour calculer les dérivés.

Checkpointing est une manière attrayante d'exploiter la différence entre stocker et recalculer. Une stratégie de checkpointing se compose de deux éléments, de snapshot et de placement de points de contrôle. Nous présentons une équation formelle pour calculer le snapshot basé sur l'analyse ci-dessus. Cette formalisation et le fait que le checkpointing change la structure des programmes differenciés nous forcent à adapter le modèle de DA pour le mode inverse. Par conséquent, nous présentons un modèle prolongé de DA qui inclut le checkpointing. Cet modèle mets les points de contrôle systématiquement avant les appels de sous-programmes. Cette stratégie est suboptimale, et étant donné qui pour quelques segments de code est un meilleur pas point de contrôle d'utilisation, nous donnons la possibilité à l'utilisateur pour choisir l'emplacement du checkpointing. Toutes les contributions mentionnées sont actuellement mises en application dans notre outil TAPENADE. Cela nous permettent de foire des expériences complètes avec des codes scientifiques de vrai-taille.

Le analyse de flot de données et checkpointing sont liés, et les améliorations du

premier ont un impact positif sur la deuxiemme. Le rapport entre elles vient du fait que la définition de snapshot est basée sur le analyse de flot de données .

Checkpointing a une bonne performance, réalisant la diminution jusqu'à de 35% de la consommation maximale de mémoire, et la diminution jusqu'à de 90% du temps d'exécution. Par conséquent, une bonne stratégie checkpointing peut améliorer l'exécution des codes différenciés en mode inverse.

5.2 Remarques-conclusion

Dans cette thèse nous avons couvert deux aspects importants de la DA, chacune d'eux est liés à des caractéristiques fondamentales de DA, telles que la fiabilité et la performance.

Le premier aspect est lié à la fiabilité des dérivés, grace à notre recherche les utilisateurs qui doute de la différentiabilité de leurs modèles implementés auront la possibilité de valider leurs entrées et éxecuter leurs codes sans hésitation.

Le deuxième aspect est lié à la performance du programme différencié en mode inverse. Nous avons etabli des manières qui permettent à l'utilisateur d'exécuter leurs programmes sans être occupés par la question d'espace mémoire. C'est un grand souci quand les programmes originaux sont grands ou les internals des programmes originaux incluent des structures de données colossales. Grâce à nos développements l'utilisateur sera en mesure pour accorder la stratégie checkpointing qui peut mener à une économie importante dans l'espace mémoire aussi bien que dans le temps d'exécution.

Bibliographie

[1] Aho A., Sethi R., and Ullman J. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools.* SIAM, Philadelphia, PA, 1996.

[3] B. Dauvergne and L. Hascoët. The data-flow equations of checkpointing in reverse automatic differentiation. In *International Conference on Computational Science, ICCS 2006, Reading, UK*, 2006.

[4] C. Faure and U. Naumann. Minimizing the tape size. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 34, pages 293–298. Springer, New York, NY, 2001.

[5] David M. Gay. Semiautomatic differentiation for efficient gradient computations. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering. Springer, 2005.

[6] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[7] Andreas Griewank. Automatic directional differentiation of nonsmooth composite functions. In Roland Durier, editor, *Recent Developments in Optimization, French-German Conference on Optimization*, pages 155 – 169, Dijon, 1994. Springer Verlag.

[8] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[9] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Software*, 26(1):19, 1999.

[10] L. Hascoet, U. Naumann, and V. Pascual. "to be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8), 2004.

[11] L. Hascoët and V Pascual. TAPENADE 2.1 user's guide. Technical report 300, INRIA, 2004.

[12] Laurent Hascoët and Mauricio Araya-Polo. The adjoint data-flow analyses: Formalization, properties, and applications. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lecture Notes in Computational Science and Engineering. Springer, 2005.

[13] Laurent Hascoët, Mariano Vázquez, and Alain Dervieux. Automatic differentiation for optimum design, applied to sonic boom reduction. In V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *Computational Science and Its Applications – ICCSA 2003, Proceedings of the International Conference on Computational Science and its Applications, Montreal, Canada, May 18–21, 2003. Part II*, volume 2668 of *Lecture Notes in Computer Science*, pages 85–94, Berlin, 2003. Springer.

[14] Antony Jameson. Aerodynamic design via control theory. *J. Sci. Comput.*, 3(3):233–260, 1988.

[15] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and H.A.G Wijshoff. Iterative compilation in program optimization. In *In Proc. CPC2000*, pages 35 – 44, 2000.

[16] Gordon E. Moore. Cramming more components onto integrated circuits. In *Proceedings of the IEEE*, volume 86 (1), pages 82 – 85. IEEE, 1998.

[17] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufman, 1997.

[18] U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 1039–1048, Berlin, 2002. Springer.

[19] Thomas Kaminski Ralf Giering. Generating recomputations in reverse mode AD. In Uwe Naumann George F. Corliss, editor, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, chapter 33, pages 283–291. Springer Verlag, Heidelberg, 2002.

[20] Julia Sternberg and Andreas Griewank. Reduction of storage requirement by checkpointing for time-dependent optimal control problems in ODEs. In H. Martin Bücker, George F. Corliss, Paul D. Hovland, Uwe Naumann, and Boyana Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, pages 99–110. Springer, 2005.

[21] R. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

[22] Y. Xiao, M. Xue, W. Martin, and J. Gao. Development of an adjoint for a complex atmospheric model, the ARPS, using TAF. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, 2005.