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

Mauricio Araya-Polo

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

Une thèse présentée

par

Mauricio Araya Polo

á

l'université de Nice Sophia Antipolis

pour obtenir le titre de

Docteur en Sciences

specialité

Informatique

Thèse soutenue le 24 Novembre 2006 devant le jury composé de:

M. André Galligo	(Président)
Mme. Christine Eisenbeis	(Rapporteur)
M. Trond Steihaug	(Rapporteur)
M. José M. Cela	
M. Laurent Hascoët	

Sophia-Antipolis, France

Thesis advisor
Laurent Hascoët

Author
Mauricio Araya Polo

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

Abstract

The context of this work is Automatic Differentiation (AD). Fundamentally, AD transforms a program that computes a mathematical function into a new program that computes its derivatives. Being automatic, AD can spare a large amount of development effort. AD is increasingly used in Scientific Computing, but some problems still delay its widespread use. In this thesis we propose elements of solution for two of these problems. The first problem is non-differentiability of most real-life programs for some particular inputs. AD tools often overlook this problem. However users need a strong degree of confidence in the derivatives they obtain, to use them e.g. in optimization engines. Non-differentiability in programs may come from various causes, from the mathematical model to the actual implementation choices. Practically, non-differentiability in programs is embodied by the presence of control. Rather than studying notions of extended derivatives, our approach is to describe the domain in the input space for which the function actually remains differentiable. We describe several alternatives to find this domain of validity and we evaluate their complexity. We formally study one method to compute the complete domain, but we discard it because of its cost. Alternatively, we propose a simpler directional method that we have implemented and validated on several examples. The second problem is efficiency of the reverse mode of AD, which computes gradients and is therefore of high interest. Reverse AD programs use the intermediate values from the original program in the reverse order, and this has a cost, whatever the strategy used. Available strategies all rely on a combination of storing and recomputing intermediate values, thus costing memory space and execution time. The central tactic, called "checkpointing", trades duplicate execution of complete segments of the code for memory space. In this work, we formalize the static data-flow behavior of reverse AD programs, taking checkpointing into account. Based on these formal results, we contribute two improvements to reverse AD strategies. First, the data-flow analysis lets us reduce the number of stored values to a minimum, and we give elements of proof of minimality. Second, we obtain indications on the code segments that are most appropriate for checkpointing. To experiment on checkpointing schemes, we extend the data-flow analyses and the reverse mode in our AD tool. We show the benefits on several large Scientific Computing codes.

Directeur thése
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 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 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.

Contents

Title Page i
Abstract ii
Resume iii
Table of Contents iii
List of Figures viii
List of Tables x
Citations to Previously Published Work xii
Acknowledgments xiii
Dedication xiv

1 Introduction **1**
1.1 Abusive Use of Derivatives 3
1.2 Memory Shortage in Reverse Mode 4
1.3 Overview of the Thesis 6

2 Automatic Differentiation **8**
2.1 AD: Motivation, Goal and Framework 9
2.1.1 Motivation 9
2.1.2 AD's Goals 9
2.1.3 Framework 10
2.2 Programs and Mathematical Functions 11
2.2.1 Program Representation 11
Programs and Instructions 11
Control Flow and the Flow-Graph 12
Subroutine Calls and the Call-Graph 12
2.2.2 Mathematical Functions Implemented by Programs 13
Structure 13
Visualization 14
2.3 The Chain Rule 16
2.3.1 The Chain Rule and Programs 16
2.3.2 Example 18
2.3.3 The Chain Rule on DAG 18
2.4 The Tangent Mode of AD 20

2.4.1 Definition and AD Model 20
2.4.2 Example 21
2.4.3 The Tangent Mode on DAG 22
2.5 The Reverse Mode of AD 23
2.5.1 Definition and AD Model 23
2.5.2 Example 24
2.5.3 The Basic Problem of the Reverse Mode 25
2.5.4 The Reverse Mode on DAG 26
2.5.5 Relationship between Tangent and Reverse Modes 27
2.6 Theoretical Performances of AD Modes 27
2.7 Strategies and Architectures for AD tools 29
2.7.1 AD Implementation Strategies 29
Overloading 29
Source Transformation 30
2.7.2 AD Tool Architectures 31
Source-to-Source Architecture 31
Other Architectures 33
2.8 Examples of AD Application 34

3 The Domain of Validity of Derivatives 35
3.1 The Validity Problem 37
3.2 Analysis of the Validity Problem 41
3.2.1 The Differentiability of Intrinsic Functions 41
3.2.2 Differentiability of Conditional Statements 43
3.3 Defining and Propagating Validity Information 45
3.3.1 Validity Information Definition 46
3.3.2 Propagating Validity Information 47
3.4 Half-Spaces Backward Propagation (HSBP) 48
3.4.1 Definition 48
3.4.2 Problems with HSBP 49
Size and Representation 49
Reducing the Size of the Set of Constraints 49
Changing the Constraint Representation 50
3.5 Directional Forward Propagation (DFP) 52
3.5.1 Definition 52
3.5.2 Problems with DFP 54
3.5.3 DFP Implementation 54
3.5.4 DFP and AD Modes 56
3.6 Experimental Results 56
3.6.1 Basic Example 56
3.6.2 Experiments with the Newton Method 59
3.6.3 Experiments with Real-Life Scientific Programs 64

3.7 Related Techniques . . . 65
3.7.1 Interval Extension . . . 65
3.7.2 Sub-differentials . . . 66
3.7.3 Laurent Series . . . 67
3.8 Conclusions and Future Work . . . 68

4 Data-Flow Analyses and Checkpointing for the Reverse Mode 70
4.1 The Memory Consumption Problem of the Reverse Mode . . . 72
4.1.1 Store-All Strategy vs Recompute-All Strategy . . . 72
4.1.2 Experimental Measurements . . . 74
4.1.3 The Store-All Strategy and Memory Constraint . . . 76
4.2 Classical Strategies for the Store-All Approach . . . 77
4.2.1 Fine-Grain Strategies . . . 78
Data-Flow Analyses . . . 78
Recomputing versus Storing . . . 81
4.2.2 Coarse-Grain Strategies . . . 82
Checkpointing . . . 82
Assumption Regarding the Tape and the Snapshot . . . 84
Checkpoint Placements . . . 84
Snapshot Definition . . . 86
4.3 A Formal Model of Store-All Reverse Mode of AD . . . 88
4.4 Contributions to the Fine-grain Strategies . . . 89
4.4.1 Adjoint Data-flow Analyses . . . 89
Adjoint Liveness Analysis . . . 90
Refined TBR Analysis . . . 90
Adjoint Write Analysis . . . 91
4.4.2 Improved Model of the Store-All Reverse Mode of AD . . . 91
4.4.3 Experimental Measurements . . . 92
4.5 Contributions to Coarse-grain Strategies . . . 93
4.5.1 A Formal Model of Store-All Reverse Mode AD with Checkpointing . . . 93
4.5.2 Improved Snapshot Definition . . . 93
4.5.3 Example . . . 94
4.5.4 The Systematic Checkpointing . . . 94
Simulation of Hybrid Strategies Assuming $Snapshot < tape$ . . . 97
Simulation of Hybrid Strategies Assuming $Snapshot > tape$ . . . 98
4.5.5 Implementation . . . 99
Implementation of the Split Mode in TAPENADE . . . 99
4.5.6 Experimental Observation of Problems and Results . . . 100
Example Codes . . . 100
STICS . . . 100
UNS2D . . . 102

SONICBOOM 105
4.5.7 Discussion 106
4.6 Conclusions and Future Work 107

5 Conclusions and Further Research Directions 110
5.1 Summary and Conclusions 110
5.1.1 The Validity Problem 110
5.1.2 The Memory Problem of The Reverse Mode 111
5.2 Contributions 113
5.2.1 The Validity Problem 113
5.2.2 The Memory Problem of the Reverse Mode 113
5.3 Future Research Directions 114
5.3.1 The Validity Problem 114
5.3.2 The Memory Problem of the Reverse Mode 115
5.4 Concluding Remarks 115

Bibliography 116

A 123
A.1 Addendum to TAPENADE Tutorial 123
A.1.1 Requirements 123
A.1.2 Domain of Validity 123
Procedure 123
Example of Application 124
A.1.3 Checkpointing 126
Procedure 126
Example of Application 126

B 133
B.1 Program Examples 133
B.1.1 Extended Results for Validity Information with Newton Method 133

List of Figures

1.1 Sonic Boom Optimization, CFD using a gradient based method [34]. . 1
1.2 From models to implemented derivatives. 2
1.3 Non-differentiable functions for particular input. 3
1.4 Reverse mode of AD with SA strategy. 5

2.1 Two valid execution paths. 12
2.2 Call-graph: nested calls. 13
2.3 Computational graph and evaluation list for Formula 2.7. 15
2.4 Vertices hold the elementary mathematical functions and along the edges the partial derivatives of the destination vertex 19
2.5 General view of AD tool structure. Compiler-like structure. 32
2.6 General view of AD tool TAPENADE structure. 33

3.1 Plot of the example and the modified example around a given input. . 39
3.2 Plot of directional derivatives with input space direction $(x1d, x2d) = (1, 1)$. 41
3.3 Differentiated flow-graph for example. 43
3.4 Examples of problematic functions. 44
3.5 Representations for an arbitrary example. 52
3.6 Directional representation for an arbitrary example. 54
3.7 Experiments with two directions of the input space, and the computed validity information. 59
3.8 Newton method using derivatives generated by AD. 61
3.9 Examples of piecewise functions for the Newton method. 61
3.10 Piecewise functions. 63
3.11 Sub-differential example. 67

4.1 Plot of RA and SA strategies, where the horizontal axis represents the amount of values currently on the stack. 75
4.2 Experimental results. Graphical distribution in a memory vs time. . . 76
4.3 Graphical projection of local optimization for both, SA and RA strategies. 77
4.4 Example dependency graph. 79

4.5 Predecessor (P_i) and successor (S_i) blocks of basic block B. 81
4.6 Checkpointing on Reverse Mode AD. 83
4.7 Checkpointing on all calls in Reverse Mode AD (joint-all mode). . . . 96
4.8 No Checkpointing in Reverse Mode AD (split-all mode). 97
4.9 Hybrid approach (split-joint) 97
4.10 Simulation results, tape = 10, snapshot = 6. 98
4.11 Generic Numerical Results, tape = 6, snapshot = 10. 99
4.12 STICS Call-graph. 101
4.13 UNS2D Call-graph. 103
4.14 SONICBOOM Call-graph. 105

A.1 UNS2D Call-graph. 127

List of Tables

2.1 Automatic generated tangent differentiated code. 21
2.2 Automatic generated reverse differentiated code. 24
2.3 Automatic generated reverse differentiated versions of example code. . 25
2.4 Derivatives for elementary operations. 28

3.1 Automatic generated tangent differentiated modified code. 40
3.2 Function replacements and its derivatives for intrinsics in AD. 42
3.3 Representation of solution space, trade-off. 51
3.4 Algorithm to update β. 55
3.5 Tangent Differentiated example code with Validity Analysis. 57
3.6 Numerical results of the example, with $(x1d, x2d) = (1, 4)$. 58
3.7 Numerical results of the example, with $(x1d, x2d) = (1, -1)$. 58
3.8 Function $f()$ and its tangent differentiated version. 60
3.9 Function $f\_piecewise()$ and its tangent differentiated version. 62
3.10 Numerical results of the Newton method on piecewise function. . . . 63
3.11 Numerical results of Newton method and DFP on piecewise function. 64
3.12 Real-life program settings. 65

4.1 RA ruleset 73
4.2 SA ruleset 74
4.3 Experiments RA vs SA 75
4.4 TBR analysis example. 82
4.5 Computation of snapshot example. 87
4.6 Scientific codes description. 92
4.7 Time and memory improvements on three large scientific codes. . . . 92
4.8 Reverse differentiated version of example code of Table 4.5 95
4.9 From original call-graph to reverse differentiated with checkpointing call-graph. 96
4.10 Memory and time performance for STICS. 102
4.11 Profiling information about snapshots, tapes and number of calls to subroutines of UNS2D. 103
4.12 Memory and time performance for UNS2D. 104

4.13 Memory and time performance for SONICBOOM. 106
4.14 Time and memory improvements on three large scientific codes. . . . 108

A.1 Results from validated code of the example with direction (xd,yd) = (1,1). 126

Citations to Previously Published Work

Large portions of Chapters 3 have appeared in the following two papers:

> "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.

Most of Chapter 4 has been published as:

> "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.

Electronic preprints are available on Internet at the following URLs:

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

Acknowledgments

I am very glad to write this page. After all this time and work, the moment of look back and to express my gratitude has come. First or all, I am deeply indebted to my supervisor, Dr. Laurent Hascoët for offering me an opportunity within the TROPICS project. I would like to thank you for all the guiding, cooperation, encouragements, and lasting support throughout the research. To the members of the jury - Dr. Christine Eisenbeis, Dr. Trond Steihaug, Dr. José Maria Cela and the president of the jury Prof. Andre Galligo - I am grateful for the time you all devoted to reading this. It is my honor and I thank you for the advice and the constructive criticism that contributed to bringing the original draft to this final stage.

I would like to thank also, Dr. Johan Fabry for reviewing most of the thesis. His discussion and remarks have influenced the way of express the ideas of this work. It is an impossible task to acknowledge all the people that have help me to achieve this work. I am in debt to many other colleagues for the useful discussions, in particularly I wish to thank Benjamin Dauvergne, Stephen Wornom, Valérie Pascual, Alain Dervieux, Hicham Tber, Gonzalo Robledo, Tomas Barros, Nelson Morales, Mariano Vazquez, Christophe Massol, Javier Bustos, Shaun Forth, Jean Utke, Uwe Naumann. If your name is not listed, my gratitude is with you as much as with the people listed.

The sacrifices that this work has required, have been felt most strongly by my family and friends. Thus, I would like to express my gratitude to my family, my father and my late mother, and sister for laying the fundaments for this work. Last but not least, I want to thank my Anne-Cécile for all the support that she gave me during the years, specially considering the workload on her own back. You have been very helpful and enduring.

I also gratefully acknowledge the financial support I received from the Conicyt Chile, as part of a scholarship offered under the cooperation program between Conicyt and INRIA Sophia-Antipolis.

The solid work done by the open source community is also acknowledged in fact invaluable from a computer science research perspective. Debian GNU/Linux has provided a robust platform for research. For writing articles and the thesis, LaTeX, Vim, and Svn have been used.

Mauricio Araya Polo.
November 2006.

Dedicated to my father Pedro, my mother Yolanda and my sister Gabriela, to my beloved Anne-Cécile, and finally but no less important to my friends and relatives.

Chapter 1

Introduction

Scientific computing is concerned with constructing computer programs that implement the analysis and solutions for scientific and engineering problems. Mainly, the analysis and solutions implemented are based on mathematical models. For instance, some applications of scientific computing are: Computational Fluid Dynamics (CFD), weather forecast models, biological models, particle collision simulations.

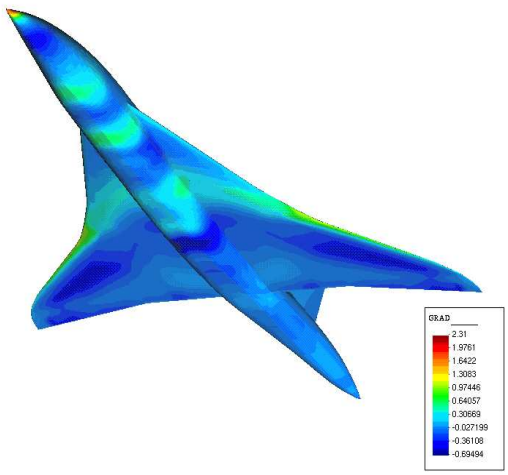


Figure 1.1: Sonic Boom Optimization, CFD using a gradient based method [34].

All those mathematical models rely on methods and algorithms that require derivatives. Therefore, the models are implemented as program required to implement these derivatives. In Figure 1.1, we can observe an example of application of scientific computing, specifically CFD research which include shape optimization [38], their mathematical methods heavily rely on derivatives, in particular gradients. In fact, in Figure 1.1 shows the variation of the gradients for an objective function, which relates the shape of the airplane and the pressure.

As we can observe in Figure 1.2, when we need an implementation of the derivatives we can either do it by hand or we can use Automatic Differentiation (AD) [5, 27].

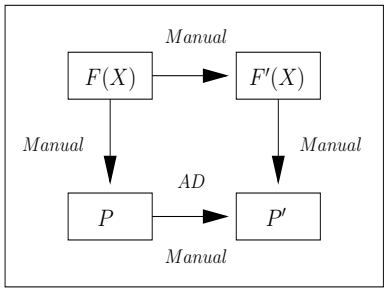


Figure 1.2: From models to implemented derivatives.

In Figure 1.2, the cost of implementing P', where P' is the the program that computes the derivatives required by model $F'(X)$, depends on which path we follow. The cheapest (in terms of development time) path goes from the mathematical model $F(X)$ through the manually implemented program P to finally using AD to the differentiated program P'. On the other hand, the manual implementation of program P' requires a lot of work, due to the size of codes and the complexity of the task, also the manual implementation is not reliable because it may introduce bugs.

AD is extremely fast, and the generated derivatives reliable. It is extremely fast because the AD models are based on the efficient application of the chain rule (differentiation of composed functions). For instance, using our AD tool TAPENADE [32], it takes less than ten minutes to generates derivatives for 50.000 lines of code (LOC). In contrast, doing this by hand would take many man hours of work. The derivatives obtained using AD are reliable because they are the result of the systematic and mechanic application of the chain rule, therefore these derivatives are exact and as precise as the machine precision. Also, the differentiated programs have an equally high efficiency as the efficiency of the original program.

AD has a long history as a way to differentiate programs, the first proposition was made in 1964 [62]. Although the AD models and tools have evolved to high level quality, still AD has its problems, inconsiderate use of AD may lead to unreliable derivatives, as we show later in Section 1.1. Furthermore, when AD is used to generate certain types of derivatives, for instance gradients, this may lead to memory shortage for large programs. In Section 1.2 we discuss this in more detail.

The goal of this thesis is to address the above two problems. In the remainder of this chapter we give an overview of each of the above problems together with the

solution we propose. Finally, we give an overview of the rest of the chapters of the thesis.

1.1 Abusive Use of Derivatives

The behavior of most programs with control flow depends on the input values. This is implemented by conditional statements, each conditional statement decides which is the next segment of code to be executed among a set of segments. AD models respect the logical structure of programs, therefore once AD is applied, each conditional statement chooses among a set of segment of code which implement derivatives. Given two slightly different input, the conditional statement may chose two completely different segments of code to execute. As a result, the differentiated program would return two completely different derivatives from the given slightly different input. This means that the derivatives generated by AD tools might be inconsistent and even incorrect. With incorrect we mean that AD tools implement derivatives even when the functions are non-differentiable for certain input, in particular when the functions are implemented using conditional statements.

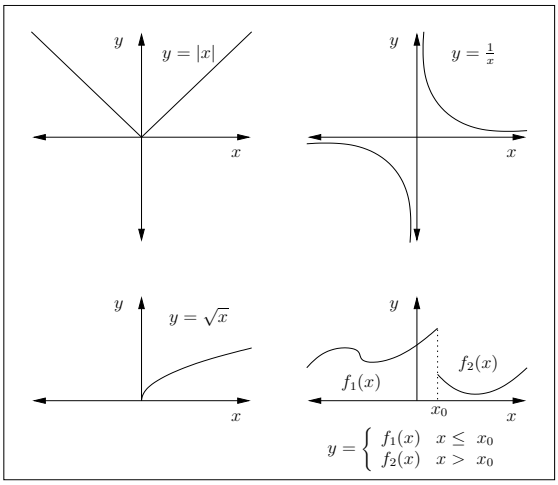


Figure 1.3: Non-differentiable functions for particular input.

In Figure 1.3 we can observe some of the cases that may produce unreliable derivatives, because AD tools generate differentiated codes and those codes can be executed with inputs that are not valid. Where the invalid input are those input for which the presented functions have non-differentiabilities, or input too close to radical changes in the definition of the functions, for instance the function in the bottom-right function of Figure 1.3.

AD tools can not modify the original program because the internals of the implemented models are not known. Thus, we can only notify the user that for certain input the derivatives are not reliable.

We aim to make the derivatives useful even under the described conditions, for this we want to provide the biggest safe neighborhood. This safe neighborhood is defined as a region in the input space of which we can obtain consistent derivatives. In order to provide that neighborhood we compute the biggest possible variation around the given input that does not change the segments of code selected by the conditional statements to be executed.

We have developed two strategies to compute the safe neighborhood. The first strategy, called HSBP, provides a complete and precise (at first order) description of the neighborhood, but has a prohibitive computational cost. The cost comes from the fact that a large number of conditional statements must be analyzed, and every analysis itself is computational expensive. This analysis relies on the reverse mode of AD, thus improvement on this mode, which are developed in the second part of this work, may help to reduce the computational burden of HSBP. The second strategy, called DFP, provides a less complete description of the neighborhood, at the same level of precision of HSBP, but at low computational cost.

We have conducted a number of experiments to validate the DFP strategy. We have found that the strategy indeed allows us to compute the safe neighborhood. This serves as warning of inconsistent derivatives or non-differentiabilities within the program.

1.2 Memory Shortage in Reverse Mode

Gradients are the most popular kind of derivatives used by mathematical methods. The most efficient way, in execution time, to generate gradients is to use what is called the reverse mode of AD [27]. Unfortunately, during the execution of the reverse mode a large amount of intermediates values need to be accessible. To cope with that problem we have two main possible solutions. The first strategy is called Store-All (SA) [27], which consists in storing the intermediate values until they are required. Thus the efficiency of the reverse mode comes at high price in memory consumption. The second strategy is called Recompute-All (RA) [53], which consists in recomputing the intermediate values every time they are required. Thus, RA drastically reduces the execution time efficiency of the reverse mode.

In this thesis we address the memory consumption problem of SA strategy. We chose this strategy because in the tool (TAPENADE) developed by our team we already

achieved high efficiency in execution time, using SA strategy, and we aim at reducing the memory usage without loosing too much in execution time. Therefore, in this thesis we investigate the trade-off between execution time and memory consumption, looking for provide the most efficient derivatives, in particular for gradients.

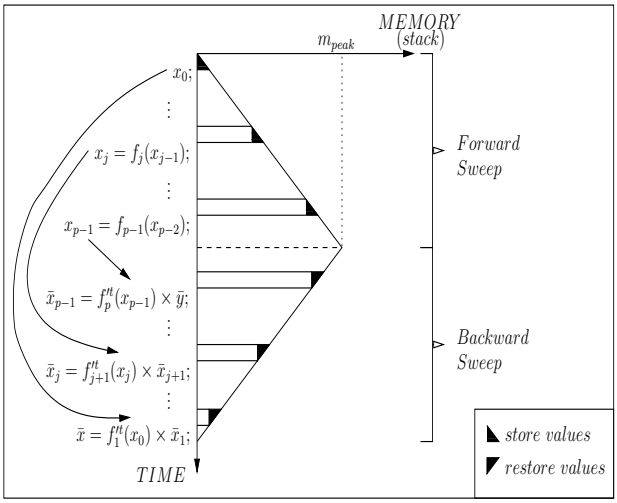


Figure 1.4: Reverse mode of AD with SA strategy.

In Figure 1.4 we can observe the general features of the AD reverse mode with SA strategy. Thus during the forward sweep some intermediate values are stored in the stack, thus the peak of memory usage is reached at the end of the this sweep. After, during the reverse sweep the values are restored allowing to compute the derivatives.

In order to address the problem, we have to two kind of strategies: fine-grain and coarse-grain strategies. Where fine-grain strategy focus in local optimization, at the level of instructions. Coarse-grain strategy treat the problem in big segments of code.

We perform research in both kind of strategies. Now, we give an overview of our contributions. First, we discuss our work in fine-grain strategy. Second, we present our research in coarse-grain strategy.

Fine-grain strategy rely on data-flow analysis. Data-flow analysis is a set of techniques which run on the source code of the program. The main goal of these techniques within AD is to assist in generating the best possible differentiated code. In particular, some of these techniques are focused on computing the sets of values to store for the reverse mode.

We improve the existing data-flow analyses, and also added new analysis. As a

result, we are able to obtain smaller sets of values to store, and fewer instructions to implement. Therefore, our tool TAPENADE generates codes of high quality and efficiency.

In contrast to fine-grain strategy, coarse-grain strategy is focused on taking advantage of the trade-off between storing and recomputing, where the trade-off is most effective, that is at the level of segments of code.

The most important coarse-grain mechanism is called checkpointing. In reverse mode of AD, and following SA strategy, some values of intermediate instructions are stored stack-wise. When these values are required to compute derivatives they are restored. Alternatively, checkpointing mechanism consist in deactivating the storing for some segments of code. When intermediate values are required to compute derivatives these segments of code are recomputed, and this time with the storing activated. Thus, checkpointing only stores what is needed to perform the re-computation of the checkpointed segment, this set of values is called snapshot. Usually, this amount of data is smaller than the stored data during the computation of the segment, thus we gain in memory. The trade-off here is that, because we need to recompute some segments, we have lost some execution time.

Using improved data-flow analysis from the fine-grain strategies, we are able to compute smaller snapshots. Furthermore, we introduce flexibility in the placement of the checkpoints. Before this, TAPENADE places the checkpoints systematically before every subroutine call, but this strategy is not optimal. We have provide the possibility to the user to selects the checkpoints placements.

We carried out experiments on a number of scientific programs, in order to validate our approach. The results are promising for both level of optimization. The results confirm also the expected level of improvements of these strategies.

1.3 Overview of the Thesis

We organized this document in three parts:

- Chapter 2: Automatic Differentiation.

  This chapter is devoted to introduce the basics of AD. We start defining our framework, which includes the way we represents programs, and the relationship between mathematical models and programs. After we introduce the chain rule and its relation with implementation of derivatives. We present the tangent mode of AD, which is the first specialization of the chain rule, this mode

generates directional derivatives. The second specialization of the chain rule is the reverse mode of AD, which generates adjoint code, which is very efficient to implement gradients. Finally we review performance issues of AD modes, and architectural characteristics of AD.

- Chapter 3: Domain of Validity of Derivatives.

  This chapter covers our research about the first mentioned problem. First, we analyze the problem in detail. Second, we present the general approach to the problem. After that, we present the first method based on the general approach, which results to be expensive. Thus, we discuss alternatives to improve it. We introduce a second method based on the general approach, this method results to be of an acceptable cost. We present the implementation of the second method. Finally, the experimental results obtained using the second method are presented and discussed.

- Chapter 4: Data-flow Analysis and Checkpointing Strategies for Reverse Mode

  This chapter address the second mentioned problem. The problem is presented in detail, including the two main strategies to solve it, RA and SA. After we present an overview of the classical techniques to improve the two main strategies. We chose to keep working only on SA strategy. We have mainly different levels of optimization. The first level, the one we called fine-grain level, is related to data-flow analysis. The second level, the one we called coarse-grain level, is related to the checkpointing mechanism. We presents the classical, improved and new data-flow analysis for reverse mode AD. Experimental results are presented for these analysis. The checkpointing mechanism is analyze in detail. A checkpointing strategy is composed by two elements, the checkpoints placement and the composition of the snapshots. We introduce an improved equation to compute the snapshot. We present different checkpointing strategies for reverse differentiated programs. Finally, experimental results are presented along with the discussion.

Chapter 2

Automatic Differentiation

This chapter introduces concepts and fundamentals of Automatic Differentiation (AD). AD is a general approach to generate programs that implement derivatives [37], it is general because it includes several strategies to obtain derivatives from a program, these are all valid, but with different levels of efficiency and reliability. The derivatives can be generated as first order approximations to a function, on the other hand exact derivatives can be obtained using the chain rule and the well known rules of differential calculus. The use of the chain rule is the generally accepted approach.

The chain rule can be used fundamentally in two ways: forward and reverse. Forward and reverse indicate the direction of application of the chain rule with respect to the inputs and to the outputs of the function. That dichotomy supports the two most classical AD modes, *tangent* and *reverse* mode [27]. We called tangent the first mode, the one related to the forward application of the chain rule, in order to avoid future complications with notation. The reverse mode, the one related to the reverse application of the chain rule, receives most attention by researches in the domain, because it is used to compute gradients, which are highly demanded in scientific applications.

The reverse mode is very efficient in execution time but is a great consumer of memory, which justifies the AD community [3] effort to improve it. In this chapter we present a practical and theoretical evaluation of these modes, and we introduce the main lines of improvement we investigated for the reverse mode.

Before introducing the formalizations we give the motivation and framework within the AD field where this thesis stands.

2.1 AD: Motivation, Goal and Framework

2.1.1 Motivation

The interest in AD is motivated by domains where computational derivatives are required, which can include all sciences and technology. Particularly, scientific computing is a great consumer of all order of derivatives, mainly because most methods rely on numerical analysis techniques, and those techniques are all based or partially based on differential calculus. Among the applications of scientific computing we can mention: Computational Fluid Dynamics (CFD), optimization, environmental simulations, financial modeling and computational physics.

For example, Gradient Based Methods (GBM) are iterative methods where the decisions regarding convergence are strongly tied to derivatives and their precision. When GBM are implemented in computer programs, the derivatives provided must be precise to ensure the correct flow of the algorithms. Also due to the iterative nature of the methods, these derivatives need to be computed efficiently.

After having mentioned one example where computational derivatives are useful, we can now clearly state what are the requirements to use computational derivatives within a scientific application. We call these requirements the goals of AD, and they are described in the next section.

2.1.2 AD's Goals

The main goals of AD are to provide precise and efficient derivatives. To provide precise derivatives AD fights on with two fronts, the first is the precision that can be achieved, and the second is related to the way the derivatives are obtained.

AD derivatives must be precise because most of the applications -mathematical models- which use these derivatives rely on the level of precision of the computer. Therefore, AD derivatives must be as precise as the computer precision. AD looks for automatic ways to obtain derivatives in order to reduce errors (bugs) introduced by human intervention. Consequently, AD became a human triggered but automatic process. In order to generate derivatives from a computer program composed by thousands to millions of lines of code, human intervention seems like a hazard. This is because it is a potential bug generator activity. Moreover we have a potentially low reliability of human-made derivatives. Ideally, human intervention should be reduced to provide the input, to launch the process and to integrate the differentiated code to the already existing application.

Computational derivatives should not degrade the applications performance. E-

fficiency is always required when resources are limited, in our case the derivatives will be part of large models which are very demanding on execution time and memory space. Especially, AD focuses on memory consumption due to fact that this resource is the most limited.

Mainly, AD provides first order derivatives, specifically directional derivatives and derivatives obtained by manipulation of the Jacobian matrix (Jacobian), like gradients and adjoints. Also, higher order derivatives can be obtained, for example the Hessian matrix and partial elements of this matrix.

2.1.3 Framework

There are mainly two ways of computing computational derivatives, one way is called *numerical* and the other way is called *symbolic*. In this thesis, we solely focus on the first. We discard the latter due to the complexity introduced by the manipulation of the formulae and extra amount of memory and computing time required.

Within numerical differentiation we have two main approaches to compute the derivatives, one is called *Divided Differences* (DD) and the other is called *Automatic Differentiation* (AD).

In order to obtain derivatives by the DD approach one of the equations of Formula 2.1 is applied.

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

A better approximation is possible with *centered* DD, as is follow:

$$f'(x) = \frac{f(x+\epsilon) - f(x-\epsilon)}{2 \times \epsilon} \tag{2.2}$$

Formula 2.2 is more expensive in execution time than Formula 2.1, because requires two execution of the original function.

We discard DD approach for the following three reasons. First, unlike AD where the precision is bound to the machine precision, this approach introduces truncation errors. Therefore the precision is limited by the manipulation of floating point. Secondly, the selection of the correct ϵ in Formula 2.1 is a never ending process of trial and error, which eventually affects the derivatives precision. Thirdly, at first view this approach looks cheap from the computational point of view, it just requires two evaluations of the original program plus some elemental operations. However, if derivatives for a large number of input variables are needed, the performance in terms of execution time decreases in direct proportion with the number of variables, which

is unacceptable in real life applications.

As the results of the above discussion, this thesis is devoted to the numerical-analytical approach of Automatic Differentiation.

In the following sections we will present the fundamentals in which AD is based. We start with the relation between computational programs and mathematical functions. The rest of the chapter is organized as follows. In Section 2.3 we present the chain rule. In Section 2.4 we introduce the tangent mode of AD. In Section 2.5 we introduce the reverse mode of of AD. In Section 2.6 we analyze the AD modes performance. In Section 2.7 we present the classical strategies and architectures for AD tools. Finally in Section 2.8 we review some examples of application domains of AD.

2.2 Programs and Mathematical Functions

2.2.1 Program Representation

Programs and Instructions

We consider programs as concatenated sequences of instructions, where instructions are represented by the symbol I_j, with $j \in \{1, q\}$. Thus, an arbitrary program P has the following form:

$$P = I_1 \ ; \ I_2 \ ; \ldots ; \ I_j \ ; \ldots ; \ I_{q-1} \ ; \ I_q \tag{2.3}$$

where the semicolon represents the concatenation operator.

Instructions can be classified into the following groups [14]: input/output, memory management, general statements (assignments and expressions), control flow and subroutine calls. The first group is not relevant to this work. Memory management (allocation/deallocation of memory) only plays a role in the reverse mode (Section 2.5).

The remaining groups are the most relevant to our work, basically because they implement the features of the mathematical models. The third group of instructions holds the numerical expressions, which are composed of the elementary operations and the mathematical functions given by the programming language (*intrinsic* functions). Ideally, this should be all that is needed to represent any mathematical model, but that is not true due to the fact that programs have a high complexity and large size. Therefore, to cope with complexity, programs make use of control structures (for instance, the conditional statement *if... then... else*, which we call a *test*) and

loops, both components of the fourth group of instructions. Finally, to cope with large programs, which may include multiple use of non sequential segments of code, programs can be divided in manageable pieces (subroutines), which introduces the use of subroutine calls.

Control Flow and the Flow-Graph

The control flow determines the dynamic behavior of programs, and reveals the existence of several sub-sequences of instructions which may perform the program execution. A sub-sequence of instructions s_i, where $s_i \subset P$, is called an *execution path* if the instructions in s_i compute a valid output of the program P for a certain input set. Thus, a program with a control flow structure is a set of possible execution paths, but when a program is executed, only one execution path is carried out.

If an instruction I_j of a program P implements a test, the Expression 2.3 will not change, because it is a static representation which hides the information flow within the program control structure. Thus, in order to represent the set of execution paths we make use of the flow-graph [1, 46] (Figure 2.1).

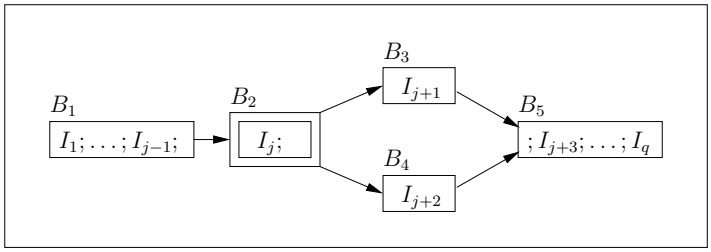


Figure 2.1: Two valid execution paths.

In Figure 2.1, the conditional structure in I_j introduces two possible branches (I_{j+1}, I_{j+2}). Therefore, the program P has the following execution paths: $s_1 = I_1 \; ; \ldots ; \; I_j \; ; \; I_{j+1} \; ; \ldots ; \; I_q$ and $s_2 = I_1 \; ; \ldots ; \; I_j \; ; \; I_{j+2} \; ; \ldots ; \; I_q$.

In order to simplify the notation and as it is usually presented [1], a sequence of instructions without control structure is called a *basic block* B_i, for instance in Figure 2.1 we have block $B_1 = I_1 \; ; \ldots ; \; I_{j-1}$. Also, a block which only holds a conditional statement (test) we called a *header block*, for instance we have a block B_2 in Figure 2.1.

Subroutine Calls and the Call-Graph

Next, we introduce the notation to manipulate subroutine calls, let us re-define program P as follows:

$$P = B_1 \ ; \ Call \ B \ ; \ B_2 \ ; \ Call \ C \ ; \ B_3 \tag{2.4}$$
$$C \ = B_1 \ ; \ Call \ D \ ; \ B_2$$

where P has a main body from where two subroutines (B and C) are called, and it has other three basic blocks, subroutine C has just two basic blocks, and one call to subroutine D.

Subroutines called by program P may call other subroutines. For example, subroutine C calls subroutine D (Figure 2.2). This mechanism is called nesting. The level of nesting called *depth* is arbitrary, and the depth is counted from level zero which is the program main body to the last level of called subroutines. This counting process is possible to carry out because the calling structure resembles, in runtime, the tree structure.

In order to represent graphically the subroutine call structure of programs we use the call-graph [1, 46] representation (Figure 2.2).

In Figure 2.2, we denote the subroutines with named boxes corresponding to the names given in algebraic form (Formula 2.4), the calls are represented by arrows.
The direction of the arrow indicates the calling direction, for example, program P calls subroutine B.

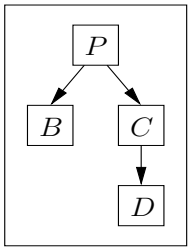


Figure 2.2: Call-graph: nested calls.

Note that, the control structure is closely related to the calling structure of the program, for instance, a subroutine might be called from a loop or from one branch of a test. Unfortunately, the call-graph representation fails to illustrate this dynamic behavior within subroutines. Therefore, we will use call-graphs to show only the static structure of the calls within a program, without repetitive or recursive calls, and assuming that all the subroutines are called at least one time, even if they are inside conditional statements.

2.2.2 Mathematical Functions Implemented by Programs

Structure

Mathematical models or functions are composed out of elemental mathematical functions. We denote these elemental functions as $f_i()$, with $i \in \{1, p\} \subset I\!N$. For example, we define the function F, which receives a vector X as input, that vector

is composed of variables from an input space of dimension n. Function F returns a vector Y which belongs to an output space of dimension m. Thus, F is expressed as follows:

$$Y = F(X) = f_p(\dots f_i(\dots f_1(X))) \tag{2.5}$$
$$F : X \in I\!R^n \rightarrow Y \in I\!R^m$$

After the computation of $f_1(X)$ the output vector of variables becomes the input vector X_2 to function $f_2()$. This process is repeated until the last function $f_p(X_p)$ is computed. Formula 2.6 represents the same mechanism, and introduces an easy-to-follow notation for functional composition, where the symbol $\circ$ represents the composition operator.

$$Y = F(X) = f_p(X_p) \circ \dots \circ f_i(X_i) \circ \dots \circ f_1(X) \tag{2.6}$$

where the intermediate vector of variables $X_i = f_{i-1}(X_{i-1}) \circ \cdots \circ f_1(X)$ for every $i \in \{1, p\}$ is obtained just after the evaluation of the elemental function $f_{i-1}()$, and given as input to function $f_i()$.

Programs that implement mathematical functions take advantage of this structure, and so, each elemental mathematical function $f_i()$ is represented by an instruction I_i. Thus, in principle, the number of instructions is the same as the number of elemental functions. In programs, the mechanism described in Formula 2.6 is executed in the same order, but the index numbers of the instructions are assigned with inverse order with respect to the index number of the elemental functions. For instance the first instruction to be executed is I_1, and usually it represents the last elemental function $f_p()$, which is the first elemental function to be evaluated in Formula 2.6 mechanism.

Usually, the function output is defined by different expressions depending on various input domains. In other words, the function expression is composed by several disjunct pieces. Once an input is given, only one of those pieces returns the output of the function. This kind of function is called *piecewise*, and it is implemented using conditional statements.

In programming languages, the elementary mathematical functions are implemented by the basic algebraic operations $\omega_1 = (+, -, /,$ etc.), plus some classical mathematical functions $\omega_2 = (sin, cos, abs,$ etc.), thus the set of elementary mathematical functions are defined as $\omega = (\omega_1, \omega_2)$.

Visualization

Computational Graphs are one way of visualizing programs that implement mathematical functions. In particular, Directed Acyclic Graphs (DAG) [48] are used.

These are composed by a set of vertices V, and a set of edges E which connect the vertices. The vertices $v_i \in V$ represent the elementary functions from ω. The edges $e_{i,j} \in E$ represent the dependency between the two sequential vertices, for instance $e_{i,j} = (v_i, v_j)$ with $v_i \prec v_j$ where $\prec$ is the dependency relationship, and $\prec^*$ its transitive closure. The DAG is acyclic because the condition $v_i \prec^* v_j \prec^* v_i$ never happens, or in other words, a vertex never depends on itself. Finally, the graph is directed because the edges only follow the direction origin (v_i) to destination (v_j), where $v_i \prec v_j$, denoted in the DAG by arrows.

In order to illustrate the previous definitions we present a simple example, given by the following formula:

$$Y = F(x_1, x_2) = \frac{sin(x_1^2 + x_2^2) \times (1 - x_2)}{x_1^2 + x_2^2} \tag{2.7}$$

$$F : X \in I\!R^2 \rightarrow Y \in I\!R$$

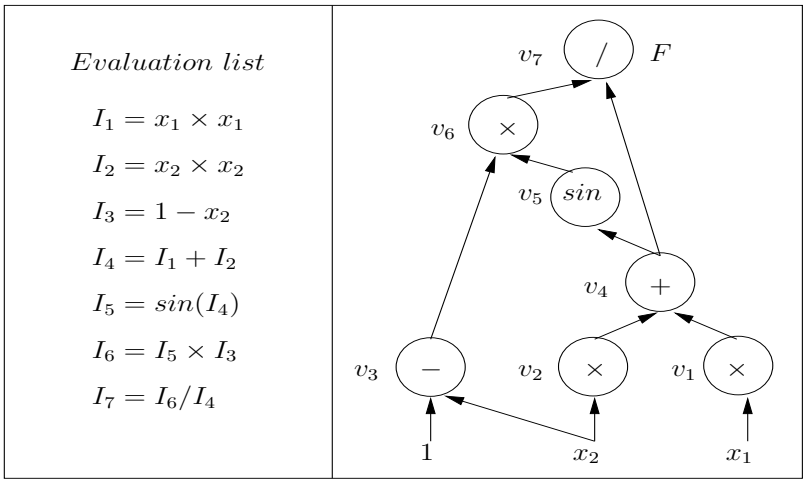


Figure 2.3: Computational graph and evaluation list for Formula 2.7.

The list of instructions which are computed by a program is called the *evaluation list*, for example the left square of Figure 2.3. The instructions in an evaluation list are implemented based on elementary functions, thus the operation count of every instruction just is one. As often happens, real-life programs evaluation lists include instructions composed by several elementary functions, for instance, in our example we may re-write instruction I_5 as follows $I_5 = (sin(I_4) \times I_3)/I_4$, thus reducing the evaluation list by two instructions. However, after that change, the operation count for the evaluation list remain the same, which means that we do not save execution time. On the other hand, we define fewer instructions and store fewer local values,

which could lead to reduced memory consumption. This remark depends on the computing platform, e.g. the hardware-software combination.

The order of the first three instructions of the evaluation list of Figure 2.3 can be re-arranged in different ways without modifying the output. That fact allows us to foresee possible improvements to the program performance, for instance reducing the number of temporary variables by re-using them several times.

AD takes advantage of the fact that no matter how complex a mathematical function is, it is possible to decompose it into fundamental functions, and the derivatives for those functions are well known. In the next section we present the basics of this mechanism.

2.3 The Chain Rule

2.3.1 The Chain Rule and Programs

The chain rule of calculus can be applied only if the involved functions are differentiable. For instance, for a function $F(X) = f_2(f_1(X))$, with $f_2()$ and $f_1()$ differentiable, the chain rule takes the following form:

$$Y' = F'(X) = f_2'(f_1(X)) \times f_1'(X) \tag{2.8}$$

or alternatively,

$$Y' = F'(X) = (f_2'(X_2) \circ f_1(X)) \times f_1'(X) \tag{2.9}$$

In the context of this work, where the function F is a mathematical model implemented by a computer program, we assume that each elementary mathematical function $f_i()$ component of F is differentiable. Therefore, the composition of such functions, for instance Formula 2.6 is differentiable, and its differentiated form is:

$$Y' = F'(X) = f_p'(X_p) \times \ldots \times f_i'(X_i) \times \ldots \times f_1'(X) \tag{2.10}$$

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

where every $f_i'()$ is a Jacobian matrix, thus F' holds the Jacobian matrix for function F, and $X_i = f_{i-1}(X_{i-1}) \circ \cdots \circ f_1(X)$ for every $i \in \{1, p\}$ is the set of intermediate variables computed by the composition of the original functions, the composition goes from original functions $f_1()$ to $f_{i-1}()$. Thus, the function $f_i'()$ receives the updated set of variables as input.

Using the fact that the instructions of program P implement the elementary mathematical functions of F, we obtain the following expression for the program P'

which computes Jacobian F'.

$$P' = I'_{11} \; ; \ldots ; I'_{1r} \; ; \; I_1 \; ; \ldots ; I'_{j1} \; ; \ldots ; I'_{jr} \; ; I_j \; ; \ldots ; \; I'_{q1} \; ; \ldots ; I'_{qr} \tag{2.11}$$

where each sequence of instructions I'_{j*} with $j \in \{1, q\}$ implements $f'_i(X_i)$ with $i \in \{1, p\}$. Because $f'_i()$ is a Jacobian, each instruction I'_{jk} with $k \in \{1, r\}$ represents a partial derivative, where a partial derivatives is defined as the derivative of a multi variable function with respect one of its variables while the others held constant. For example, a function F is defined as $F(x_1, x_2) = sin(x_1 * x_2)$, with $f_1(x_1, x_2) = x_1 * x_2$ and $f_2(f_1(x_1, x_2)) = sin(f_1(x_1, x_2))$, the jacobian of F is:

$$F'(x1, x2) = f'_2(f_1(x_1, x_2)) \times f'_1(x_1, x_2) \tag{2.12}$$

We can define $X_0 = (x_1, x_2)$ and $X_1 = f_1(x_1, x_2)$, thus Formula 2.12 becomes:

$$F'(x1, x2) = f'_2(X_1) \times f'_1(X_0) = \frac{\partial f_2}{\partial X_1} \times (\frac{\partial f_1}{\partial x_1} \frac{\partial f_1}{\partial x_2}) \tag{2.13}$$

A program P that computes F may have the following instructions:

$$P = I_1 \; ; \; I_2 \tag{2.14}$$

where the instructions are defined as $I_1 = x_1 * x_2$ and $I_2 = sin(I_1)$. Thus, the program P' (differentiated version of P) that computes F' of Formula 2.13 has the following form:

$$P' = I'_{11} \; ; \; I'_{12} \; ; \; I'_{21} \; ; \; I'_{22} \tag{2.15}$$

where the instructions in Formula 2.15 are defined as follows:

$I_1 = x_1 * x_2$
$I'_{11} = \frac{\partial f_1}{\partial x_1} = x_2$
$I'_{12} = \frac{\partial f_1}{\partial x_2} = x_1$
$I'_{21} = \frac{\partial f_2}{\partial X_1} \frac{\partial f_1}{\partial x_1} = sin(I_1) * I'_{11}$
$I'_{22} = \frac{\partial f_2}{\partial X_1} \frac{\partial f_1}{\partial x_2} = sin(I_1) * I'_{12}$

In order to estimate the size of P' in terms of number of instructions, we propose that the maximum number of instructions of the program P' is n (dimension of the input space) times the number of instructions of the program P (potentially a partial derivative for every input for every $f_i()$) plus the numbers of instructions of P. This is because the instructions I'_j may use some of the original I_j. On the other hand, assuming that no original instruction is required, and assuming that only one instruction is required to implement the $f'_i()$, the minimal number of instructions needed to implement P' is the same as the original program P number of instructions. Clearly, the number of instructions varies depending on two aspects: the number of original

instructions necessary to compute the differentiated ones, and the way the inputs depend on the functions $f_i()$. This leads to the concept of Jacobian sparsity, which looks for the non-zero elements of the matrix.

The chain rule also can be applied to piecewise functions, but in that case the piecewise function has to be continuous and differentiable for the connection points, thus ensuring the minimal conditions for differentiation. Nonetheless, the derivatives computed around or for the connection points may be inconsistent, thus making the results unreliable. This problem is the motivation for the third chapter of this thesis.

2.3.2 Example

In order to simplify the expressions of Formula 2.7, we introduce the following functional replacements: $a(x_1, x_2) = x_1^2 + x_2^2,\ b(x_2) = (1 - x_2)$ which transforms Formula 2.7 into the following form:

$$F(a, b) = \frac{sin(a) \times b}{a} \tag{2.16}$$

Applying the chain rule to the example Formula 2.16, we obtain the following expressions for the partial derivatives:

$$\frac{\partial}{\partial x_1} F = 2 \times x_1 \times b \times (\frac{cos(a)}{a} - \frac{sin(a)}{a^2}) \tag{2.17}$$

$$\frac{\partial}{\partial x_2} F = \frac{2 \times x_2 \times b \times cos(a)}{a} - \frac{(a - 2 \times x_2 \times b) \times sin(a)}{a^2} \tag{2.18}$$

The formulas 2.17 and 2.18 are not factorized in a random way, the factorization takes into account the fact that $sin()$ and $cos()$ are expensive functions to evaluate from computational point of view, thus minimizing the repetition of those functions is valuable in execution time.

2.3.3 The Chain Rule on DAG

Figure 2.4 shows the original DAG (in Figure 2.4) augmented with the partial derivatives $c_{i,j}$ of every elementary function $f_i()$, which in a DAG are represented by the vertices. In the context DAG formalization the partial derivatives are defined as follows

$$c_{i,j} = \frac{d}{dv_i} v_j \tag{2.19}$$

These partial derivatives are placed along the edges just before every vertex destination, for instance the partial derivative of vertex v_j with respect to v_i ($c_{i,j}$) goes

along the edge $e_{i,j}$. It is possible to obtain a partial derivative for a vertex with respect to a vertex which could not necessarily be a direct predecessor, the only condition is that the vertices must be connected by a path of dependencies.

Original evaluation list	*Jacobian evaluation list*
$I_1 = x_1 \times x_1$	$I_8 = 2 \times x_1$
$I_2 = x_2 \times x_2$	$I_9 = 2 \times x_2$
$I_3 = 1 - x_2$	$I_{10} = cos(I_4)$
$I_4 = I_1 + I_2$	$I_{11} = I_{10} \times I_7$
$I_5 = sin(I_4)$	$I_{12} = I_7 / I_4$
$I_6 = I_5 \times I_3$	$I_{13} = I_{11} / I_5$
$I_7 = I_6 / I_4$	$I_{14} = I_{13} - I_{12}$
	$I_{15} = I_{14} \times I_8$
	$I_{16} = I_{11} - I_{12}$
	$I_{17} = I_{16} \times I_9$

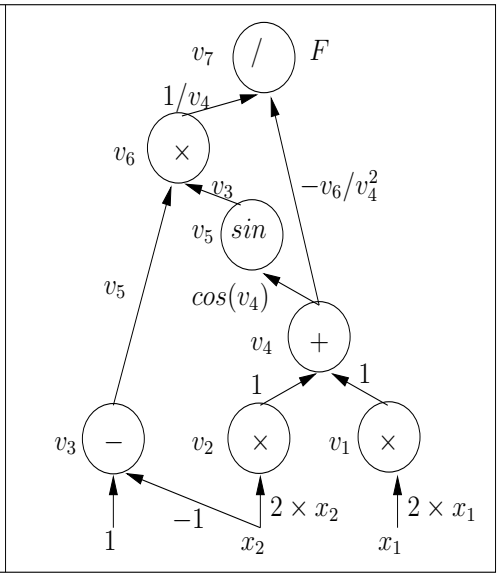


Figure 2.4: Vertices hold the elementary mathematical functions and along the edges the partial derivatives of the destination vertex

In order to obtain formulas 2.17 and 2.18 within DAG formalization, we have to apply the following rule for every input variable:

$$\frac{\partial}{\partial x_i} F = \sum_{l \in C} \prod_{i,j \in l} c_{i,j} \tag{2.20}$$

The right hand side (rhs) outer operation of Formula 2.20 sweeps over all the *dependency paths* C, a dependency path is a sequence of vertices, where these vertices satisfy the dependency relationship with their successors. every dependency path in C connects x_i to the last vertex of the DAG. The rhs inner operation multiplies the partial derivative of every vertex in the dependency path l, where $l \in C$. The latter generates a series of mathematical expressions which the former composed in the required partial derivative.

The computation of the Jacobian using the chain rule for an arbitrary function implemented by a program is expensive. In particular, it is computationally expensive due to the repetitive matrix times matrix operation. Nevertheless, the chain rule generates exact derivatives, and it is possible to be implemented and to be represented

in many ways.

The following section explores a specialization of the chain rule which is less expensive to compute, because it takes into account the directional characteristic of the derivatives.

2.4 The Tangent Mode of AD

2.4.1 Definition and AD Model

The tangent mode is the first specialization of the framework described in the previous section. The motivation for this model is the fact that the computation of the complete Jacobian is too expensive, and usually only certain directions in the input space are needed. Thus, the tangent mode produces code which implements the derivatives for certain directions in the input space. In order to produce the so-called tangent differentiated code, we have to modify the Formula 2.10, introducing the directional effect. We obtain the following form,

$$dY = F'(X) \times dX = f'_p(X_p) \times \cdots \times f'_i(X_i) \times \cdots \times f'_1(X) \times dX \tag{2.21}$$
$$F' \times dX : X, dX \in I\!R^n \rightarrow dY \in I\!R^m$$

where X represents the input variables, and dX a column-vector which holds a direction in the input space, both of dimension n; also dY is the output row-vector of variables of dimension m. In Formula 2.21 we have used the associativity property of matrix multiplication, thus the computation of the formula takes $O(n^2)$ instead of $O(n^3)$ by a classic linear algebra algorithm.

Formula 2.21 is related with the classical and general definition of directional derivative provided by the *Gâteaux derivative* [21]. But unlike Gâteaux derivative, we demand only that the function is differentiable for the given directions in the input space. Also, the derivatives generated with Formula 2.21 are obtained applying the chain rule. In contrast, Gâteaux derivative is computed as a limit, thus suffering all the already mentioned problems that make DD not a reliable way to obtain computational derivatives.

The AD model that implements the Formula 2.21 has the following form

$$P' = I'_1 \ ; \ I_1 \ ; \ldots ; \ I'_j \ ; \ I_j \ ; \ldots ; \ I'_q \tag{2.22}$$

In Formula 2.22 Jacobian times direction vector operations are represented by I'_j with $j \in \{1, q\}$, thus implementing the directional derivative of I_j, which follows a particular direction in the input space which is given as an input.

2.4.2 Example

Table 2.1 shows the tangent differentiated version of the program that implements the example Formula 2.7. The original evaluation list of instructions is given in the left part of Figure 2.3 as well as in the left part of Table 2.1, the differentiated evaluation list is given in the right part of Table 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`

Table 2.1: Automatic generated tangent differentiated code.

In Table 2.1 the tangent differentiated version of subroutine F (F_d), receives as input: variables $x1$ and $x2$, the direction on input space represented by $x1d$ and $x2d$. The output of the differentiated subroutine is the original output y and the direction on the output space yd, which represents the directional derivative for this subroutine. Instructions with suffix d implement the directional derivative of the corresponding instruction. Note that for the code presented in this work, the intrinsic functions ($sin()$, $cos()$) are denoted in capital letters, and the subroutines are stripped down to the header instructions, thus discarding variable declarations and unnecessary symbols.

The number of instructions of subroutine `F_d` is consistent with the estimation made in Section 2.3, and it is even more accurate if the instructions of the differentiated program are splited into instructions just holding an elementary function. Also, the differentiated program computes the original output y even if is not required to

compute any differentiated instruction. This feature is related with the convention within the AD community in order not to modify the original code, otherwise those instructions should not be computed. The original code is not modified in order to allow the user to recognize the code after the differentiation process.

The differentiated subroutine in right part of Table 2.1 implements the tangent differentiation of the example with respect to both input variables. Thus, in this particular example the tangent differentiated code implements the following mathematical expression:

$$dY = F'(X) \times dX = \frac{\partial}{\partial x_1} F \times dx_1 + \frac{\partial}{\partial x_2} F \times dx_2 \tag{2.23}$$

2.4.3 The Tangent Mode on DAG

Equation 2.23 can also be computed using the tangent mode within the DAG formalization. Formula 2.20 has to be modified introducing the directional characteristic, for instance for a scalar example we propose the following form,

$$dY = \sum_{i}^{n} \frac{\partial}{\partial x_i} F \times dx_i = \sum_{i}^{n} (\sum_{l \in C} \prod_{i,j \in l} c_{i,j}) \times dx_i \tag{2.24}$$

The complete Jacobian is computed by applying the tangent mode with respect to every input variable, and then evaluating the differentiated subroutine given every axis of the Cartesian system as dX.

If higher-order derivatives are required, the tangent mode can be applied repeatedly; but this procedure introduces practical complications, for instance, the notation becomes increasingly difficult to handle. Furthermore, the generated differentiated code becomes increasingly difficult to read and understand due to the long and complicated variable and subroutine names. Also, the brute-force procedure is not efficient, because some higher-order partial derivatives may not be required, usually what is needed is Hessian times vector. In order to compute the Hessian matrix it is preferred to study the pattern of sparsity of the Jacobian, before computing the partial derivatives.

The tangent mode of AD proves to be cheaper to compute than the complete jacobian for every program for certain directions in the input space. In the following section we present the second fundamental AD mode, the reverse mode, which was developed with the intention to provide an efficient way to produce gradients.

2.5 The Reverse Mode of AD

2.5.1 Definition and AD Model

The main goal of the reverse mode is to compute linear combinations of the columns of the Jacobian, a gradient. The gradients are important because they are used in several numerical methods, for instance in gradient based methods.

For scalar functions, the gradient is defined like the column vector whose components are the weighed partial derivatives of the function. The interest on the gradient lies on the information regarding the directions in the input space, which show the largest ratio of change in the output space. Therefore, we build a vector $\overline{Y}$ that defines the weights of each component of the original output $Y = F(X)$. The composition of this vector with the original output vector Y defines a scalar output,

$$\bar{Y}^t \times Y = Y^t \times \bar{Y} \tag{2.25}$$

replacing $Y = F(X)$ in 2.25 we obtain

$$\bar{Y}^t \times Y = F^t(X) \times \bar{Y} \tag{2.26}$$

and its gradient has the following form (using the chain rule):

$$\bar{X} = F'^t(X) \times \bar{Y} = f_1'^t(X) \times \ldots \times f_i'^t(X_i) \times \ldots \times f_p'^t(X_p) \times \bar{Y} \tag{2.27}$$
$$F' \times \bar{Y} : \bar{Y} \in I\!R^m, X \in I\!R^n \rightarrow \bar{X} \in I\!R^n$$

where every $f_j'^t()$ and $F'^t(X)$ are transposed Jacobians. X_i are computed in the same way as they are computed in the chain rule, that is through the evaluation of the function from the original input. However in Formula 2.27 the order of computation is inverted, thus the first function to be computed is $f_p'^t(X_p)$ the input of which is X_p, and recalling from Section 2.3, X_p is obtained after the computation of all original functions $f_i()$ except $f_p()$. Therefore, to compute the reverse mode, first we have to compute the original program, and in a second stage, we have to compute the Formula 2.27.

In Formula 2.27 as well as in the tangent mode (Formula 2.21) the first operation is matrix times vector, the result of which is propagated through the suite, thus, in principle the cost of reverse mode computation is similar to the tangent mode. Unfortunately, due to the fact that the reverse computation requires the original intermediate variable values, the original program needs to be computed. Furthermore, some price has to be paid for the computation of the transposed elements of the Jacobian. These facts diminish the efficiency of the reverse mode.

The AD model that implements the Formula 2.27 has the following form

$$\bar{P} = \overrightarrow{P}; \overleftarrow{P} = I_1 \; ; \ldots ; \; I_j \; ; \ldots ; \; I_{q-1} \; ; \; I'_q \; ; \ldots ; \; I'_j \; ; \ldots ; \; I'_1 \tag{2.28}$$

where the reverse differentiated program $\bar{P}$ is composed of two parts: the first one is called forward sweep $\overrightarrow{P}$, and is basically the original sequence of instructions. The second part is called backward sweep $\overleftarrow{P}$, and it is composed of the instructions that implement the transposed Jacobian times vector operations. Remarkably, the last instruction of the forward sweep is never required in the backward sweep. This is because the input set of variables for the first instruction of the reverse sweep is the output of the forward sweep minus the last instruction. Therefore whatever that last instruction computes is not required (except if the last instruction is defined as $I_n = \sqrt{(}I_{n-1})$, where the derivative depends on I_n).

2.5.2 Example

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

Table 2.2: Automatic generated reverse differentiated code.

In Table 2.2 the reverse differentiated version of subroutine F (F_b), receives as inputs variables $x1$ and $x2$, and the weight of the original output represented by yb.

The outputs are the elements of the gradient (in this case) $x1b$ and $x2b$. Instructions with suffix b implement the derivatives corresponding to the original instructions.

2.5.3 The Basic Problem of the Reverse Mode

The previous program example is not realistic with respect to the usage of the variables, usually in real codes the variables are re-defined, for example the original evaluation list of left part of Table 2.2 might be re-written in many ways as presented in upper section of Table 2.3.

	Version 1	**Version 2**	**Version 3**
f	`i1 = x1 * x1`	`CALL PUSH(x1)`	`CALL PUSH(x1)`
o	`i2 = i1`	`x1 = x1*x1`	`x1 = x1*x1 + x2*x2`
r	`i1 = x2 * x2`	`i1 = x2*x2`	`CALL PUSH(x2)`
w	`i2 = i2 + i1`	`i1 = x1 + i1`	`x2 = sin(x1)*(1-x2)`
a	`i1 = 1 - y`	`CALL PUSH(x2)`	
r	`i3 = sin(i2)`	`x2 = 1 - x2`	
d	`CALL PUSH(i1)`	`x1 = SIN(i1)`	
	`i1 = i3 * i1`	`CALL PUSH(x1)`	
		`x1 = x1*x2`	
b	`i1b = yb/i2`	`x1b = x1b + yb/i1`	`x2b = x2b + yb/x1`
a	`i2b = -(i1*yb/i2**2)`	`i1b = -(x1*yb/i1**2)`	`x1b = x1b -`
c	`CALL POP(i1)`	`CALL POP(x1)`	`  x2*yb/x1**2`
k	`i3b = i1*i1b`	`x2b = x2b + x1*x1b`	`CALL POP(x2)`
w	`i1b = i3*i1b`	`x1b = x2*x1b`	`x1b = x1b +`
a	`i2b = i2b+COS(i2)*i3b`	`i1b = i1b+COS(i1)*x1b`	`  (1-x2)*COS(x1)*x2b`
r	`x2b = x2b-i1b`	`CALL POP(x2)`	`x2b = 2*x2*x1b -`
d	`i1b = i2b`	`x2b = 2*x2*i1b - x2b`	`  SIN(x1)*x2b`
	`x2b = x2b + 2*x2*i1b`	`x1b = i1b`	`CALL POP(x1)`
	`i1b = i2b`	`CALL POP(x1)`	`x1b = 2*x1*x1b`
	`x1b = x1b + 2*x1*i1b`	`x1b = 2*x1*x1b`	

Table 2.3: Automatic generated reverse differentiated versions of example code.

Once some variables required in the reverse sweep are re-defined in the forward sweep, the values that they used to hold are not available for the reverse sweep. This reveals the hardest problem of the reverse mode of AD. The problem is how to make those vanished variables values available when they are required in the reverse sweep. There are mainly two strategies to cope with this problem: *Store-All* (SA) or *Recompute-All* (RA). The former strategy consists in storing all the values, during

the forward sweep, thus making them accessible in the reverse sweep. The latter strategy consists in re-computing every required (but lost) value in the reverse sweep. A trade-off is immediately apparent; SA consumes a great deal of memory, on the other hand RA increases the execution time. The best strategy lies somewhere between these two extremes. This discussion serves as introduction to the fourth chapter of this thesis, where we shall deal with that trade-off.

Table 2.3 shows some variations over the original example code, but also presents the reverse differentiated version of these variations. One way to have access to the original values of variables is to store them, as we described for SA strategy. For instance, a simple mechanism to store and restore the values in the correct order is the stack, when a variable is about to be re-defined, the value is stored in the stack. In the case of Table 2.3, we use a function named PUSH to perform the storing process. Conversely, when a derivative is about to be computed and the instruction that implements that derivative requires a vanished variable value, a function named POP is called to restore it from the stack. Thus, the various differentiated codes of Table 2.3 implement the reverse mode AD with the SA strategy.

2.5.4 The Reverse Mode on DAG

To compute the reverse mode using the DAG formalization, we have to modify the Formula 2.20 in order to introduce the fact that now we compute adjoints. Adjoints are computed with respect to the outputs, as follows:

$$\bar{v}_j = \frac{\partial y_i}{\partial v_j} \tag{2.29}$$

The systematic application of Formula 2.29 allows to obtain the adjoints for function F with respect $\bar{y}_i \in \bar{Y}$, which adjoints compose the gradient (in scalar case). One of the desirable characteristic of the DAG formalization is that once the DAG including the partial derivatives is obtained, it can be used to generate both forward and backward mode for the represented function, or in other words, it allows the chain rule to be easily exploited in two directions. Thus, using the partial derivatives in the DAG, we get the following formula for the backward propagation of the adjoints,

$$\bar{x}_i = (\sum_{l \in \overleftarrow{C}} \prod_{i,j \in l} c_{i,j}) \times \bar{y}_i \tag{2.30}$$

After the $\bar{y}_i$ are initialized with the weights for every original output, the Formula 2.30 sweeps over all the paths $l \in \overleftarrow{C}$, where $\overleftarrow{C}$ is the same set as C in Formula 2.20, but with the paths following the edges in inverse order. That change can be seen as the transposition of the Jacobian, which is the key operation within the reverse mode formalization.

2.5.5 Relationship between Tangent and Reverse Modes

Both modes (tangent and reverse) implement elements of the Jacobian matrix, but depending on the dimension of the input (n) and output (m) spaces, it is possible to advise which mode to apply to a given program. If $n > m$ and specially if $m = 1$ (gradient) the reverse mode is the choice, this is because after Jacobian transposition, the matrix times vector operation ($M_{n\times m} \times V_{m\times 1}$) will be cheaper than the original Jacobian times vector ($M_{m\times n} \times V_{n\times 1}$). On the other hand, but for the same reasons, if $n < m$ the mode to choose is the tangent.

Nevertheless, there is a relation between the modes, which can be used for validation purposes. We can use the following relationships to verify that reverse mode results are consistent with the tangent mode results. Starting from:

$$\bar{X} = \bar{Y} \times F'(X) \tag{2.31}$$

$$dY = F'(X) \times dX \tag{2.32}$$

and we want to define a simply relationship between vectors, thus we can replace the definition of $F'(X)$ in 2.31 on equation 2.32, finally obtaining

$$\bar{Y} \times dY = \bar{X} \times dX \tag{2.33}$$

The relationship in Formula 2.33 is useful at practical level. This is because it can be used as a test of consistency between the results obtained of the differentiation of a program with both fundamental modes of AD. In the next section we present the classical theoretical performances of those modes.

2.6 Theoretical Performances of AD Modes

The complexity of the AD modes can only be bound [27, 59]. This is because the performance of the modes depend on the particularities of the implementation. The bounds are build based on the original program complexity. This is reasonable because both modes includes large parts of the original program.

We denote the execution time required by the evaluation of a program P as *TIME(P)*, and the memory consumption required by the evaluation of a program P as *MEMORY(P)*. Both definitions are based on the requirements at instruction level. For instance, *TIME(P)* is the accumulation of the execution time required by the evaluation of every instruction in program P. The execution time of every instruction is computed as the cost of the operations involved, thus tables with operation costs are required. In order to compute the cost of a differentiated instruction we have to use tables like Table 2.4, but with every possible elementary function, also a table

Elemental function	Tangent	Reverse
$f = x \pm y$	$fd = xd \pm yd$	$yb = xb = fb$
$f = x \cdot y$	$fd = xd \cdot y + x \cdot dy$	$xb = y \cdot fb$ $yb = x \cdot fb$
$f = x/y$	$fd = \frac{y \cdot dx - x \cdot dy}{y^2}$	$xb = fb/y$ $yb = \frac{-(x \cdot fb)}{y^2}$

Table 2.4: Derivatives for elementary operations.

with the cost of every operation is needed.

The bounds of the tangent mode are:

$$TIME(P') \leq c_{tan} \times TIME(P) \tag{2.34}$$

$$MEMORY(P') \approx \times MEMORY(P) \tag{2.35}$$

In Formula 2.34, P' denotes the tangent differentiated program. The coefficient c_{tan} is computed based on the mentioned tables of costs. The number of operations per tangent differentiated instructions depends on the number of directions of derivation. Therefore, c_{tan} depends on this number as well. Practically, the number of directions of derivation is the main factor that determines the performance of the tangent mode.

Formula 2.34 shows that the memory consumption of the tangent mode is close to the original program memory consumption. This is because, although we now tangent codes use more memory than the original, the memory consumption of both program (differentiated and original) is small. Practically, the difference in memory consumption is negligible.

The bounds of the reverse mode are:

$$TIME(\overline{P}) \leq c_{rev} \times TIME(P) \tag{2.36}$$

$$MEMORY(\overline{P}) \leq t \times MEMORY(P) + tape \tag{2.37}$$

In Formula 2.34, $\overline{P}$ denotes the reverse differentiated program. The coefficient c_{rev} is computed based on the cost of the operations needed to computed the derivatives, but also the computation of c_{rev} includes the time required by the memory management inherent of the structure of reverse differentiated. This extra execution time comes from the store/restore process needed to supply the required intermediate variables values. If the reverse mode is used to compute gradients, for industrial size codes the value of c_{rev} lies between 5 and 8. This is because the number of inputs, as for the tangent mode, play a role. We can consider $c_{rev} \approx m \times c_{rev}$, where m is the dimension of the output domain of the function implemented by program P. Therefore,

when $m = 1$, the cost to obtain gradients only depends on the cost of evaluating program P times the stack accessing cost plus the extra cost of computing the derivatives.

If restore-all, i.e. recompute the intermediate variables values, is use by the reverse differentiation, the Formula 2.34 has to be extended to captures the impact of high number of the instructions re-execution.

If the program is reverse differentiated using store-all strategy, the bound of Formula 2.37 is composed of two elements. The first element depends on coefficient t. This is computed using tables of cost per operation. The second element is called *tape*. The *tape* is the stack used to store the intermediate variables values during the forward sweep. Due to this process, for large codes, the *tape* can grow to unacceptable size. Thus, the *tape* becomes the most important element in the memory consumption of reverse mode of AD.

The real life performance of AD modes is linked to the strategies and architectures of the AD tools. This is the matter of the next section.

2.7 Strategies and Architectures for AD tools

In this section we present the two classical strategies to implement the AD models. We also present the general architecture of a AD tool. We focus on a compiler-like architecture, but we also mention other architectures. At the end of this section we present some applications of AD.

2.7.1 AD Implementation Strategies

Overloading

Overloading consists in replacing each active variable, i.e. a variable used by a derivative, with a new variable which holds two values, the first value is the original variable value, and the second value holds the differential information. Also, each elementary operation is treated in the same way, thus internally replaced by a new operation, which holds the computed value and its derivative. This procedure is carried out at compiling time. This means that the changes in the original code are just the necessary to inform where to apply the replacements. This is usually done by changing the variables and operations types and names.

The advantages and drawbacks of the overload strategy can be summarized as follows:

- *Advantage*: The original program is barely changed, since everything is done at compile time.
- *Advantage*: The implementation of the differentiation techniques is done inside libraries or as a part of a compiler, thus it is hidden to the user, and more important, the differentiated code is totally independent of the it. Therefore, a change in this implementation does not affect the differentiated code [6].
- *Drawback*: The performance of the differentiated code is low. This is because it constantly builds and destroys pairs of values.
- *Drawback*: It is hard to implement the reverse mode , mainly because the lack of context information, which is crucial in order to produce efficient code.

Examples of overloading AD tools are:

- ADOL-C (C/C++) [28], ADOL-F (F77/90) [55]. These tools are able to generate tangent, reverse differentiated code and some kinds of higher-order derivatives.
- FADBAD (C++) [4]. This tool is able to generate tangent and reverse differentiated code.
- NAGWare (F95) [49]. The tools provide tangent and reverse differentiated code.

Source Transformation

Source transformation consists in generate a complete new program, which is composed of most of the original program instructions and data definitions, and the new variables, arrays, and data structures that will hold the derivatives and the derivatives instructions.

The advantages and drawbacks of the source transformation strategy can be summarized as follows:

- *Advantage*: The resulting differentiated code is simple and can be compiled into an efficient code [6].
- *Advantage*: Improvement on the analyses is possible due to the abundant context information. This information is gathered during the initial steps of the compiler-like process, and it is reflected in rich intermediate representations. As we present in the next section.
- *Drawback*: The complexity of implementing a compiler-like tool.

- *Drawback*: Changes in the original code force to re-generate the differentiated code.

Examples of source transformation AD tools are:

- ADIC (C) [10]. This tool generates first and second order derivatives.
- ADIFOR (F77/90) [9]. This tool computes directional derivatives, gradients, and the Jacobian taking advantage of the sparsity.
- OPENAD (F90, C/C++) [60]. It computes tangent and reverse differentiated code.
- TAMC, TAF (F77/90, C/C++) [53, 24]. These tools generate tangent and reverse differentiated code. To compute the reverse mode recompute-all strategy, i.e. re-compute the required intermediate variables values, is used.
- TAPENADE (F77/90 and C) [32]. It computes directional derivatives and gradients. In order to compute the reverse in efficient way this tools uses store-all strategy, i.e. store the intermediate variables values.

2.7.2 AD Tool Architectures

Source-to-Source Architecture

Currently, the great majority of the AD tools use transformation strategy. This is because the reasons presented in the previous section, and particularly being the reverse mode the most popular AD mode, source transformation strategy is able to compute it efficiently thanks to global analysis of the code.

In Figure 2.5 (next page) we can observe the general structure of the AD tools that use source transformation. The structure in Figure 2.5 is strongly based on the classical compiler structure [1, 46]. AD tools take advantage of this fact, specially at the level of the analyses over the code.

In our AD tool TAPENADE the differentiation process flows as follows. Firstly, the original source code is parsed by the language-specific front-end. During this phase, the front-end transform the code to a semantic equivalent simplified form, the intermediate representation. In TAPENADE the intermediate representation is called *imperative language* (IL). This first stage is complex because is closely related with compiler features of the tool [63].

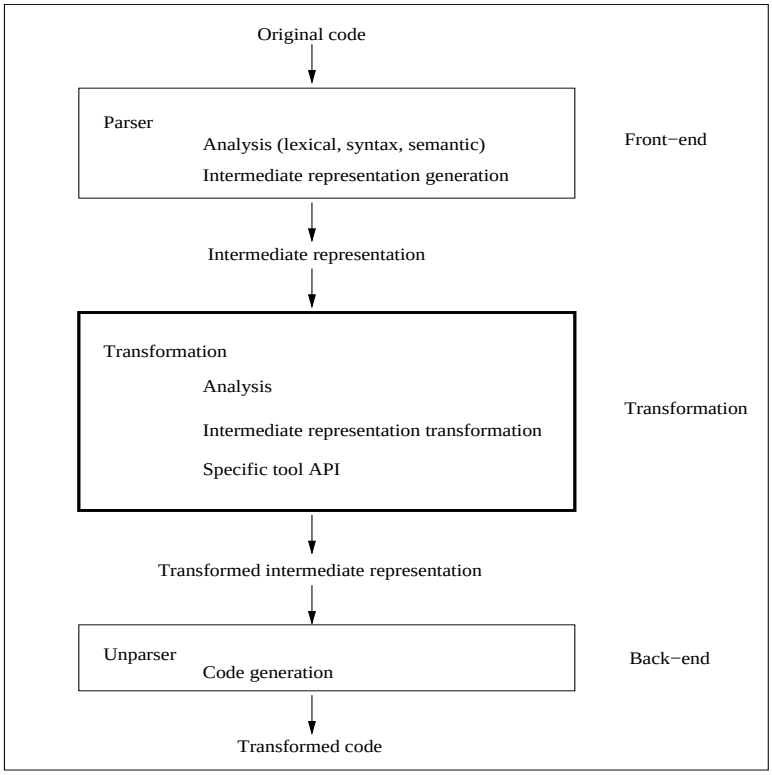


Figure 2.5: General view of AD tool structure. Compiler-like structure.

Secondly, the high-level information is gathered, such as the basic code elements, like basic blocks, expressions and subroutines. After that, the information is save in efficient structures for the analysis. In TAPENADE the structures are the flow-graph and call-graph. The analysis step includes all the necessary context analysis to execute the high-level transformation. Then the code transformation takes place.

Finally, the transformed intermediate representation code is unparsed (printed) to obtain the differentiated code.

In Figure 2.6 we can observe how TAPENADE internal structure follows the general structure of a compiler. Thus we can observe in the left-bottom area the parsers, in the center the analyses and differentiation, also the implementation of the users interface, and in the right-bottom the printers. TAPENADE is written in JAVA and some modules are written in C. it supports programs written in Fortran77 and Fortran90/95, is also under development to supports C.

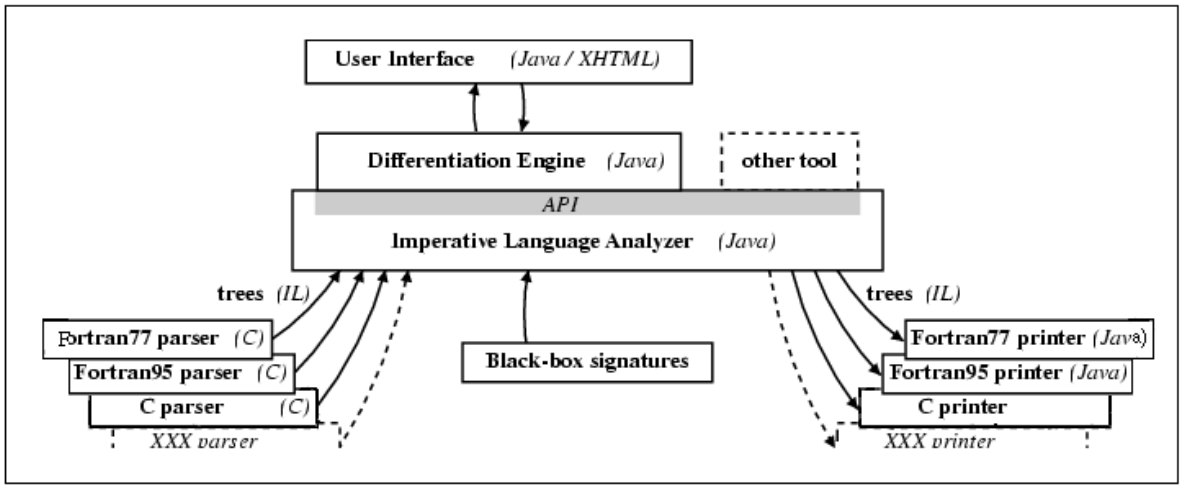


Figure 2.6: General view of AD tool TAPENADE structure.

Other Architectures

Other architectures exists. Among them let us mention:

- AD Libraries or ad-hoc libraries [44]. Basically special subroutines and types of the library are called from the original code, these subroutines provide the derivatives.
- Embedded AD in well known compilers [49]. So far, the most well known case of this architecture is the NAGWare compiler, which provides AD functionalities.
- Mathematical software (aka MATLAB) that includes AD [8, 52]. Extensions to mathematical software are mainly based on macros, and ad-hoc libraries written in the particular programming language of the mathematical software.
- AD inside general scientific frameworks [7, 56]. This last kind of architecture is similar to the previous one, but now the AD facility is built within a specific scientific framework.

2.8 Examples of AD Application

In this section we provide some examples of utilization of AD in scientific and engineering applications.

Domain	Id	Application	AD tool
Beam Physics	1	Simulation and Optimization of the Tevatron Accelerator [56].	COZY INFINITY
Biomedicine	2	Implementation of Automatic Differentiation Tools for Multicriteria IMRT Optimization [39].	ADOL-C
CFD	3	Reverse automatic differentiation for optimum design from adjoint state assembly to gradient computation [15].	TAPENADE
Engineering	4	Adjoint Differentiation of a Structural Dynamics Solver [58].	ADIFOR
Engineering	5	Streamlined Circuit Device Model Development with FREEDA and ADOL-C [30].	ADOL-C
Hydraulic	6	AD: A Tool for Variational Data Assimilation and Adjoint Sensitivity Analysis for Flood Modeling [12].	TAPENADE
Optimization	7	Application of Targeted AD to Large Dynamic Optimization [51].	TAMC
Robotics	8	Periodic Orbits of Hybrid Systems and Parameter Estimation via AD [52].	MATLAB ADMC++
Weather Forecast	9	Development of an Adjoint for a Complex Atmospheric Model, the ARPS [64].	TAF
Weather Forecast	10	Tangent Linear and Adjoint Versions of NASA/GMAO's Fortran 90 Global Weather Forecast Model [23]	TAF

Chapter 3

The Domain of Validity of Derivatives

Contributions of this chapter:

- We formalize a general approach to the validity problem for a general program. The approach consist of analyzing the variation of each conditional statement, and propagating the variation to the end of the program, where a valid sub-domain of the input space is built.

- We present a first specialization of the general approach. This is called HSPB and relies on the reverse mode of AD. Unfortunately, HSPB is unacceptable expensive. The computational cost comes from the size of problem, where the size of the problem depends on the number of constraints and the representation of the information to be propagated. We discuss alternatives to reduce this cost. As a results we present a trade-off between computational cost and accuracy of the representation.

- We present a second specialization of the general approach. This is called DFP and relies on the forward mode of AD. Unlike HSBP, DFP is computational cheap, but returns only partial information. We introduce an implementation based on DFP. The implementation can be seen as an extension to the AD tool TAPENADE.

- We discuss the possible utilization of this tool as an embedded part of a larger algorithm or method, thus the results of the tool become feedback information for such algorithm.

Automatic Differentiation (AD) tools assume differentiability of the function implemented by the given program. This assumption is fundamental, because the underlying mechanism of AD is the systematic application of the chain rule, which assumes differentiability of every component function. However, sometimes functions are composed by non-smooth elementary functions, which may lead to lose the global differentiability. Another source of wrong derivatives are switches in the control flow, mainly coming from conditional statements. These switches make most programs only piecewise differentiable. In these cases, sometimes the derivatives close to switches are inconsistent, because they are computed by different sets of instructions. Furthermore, differentiated programs may return derivatives where the implemented function is not differentiable, for instance [64]. Unfortunately, the everyday use of AD overlooks those problems, thus problems that should be essential become a matter of concern only when results are not what was expected.

Extended models of AD have been developed that return useful generalized derivatives for some classes of non-smooth functions [35]. However, there are no comprehensive studies, and to the best of our knowledge, no AD tool so far has incorporated any feature to cope with this kind of problematic functions.

From the users point of view, any kind of indication of possible problems with the derivatives is welcome, as Christianson stated explicitly "... it is useful that the AD-based model can signal automatically any potential catastrophic non-linearity ..." [13]. Without warnings, the users lost time and gain in frustration until they discover why the derivatives were incorrect. For example, Tadjouddine [58] after differentiating his code with ADIFOR [11], and discovering that the resulting derivatives were incorrect was forced to deal with non-differentiability by doing the extra effort of manually modifying the differentiated code. Furthermore, this activity may introduce extra errors to the derivatives, because it is highly error prone..

To address this problem, in this chapter we propose a general model to evaluate, along with the derivative, the size of the differentiable neighborhood of the current input where the returned derivatives do not suffer from non-differentiability. Therefore, unlike the approaches proposed by Griewank in chapter 11.2 and 11.3 of [27], our approach do not look for patch the problem by means of generalized differentiation or one-sided Taylor-wise expansions, what we look for with our approach is to inform the user about where it is safe to use derivatives computed by AD. In order to inform the user our AD model will provide a "safe neighborhood", which is a sub-domain from where the derivatives can be computed safely. This "safe neighborhood" is essential to use the derivatives consistently. We investigated several models to compute this neighborhood and study their complexity. With this model, the differentiated program computes, along with the usual derivatives, some extra runtime information about the "safeness" of the derivatives, without diminishing the computing efficiency

of the differentiated code. Finally, we present an implementation and experiments made with our AD tool TAPENADE.

In the following section, we introduce the validity problem, especially focusing on the control flow switches, but not discarding problems inherent to the implementation of fundamental mathematical functions by programming languages. The rest of the chapter is organized as follows. In Section 3.2 we present the in-deep analysis of the validity problem. In Section 3.3 we define the the validity information and its propagation. In Section 3.4 we propose the first method to cope with the validity problem. In Section 3.5 we propose the second method. In Section 3.6 we present the experimental results. In Section 3.7 we present related techniques. Finally in Section 3.8 we discuss the results and give some prospects about the proposed methods.

3.1 The Validity Problem

Providing precise and reliable derivatives are among the goals of AD community, thus AD tools should provide valid derivatives within the input domain of the applications. Unfortunately, current AD models do not include verification of the differentiability of the functions. Furthermore, experience shows that the differentiability of functions implemented by programs can be rather easily corrupted, and to worsen the problem, the means of this corruption are usually overlooked, particularly at the development stage.

We have identified two main sources of non-differentiability in programs: first, the use of intrinsic functions of a programming language that are not differentiable for their current inputs. Second, the changes in the control flow that break the differentiability of the functions.

The first source of problems is due to functions like: $abs()$, $min()$, $max(,)$, $sqrt()$. These functions are non-differentiable or exhibit discontinuities for certain inputs in their domains. If functions of this kind are composed with smooth functions, the composition is at risk of being non-smooth. The chain rule can be nevertheless applied in that condition, but the results might be wrong.

The second source of problems, which is more common and overlooked, is introduced by conditional statements (*test*). The tests are part of the control flow structure of the original program, and this structure is preserved in the differentiated version of the original program. If the input values vary, even by a very small amount, this may make some tests branch differently (switch). Consequently, the actual statements executed change, and therefore the sequence of derivative statements changes accordingly. The implementation and differentiation of piecewise functions is prime

candidate for producing wrong derivatives, because piecewise functions are implemented using tests, and those test may switch depending on small variation of the inputs.

A couple of particular cases are worth to be mentioned. First, the function $1/x$, if the function input is small but non zero, then the result is correct. In this case, the problem arises when the derivative of the function is evaluated for a small enough input, in that case the result can be a infinite or not a number (depending on the computing platform), which is neither correct nor useful. Second, implementation of function by using look-up tables and interpolation mechanism. Clearly non-differentiable case, so far the best way to get derivatives in this case is to add another look-up table with pre-computed derivatives. This case is neither address in this work nor in the literature of AD.

To illustrate the validity problem and consequences, we introduce the programs on which we carried out our experiments. The first code is a modified version of the example Formula 2.7 given in Section 2.2.2. The second program is a implementation of the Newton method to find roots and to find the local minimum/maximum. The particularity is that in this case the function on which the method is applied is defined piecewise. Finally, we experiment with real-life scientific programs; those programs belong to the pool of program given by scientific partners of our research team.

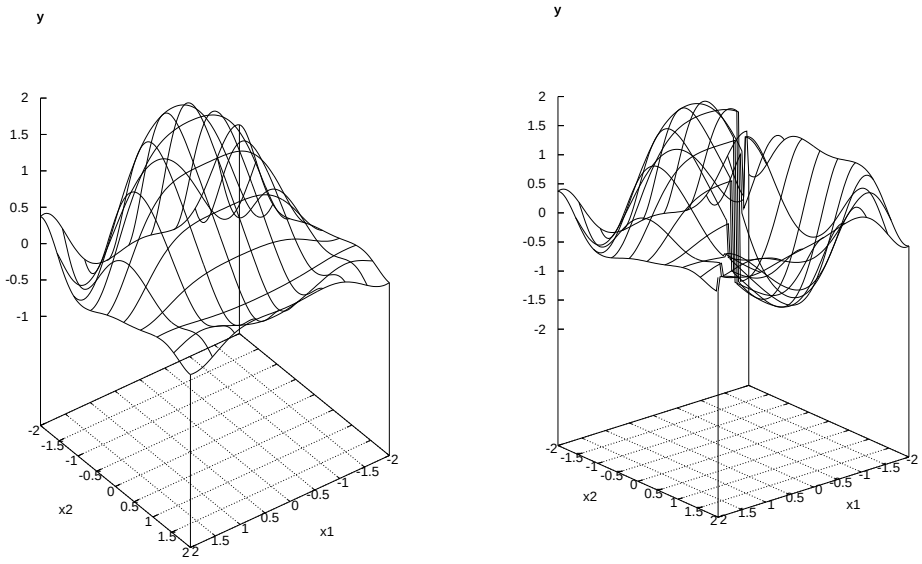
As we mentioned above, in order to illustrate the validity problem, we present the following example. We modify the example problem given in Section 2.2.2, introducing a test which shall reveal the effects and consequences of the conditional statements over the derivatives. The original example Function 2.7 has a good behavior in terms of continuity and differentiability in its domain. The output of Function 2.7, with input $X \in [-2, 2][-2, 2]$, is represented by the left part of Figure 3.1.

In order to show the effects on the derivatives of switches in the control flow, we modify the Formula 2.7 as follows:

$$Y = F(x_1, x_2) = \begin{cases} \frac{sin(x_1^2+x_2^2)\times(1-x_2)}{x_1^2+x_2^2} & \text{if } x_2 < x_1 \\ \frac{-1\times sin(x_1^2+x_2^2)\times(1-x_1)}{x_1^2+x_2^2} & \text{if } x_2 >= x_1 \end{cases} \tag{3.1}$$

$$F : X \in I\!R^2 \rightarrow Y \in I\!R$$

where the second expression is minus one time the first expression, also $(1 - x_2)$ of the first expression is replaced by $(1 - x_1)$ in the second expression. Both changes are taken into account at implementation level. We can observe in the left sector of Table 3.1, the implementation of the above changes by instruction `i5` within the test `T1`. After the differentiation of the piecewise example, we obtain a differentiated



original example **piecewise example**

Figure 3.1: Plot of the example and the modified example around a given input.

code, which is presented in the right sector of Table 3.1.

The differentiation is applied regardless of the differentiability of the component functions, thus overlooking any local differentiability problem. The control flow structure is preserved (test `T1`), and we obtain two derivative statements for `i5` corresponding to every branch of the test, therefore the example is only piecewise differentiable.

Unfortunately, the differentiated code is generated without taking the possible inconsistent results into account, and it returns a derivative `yd` even when the input variables are $|x1| \approx |x2|$, which leads to two different sets of derivative statements. As a result, for a slightly different inputs two different derivatives are returned.

In Figure 3.2 the behavior of the original differentiated and piecewise differentiated is compared, for instance, in the original differentiated example the derivative for points $p1 = (x1, x2) = (0.100001611, -0.0999996886)$ and $p2 = (x1, x2) = (-0.0999983624, 0.100000314)$ is $yd = -0.999933362$, but in the piecewise case, the

Modified Example 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 = i1d + i2d`
`i3 = i1 + i2`	`i3 = i1 + i2`
	`i4d = i3d * COS(i3)`
`i4 = SIN(i3)`	`i4 = SIN(i3)`
`T1 IF ( x2 .LT. x1 ) THEN`	`T1 IF (x2 .LT. x1) THEN`
	`i5d = -x2d`
`i5 = 1 - x2`	`i5 = 1 - x2`
`ELSE`	`ELSE`
	`i5d = -x1d`
`i5 = 1 - x1`	`i5 = 1 - x1`
	`i5d = -i5d`
`i5 = -i5`	`i5 = -1 * i5`
`END IF`	`END IF`
	`i6d = i4d * i5 + i4 * i5d`
`i6 = i4 * i5`	`i6 = i4 * i5`
	`yd = (i6d * i3-i6 * i3d) / i3**2`
`y = i6 / i3`	`y = i6 / i3`

Table 3.1: Automatic generated tangent differentiated modified code.

derivative for $p1$ is $yd = -0.999933362$ and for $p2$ is $yd = 0.999933362$.

A minor modification in the function definition, and subsequently in the implementation, may introduce discontinuities, therefore the differentiated piecewise function may return different derivatives in a neighborhood of a critical point within the domain. If this fact is not taken into account, the resulting derivatives may be useless in certain domains. The numerical results of the example show that a very careful use of these result derivatives is mandatory. Otherwise, if the rest of the code is not adapted to those changes in the derivatives depending on certain input, the general results could be spoiled.

What happened in this example may happen systematically within large codes. In codes which contain hundreds of conditional statements, even if only one of those conditional statements affects the derivatives, by propagation the whole program re-

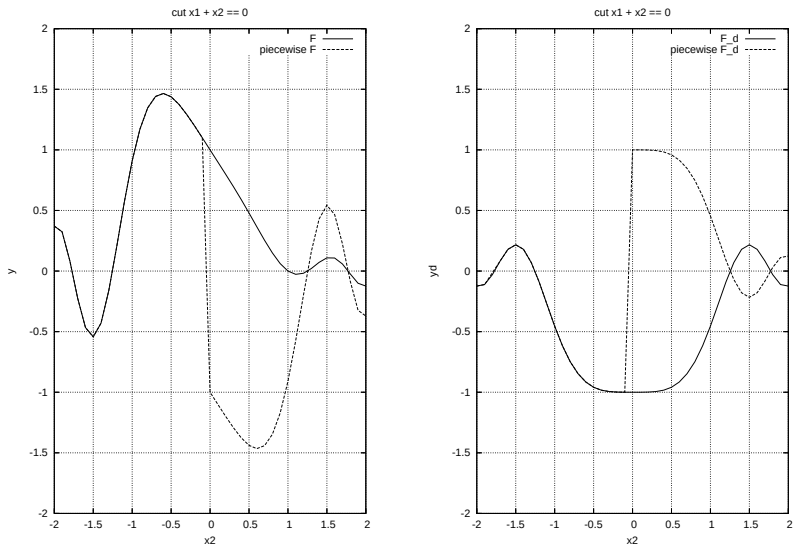


Figure 3.2: Plot of directional derivatives with input space direction $(x1d, x2d) = (1, 1)$.

turning derivatives may become unreliable.

The validity problem is discussed in the next section in detail. There we mainly include the description of the two sources of the problem, and the existing reduction procedure which leads to only one fundamental problem.

3.2 Analysis of the Validity Problem

3.2.1 The Differentiability of Intrinsic Functions

Some of the intrinsic functions of the programming languages hide discontinuities and non-differentiability. A short and open list of dangerous intrinsic functions is: $abs(x) = |x|$, $sqrt(x) = \sqrt{x}$, $|x, y| = \sqrt{x^2 + y^2}$, max(x,y), min(x,y), sign(x), heav(x).

These functions are commonly used, because they are provided by the programming language, but their implementations and derivative definitions require special attention. This comes from the fact that all of them include discontinuities, or that their derivatives are well known for their problems of differentiability.

One way to deal with those problematic functions is changing their implemen-

Intrinsic	Tangent Differentiated Code

C abs()
  i1 = ABS(x1)
	C derivative of abs()
	  IF (x1 .GE. 0.) THEN
	  i1d = x1d
	  i1 = x1
	  ELSE
	  i1d = -x1d
	  i1 = -x1
	  END IF

C sqrt()
  i1 = SQRT(i1)
	C derivative of sqrt()
	  IF (i1 .EQ. 0.0) THEN
	  i1d = 0.0
	  ELSE
	  i1d = i1d/(2.0*SQRT(i1))
	  END IF
	  i1 = SQRT(i1)

C max(,)
  i2 = MAX(i1, x1)
	C derivative of max(,)
	  IF (i1 .LT. x1) THEN
	  i2_d = x1d
	  i2 = x1
	  ELSE
	  i2_d = i1d
	  i2 = i1
	  END IF

Table 3.2: Function replacements and its derivatives for intrinsics in AD.

tation but not their mathematical meaning. As a result the implementation of the function can be modified from the intrinsic form (originally implemented by libraries of the programming language) to a *conditional* form. The latter is called the conditional form because in most cases the new implementation includes conditional statements to overcome the inherent problems of the functions, thus the point(s) of non-differentiability is(are) taken into account.

The previous mechanism allows to implement derivatives for those problematic differential functions, and it preserves the original computation results without loss of efficiency. Also, the mechanism can be nested, thus allowing all possible combinations of those functions. Table 3.2 shows the implementation of some of the replacement functions of the problematic functions, along with their derivatives.

The idea of replacing one implementation of a function by another to cope with non-differentiability was proposed in [26]. In that work a formalization that allows to derivate the Euclidean norm is also presented. This function is not an intrinsic function in Fortran 77 standard, but is widely implemented in ways that eventually lead to wrong derivatives. Tools that implement this idea are ADIFOR [11] and TAPENADE.

Once the above mechanism is applied, the first source of differentiability problems is transformed into the second source of problems, because all the replacements include conditional statements. Therefore, we claim that the second source of problems is the only fundamental one, and it is the problem that we shall study, derivatives computed through changes in the control flow.

3.2.2 Differentiability of Conditional Statements

Sometimes the derivatives depend on the tests. In these cases we have different derivatives depending on the switch of the test. If the test has the form "$if \ldots then \ldots else$" then we have two sets of derivatives instructions, each one corresponding to the branches of the test. recall the notation of programs: programs are composed of blocks of instructions B_i and tests T_i. After differentiation, we obtain blocks B'_i, which are the differentiated blocks corresponding to B_i, which contain the original instructions plus the derivatives instructions. The conditional statements remain the same, that is, they are represented by T_i in the differentiated code.

For example, in Figure 3.3 the control flow will follow instructions B'_3 or B'_4 depending on the sign of the test.

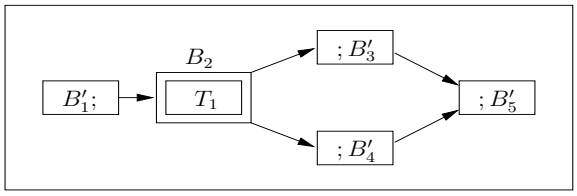


Figure 3.3: Differentiated flow-graph for example.

The problem arises when for some input, the program evaluates the sequence of blocks $B'_1; T_1; B'_3; B'_5$, and for another slightly different input value, the program evaluates $B'_1; T_1; B'_4; B'_5$. The difference between the first and second input value may be very small, but enough to switch the test T_1. The derivative instructions within B'_3 and B'_4 may be totally different. Thus, small changes in the input values may switch the test returning completely different derivatives. Note that there is a special

case when the instructions in both branches are identical. This case is not studied here because the returned derivatives should be consistent, this is due to the lack of consequences of the test switching.

Nested forms of conditional statements introduce more execution paths or sets of derivative statements. Thus, highly branching forms of tests may lead to more inconsistent derivatives. For instance for a simple two branches test we may obtain two sets of derivatives depending on the inputs, for a three branches test we may obtain three sets of derivatives and so on. From the point of view of the analysis, these nested forms however do not introduce more complexity into the problem itself. This is because the analysis is focus on the switch of one test at time.

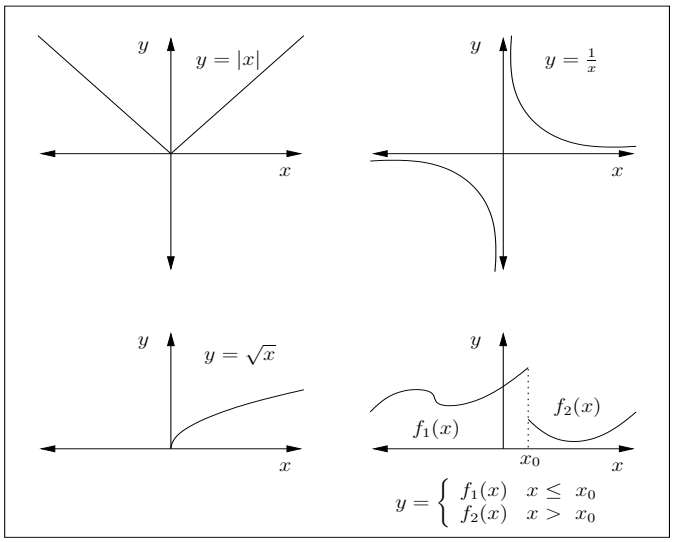


Figure 3.4: Examples of problematic functions.

In general, we can generate all forms of conditional statements based on the minimal set of forms presented in Figure 3.4. In this figure the bottom right figure represents a piecewise function, and the other forms may be changed into something similar to the latter. The ramification can grow arbitrarily, and branches may have branches by themselves, therefore the returned sets of derivatives can be counted as the execution paths of a flow-graph.

In the previous discussion we reviewed the structure of a test, but in real life codes, the structure of the tests are in average (from our experience) two levels deep with a maximum of three branches per each branch of the level 0, thus nine different execution paths. The propagation of the information is the key factor in this problem, due to the fact that the real life codes may have hundreds of tests, and a good deal

of them are related to the derivatives

It is important to remark the fact that the problem of inconsistent derivatives is exposed by repetitive execution of differentiated code, which returns different derivatives for a rather similar input. For a particular input value, the execution follows a path of instructions. But in a second execution of the differentiated code, with a similar input value, the path followed changes because of a test switch. As a result the returned derivatives are different or radically different of what was obtained in the first execution. The problem is inherent to the runtime behavior of programs, therefore it is not possible to foresee this problem, as it depends on the given input for each execution.

We believe that the information needed to ensure the validity of the derivatives as well as help to overcome the validity problem might be used to improve some algorithms, for instance algorithms which rely on a criterion which depends on derivatives to lead the iterative process.

In the following section, we present our approach to the validity problem of derivatives. This approach aims at introducing a new model of AD which does not change the structure of the program, yet adds valuable information about the derivatives at execution time without diminishing the performance.

3.3 Defining and Propagating Validity Information

In this section, we propose a new AD model which includes a method to validate the derivatives. Our approach does not try to establish differentiability close a test, but rather to accept that any test may cause non-differentiability. Therefore, the approach consists on the evaluation of how close the tests are from their switching values.

To validate the derivatives, we evaluate an interval in a neighborhood of the input data where no non-differentiability problem arises. Practically, this requires analyzing each conditional statement at run-time, in order to find for which data it will switch, and propagate this information as a constraint on the input data. We also discuss the complexity of this mode and some alternatives. Finally, we develop a mode that studies the validity of the derivatives; the implementation of that method introduces an extension of the tangent mode of AD.

We devise a method that returns a certain interval of solutions where the derivatives are not compromised by conditional switching. To do that, we develop a formalization that relates the tests, the input values and the variations of the input values.

Our idea is to evaluate the largest interval within a neighborhood of the current input data, so that there is no differentiability problem if the input remains in this interval. In the case when this interval is notably too small, this will be a warning to the user against an invalid use of these derivatives.

3.3.1 Validity Information Definition

As we introduced in the previous section, programs can be seen as a concatenation of blocks B_i and tests T_i. The notation for tests is extended to $T_i(X_i)$, emphasizing that the tests are built up from the intermediate variables X_i within the block before (B_i) the test. Therefore, an arbitrary program P has the following form:

$$P = (B_1(X), X_1)\ ;\ T_1(X_1)\ ; \ldots ;\ (B_n(X_{n-1}), X_n)\ ;\ T_n(X_n)\ ;\ (B_{n+1}(X_n), Y) \quad (3.2)$$

where the pair $(B_i(X_{i-1}), X_i)$ describes the input (X_{i-1}) for block B_i, and its output is the updated set of intermediate variables X_i.

Once AD is applied to program P, the new augmented program P' includes the derivatives instructions and the original control structure of the program, in this case the structure given in Formula 3.2.

$$P' = (B_1'(X, dX), X_1, dX_1);\ T_1(X_1)\ ; \ldots$$

$$\ldots ;\ (B_n'(X_{n-1}, dX_{n-1}), X_n, dX_n)\ ;\ T_n(X_n)\ ;\ (B_{n+1}'(X_n, dX_n), Y, dY) \quad (3.3)$$

where the expression $(B_i'(X_{i-1}, dX_{i-1}), X_i, dX_i)$ has as input the intermediate variables X_{i-1}, dX_{i-1} before block B_i, and as output the updated variables X_i, dX_i. The expression describes the differential relation between dX_{i-1} and dX_i through the following first order approximation,

$$dX_i = J(B_i(X_i)) \times dX_{i-1} \quad (3.4)$$

Also, Formula 3.4 shows the propagation of intermediate variables X_i through the evaluation of the jacobian of block $B_i()$.

In order to describe how the propagation of the intermediate variables are related to test switches, let us consider a test $T_i(X_i)$ in isolation. It uses only variables from X_i, and we can admit without loss of generality, that T_i is positive, that is

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

The analysis we developed looks for how much the intermediate variables X_i can change without switching the test, where that change is represented by dX_i. Adding

to the Formula 3.5 the variation dX_i, we obtain:

$$T_i(X_i + dX_i) \geq 0 \tag{3.6}$$

Developing Formula 3.6 as an approximation of first order, we obtain:

$$\begin{aligned} T_i(X_i) + &< T_i'(X_i), dX_i > \geq 0 \\ &< T_i'(X_i), dX_i > \geq -T_i(X_i) \end{aligned} \tag{3.7}$$

where the operator $<,>$ is the dot product.

Recursive development of the Formula 3.4 describe how dX_i depends differentially (at first order) on the input X and dX:

$$dX_i = J(B_i(X_i)\ ; \ldots ;\ B_1(X)) \times dX = J(B_i(X_i)) \times \ldots \times J(B_1(X)) \times dX \tag{3.8}$$

Using Formula 3.7 and Formula 3.8, we can state that the constraint on dX upon which the test T_i does not switch is:

$$< J(T_i(X_i)),\ J(B_i(X_i) \times \ldots \times J(B_1(X)) \times dX > \geq\ -T_i(X_i) \tag{3.9}$$

where the derivative of test T_i is expressed as $T_i'(X_i) = J(T_i(X_i))$, this is possible due to the fact that a test is implemented as an instruction.

Formula 3.9 is a constraint on dX, which represents a half-space in the input space. If the inputs remain in the half-space that satisfies the constraint, the variation dX is valid. Furthermore, due to the fact that dX is piecewise linear on the convex polyhedron, and the input X is in the interior of the valid domain, the computed derivatives are Fréchet derivatives [18, 19]. In computational terms, for a given X in the half-space that satisfy the constraints, the program follows the same execution path, or in other words remains in the same branch of the test, therefore the variations dX are consistent.

3.3.2 Propagating Validity Information

For the entire program, the computed derivatives will be valid if the variation dX of the input X satisfies all the constraints (Formula 3.9) for each test T_i. This gives a set of constraints on dX, or a set of half-spaces. The intersection of the half-spaces composes the neighborhood of the input values that returns valid derivatives; this neighborhood is what we call the *validity information*.

To compute the validity information for the entire program, systematically every test is analyzed; once the validity information is computed for the test, the information has to be combined with the information of previous tests, and so on to the end of the

program. This mechanism may be implemented in several ways. In the next section, we shall discuss a first approach which is based on the reverse mode of AD.

3.4 Half-Spaces Backward Propagation (HSBP)

3.4.1 Definition

To implement the previous method, we need to compute several jacobians, but the computation of complete jacobians is expensive, therefore we have investigated cheaper ways to implement the method. For instance, it is possible to modify Formula 3.9 in order to allow the computation of jacobians using the reverse mode of AD. Alternatively, the left side of the dot product in the left hand side (lhs) of Formula 3.9 can be computed using the forward mode of AD. Recall that from what we introduced in 2.6, the computational costs of the forward mode is proportional to the dimension of the input space.

Observing Formula 3.9, and recalling that we must solve it for dX. This means we must isolate dX. A powerful way to do that is to transpose the jacobians in the dot product, yielding the equivalent equation:

$$dX^t \times J(B_1(X))^t \times \ldots \times J(B_i(X_i))^t \times J(T_i(X_i)) \geq -T_i(X_i) \tag{3.10}$$

$$dX_i^t \times J(T_i(X_i)) \geq -T_i(X_i) \tag{3.11}$$

where in Formula 3.11, the computation of the differential relationship between dX and dX_i through X (Formula 3.8) is carried out backwards. This is as follows:

$$dX_i^t = (J(B_i(X_i)\ ;\ldots;\ B_1(X)) \times dX)^t = dX^t \times J(B_1(X))^t \times \ldots \times J(B_i(X_i))^t \tag{3.12}$$

where backward means that the first jacobian to be computed is the jacobian of the block just before the test. The expression $J(B_1(X))^t \times \ldots \times J(B_i(X_i))^t \times J(T_i(X_i))$ in Formula 3.10 is directly computed by the reverse mode of AD.

In this approach the constraint on dX is computed in a efficient way regarding execution time, also, the solution space is an exact (first order) representation, a half-space, therefore once all constraints are combined the global space of solutions shall be precise. Unfortunately, the reverse mode is expensive in terms of memory consumption, because it requires to store a large number of intermediate variables. Therefore, before implementing the model, we shall discuss alternatives to reduce the computations of Formula 3.10.

3.4.2 Problems with HSBP

We consider the model of Section 3.4.1 complete in the sense that it returns one constraint on dX for each test encountered during the execution of the program. However, in real situations, the number of tests is so large (A solver named STICS has 27.000 lines of code and around 540 tests that must to be analyzed) that this complete model is not practical, because the analysis of every test leads to compute Formula 3.10, which is demanding in memory space. This section investigates strategies to reduce the cost of this model.

Size and Representation

There are mainly two problems related to the cost of the model. The first is to compute and to propagate the validity information for a possible large set of constraints. This is expensive due to the computation of the validity information for every constraint. Thus, we aim at reducing the number of constraints.

The Second problem is the manipulation of the validity information representation might become a complex task. So far, we manipulated the representation of every analyzed test as a half-space in concordance with Formula 3.10. However, this can be a source of problems considering the dimension of the input space, and considering the dimension of the constraints themselves. For instance, the intersection of two half-spaces is not a half-space in general. Thus, we shall present alternative representations, which leads to a trade-off between the accuracy of the representation and the memory space needed to store that representation.

Reducing the Size of the Set of Constraints

The size of the set of constraints comes from the number of constraints and dimension of the inputs. Therefore, we face two alternatives: to drop certain constraints, or/and to select certain directions of derivation. The former strategy is discussed in this section, the latter strategy will be developed in Section 3.5. There are two ways to attempt the constraint elimination process, one is automatic and the other is a user driven activity.

We proposed a method to automatically drop some constraints, based on the fact that some of them may be redundant. To detect the redundant constraints, we calculated an index of relevance of constraints. The index was calculated using a measure of distance, where the distance is calculated as the separation between the constraint and the space of solution already computed. Consequently, we eliminated the useless ones, or in other words, we discarded the constraints which were far from the solution space. That criterion is founded on our goal, which is to provide a conservative neighborhood within the input space where the derivatives are valid. This method is

highly inspired by the analytic center cutting-plane methods (ACCPM) [61, 54].

The proposed method is precise in the sense that it drops constraints, which may not add solutions to the already computed solution space; thus the method may discard a great deal of constraints, and at the same time, returns a reliable solution space.

In the general case, the drawback with the ACCPM method is to determine the optimal point x^*, which initializes every iteration of the method. In fact, to determine that point can be as expensive as to calculate the index of relevance (ranking) of each constraint of the system. Our proposed method to drop constraints has an advantage over the general ACCPM, because the user gives a particular input around which we computed the safe neighborhood. We use that input as the initial optimal point, thus saving costly computation. However, we recognize that the given point has few chances of being optimal.

The proposed method was abandoned, after being tested on examples, due to the cost that it represents to compute the ranking. The computation of that ranking includes dealing with high-order derivatives, thus the cost of applying the method for large set of constraints is unacceptable. Nonetheless, the method is interesting, and the open problem of improving the ranking computation remains to be solved.

The second option to reduce the number of constraints is to let the user select which test must be analyzed. The idea is that the choice is given to the user considering the users knowledge about the meaning of the code. Therefore the user is considered to be well aware of the meaning and consequences of every test in the code; thus, the user can select the tests which have most influence over the derivatives. From an implementation point of view, the user may use programming directives in the code to indicate which test must be analyzed.

Unfortunately, the previous tactic is impracticable on large codes for several reasons, for instance, large codes are developed by teams, thus the difference between the global and local knowledge of the code may mislead the selection of the relevant constraints. Moreover, even if the manual selection of constraints is possible on some large codes, we believe that the number of selected constraints will be large, thus not helping enough the global goal of reducing the size of set of constraints.

Changing the Constraint Representation

In the previous discussion, two strategies were presented in order to reduce the size of the set of constraints; both strategies produced an exact (first order approximation) representation of the solution space, that representation is a polyhedron where the component faces are the constraints. In this section we propose to change that

representation in order to save computational costs. However once the representation is changed it becomes an coarse approximation. Therefore, we may lose some valid solutions which belong to the exact representation of the solution space.

The solution space of the constraint system can be represented in several ways, and different representations have different computational costs. Also, the representations have different degree of accuracy, so we have a trade-off between computational cost and accuracy (Table 3.3), as we discuss next.

Representation	Computational Cost	Accuracy
spherical	low	very low
hyper-rectangular	acceptable	low
polyhedral	high	good

Table 3.3: Representation of solution space, trade-off.

It is possible to represent the constraints using spheres, where the spheres are centered in the given input. The *spherical* representation is the cheapest representation in memory space, because we only need to store a point and a radius. Unfortunately the spherical representation is very inaccurate, because if the radius is set as the distance between the given input and the half-space, at the end program evaluation, the radius may be reduced to the distance between the given input and the closest half-space. Therefore, the represented solution space may be very small in comparison to the exact solution space, it may seem as an inscribed sphere within the polyhedron, where in the worst case scenario shows a sphere tangent to only one half-space, as in Figure 3.5.

The next two representations are more accurate because they fit better with geometry of intersected half spaces. The first one is so-called *hyper-rectangular*. To store a hyper-rectangle (an arbitrary dimensional rectangle) we just need to store two points for each dimension. In this case, the lost solution space is located between the inscribed hyper-rectangle and the polyhedron. A common situation which relates the representations is shown in Figure 3.5.

In Figure 3.5 the bold black lines denote the exact solution space, which we call the polyhedrical representation, and the simple lines represent the constraint half-spaces. The hyper-rectangle (in this case just a rectangle) is inscribed in the exact solution space, as well as the sphere (in this case a circle). The circles centre (the black point) is the given input.

The polyhedrical representation was the default representation assumed until this

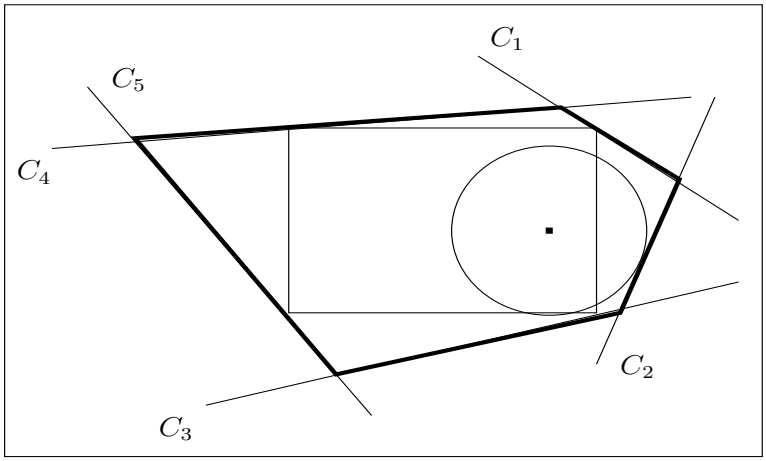


Figure 3.5: Representations for an arbitrary example.

point. It requires to store several vertices and edges, thus it is very expensive from a computational point of view, particularly expensive in memory consumption.

After reviewing several strategies and tactics developed to reduce the computational costs of the HSBP method, we estimate that the method -even for reduced cases- is not practical, because the costs are too high. In the next section, we present a method which takes advantage of a simple decision from the user that has a great impact, which is the selection of certain directions in the input space.

3.5 Directional Forward Propagation (DFP)

The size of the set of constraints remains as a big concern, but among the ways to reduce the problem size there is one which leads to DFP. That is, is possible to select certain directions of derivation, or even better to let the user identify and select certain directions of the input. This allows to simplify the constraints, and to focus on the relevant domain of validity. Besides, we can use the forward mode of AD because the needed derivatives will be directional ones.

3.5.1 Definition

The model we presented in Section 3.4 is expensive in memory consumption and execution time. To counter this, here we propose a strategy which is focused on directional derivatives. The goal is to give the user information about the validity of the derivatives regarding specific directions in the input space. This strategy returns an exact solution space, but is restrained in a certain spatial direction.

The idea behind our proposed strategy is to evaluate how much we can change the input X with regard to a particular direction in the input space, without switching an arbitrary test T_i. Therefore, the size of this change defines the "safe neighborhood" where derivatives may lay.

We recall Formula 3.9, and develop the dot product:

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

where dX can be decomposed as $dX = d\hat{X} \times \beta_i$, where $d\hat{X}$ represents the directional variation of the input, and β_i is the scalar that holds the magnitude of this variation, in particular β_i is related to the test $T_i()$. Developing Formula 3.13 with the dX decomposition, we obtain:

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

isolating β_i in Formula 3.14, we obtain:

$$\beta_i \geq \frac{-T_i(X_i)}{J(T_i(X_i))^t \times J(B_i(X_i)) \times \ldots \times J(B_1(X)) \times d\hat{X}} \tag{3.15}$$

Formula 3.15 is the constraint that β_i must satisfy for the test T_i, thus β_i holds the information about how much the input variation dX can increase following the given direction $\hat{X}$ of the input space.

To compute the directional domain of validity in a neighborhood of the input X regarding a certain direction, β_i has to be computed for every test in the program, and all the β_i must be gathered and combined into a single constraint on β.

Note that the repetition of the procedure for all the Cartesian basis in the input space returns less precise information than the model of Section 3.4, as can be observed in Figure 3.6.

Figure 3.6 shows the intervals represented by arrows starting from the input (black point), for instance the input directions in this example are the Cartesian basis. The inner shape (dotted lines rectangle) is an approximation that can be deduced using the interval representation. This deduction takes advantage of the convexity of the space of solutions (bold black lines).

The performance of the method is better in both aspects, memory and time, than HSBP method. This is because DFP computes certain directions in the input space, and the AD model behind the process is the forward mode, which is cheaper to compute than the reverser mode, which is behind HSBP. Unfortunately, DFP is less

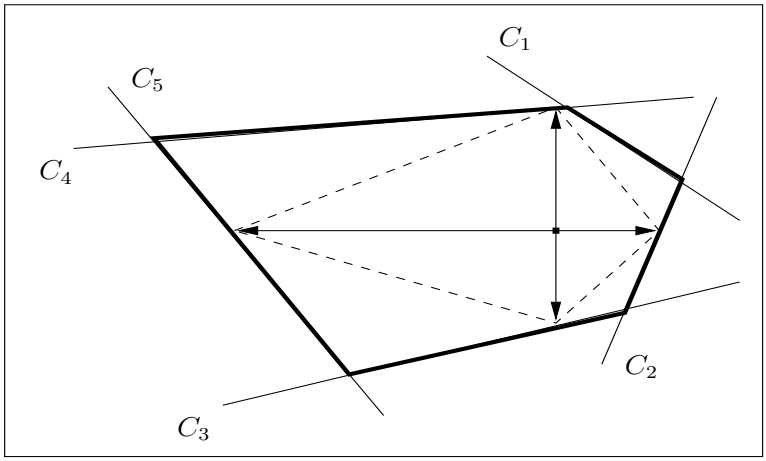


Figure 3.6: Directional representation for an arbitrary example.

precise, even if we project a polyhedron from the borders of the directional solution (as in Figure 3.6) the solution space will be a subset of the exact solution space.

3.5.2 Problems with DFP

DFP returns validity information of acceptable precision for certain directions in the input space. Unfortunately, as may happen for large code or code with an input space with many dimensions, the cost of applying the model is still expensive, and may be more than the cost that the users are able to accept.

To cope with this problem, and recalling the discussion of Section 3.4.2, we have mainly two options to reduce the cost of DFP model, the first is to select the most relevant tests, which require a good knowledge of the code, and eventually may lead to a good deal of manual work, if the number of tests is large. Secondly, the user can identify the most relevant directions in the input space, which is a cheap way to reduce the cost, because the decision about the relevant directions is made just one time before the application of the DFP model, which is automatic.

3.5.3 DFP Implementation

The implementation of DFP is based on the tangent mode of AD, because the tangent computes the directional variation of the output with respect to certain directions of the input. Thus, the implementation of DFP takes advantage of the computation of variation of the intermediate variables, therefore allowing the computation of every constraint.

For an arbitrary program $P = B_1 \ ; \ T_1 \ ; \ldots ; \ B_n \ ; \ T_n \ ; \ B_{n+1}$, the tangent differentiated version is $P' = B'_1 \ ; \ T_1 \ ; \ldots ; \ B'_n \ ; \ T_n \ ; \ B'_{n+1}$.

To implement DFP, we insert an instruction before every test of program P' . This instruction computes the value β_i for the test, and updates the global value β for the program.

For a general program P, the *domain-validated* program $\check{P}$ is as follows:

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

where V_i is the instruction that computes the value of β_i. The value of β_i is used to update the value of β. At the end, we obtain the value of β which holds the information for the whole program $\check{P}$. Finally, the interval of validity ("safe neighborhood") is built up from the β value.

Practically, the instruction V_i is not a single expression, it is a call to a subroutine, which runs the algorithm that updates the global β, which is basically an intersection of intervals algorithm. This algorithm is:

```
if ( td .ne.  0.0 ) then
    temp = - ( t / td )
    if ( temp .lt.  0.0 ) then
        if ( infmin ) then
            gmin = temp
            infmin = .FALSE.
        else
            gmin = max( gmin, temp )
        endif
    else
        if ( infmax ) then
            gmax = temp
            infmax = .FALSE.
        else
            gmax = min( gmax, temp )
        endif
    endif
endif
```

Table 3.4: Algorithm to update β.

where `t` and `td` are the inputs, and represents the test and the variation of the test. The variables `infmin, infmax, gmin, gmax` are initialized at the beginning

of the segment of code that is chosen to be analyzed. The two first variables are initialized to *true*, and their purpose is to indicate that the interval of validity has infinite boundaries, which is true at the initial stage. Variables `gmin, gmax` hold the computed bound of the interval.

3.5.4 DFP and AD Modes

The information computed by DFP is useful for all AD modes, because all of them are related by the chain rule, and codes that include tests are subject to tangent or reverse differentiation, or both. Therefore, a three-stage process to be sure about the differentiability of the output regarding the inputs is as follows. As first stage, run DFP for a certain input, and directions within the input space. As a second stage, use that information to select the input that avoids non-differentiability. As third stage, run the desired mode of AD with a safe input.

In the particular case of reverse mode of AD within an optimization process, the steps should be as follows:

1. For a given set of values, run the reverse mode of AD obtaining a gradient.
2. Use the gradient to obtain a descent direction.
3. Run DFP to explore the validity in this direction.
4. From DFP results, choose the step size and go to step 1.

3.6 Experimental Results

In this section, we show how the implementation of DFP works, and how the results are expressed. We first present the numerical results obtained by the application of DFP on the example given in Section 3.1. In Section 3.6.2 we apply DFP to the Newton method, and we discuss the results. Finally, in Section 3.6.3, we applied DFP to two scientific programs.

3.6.1 Basic Example

As first case, DFP is applied to the example code of Table 3.5 regarding both directions of differentiation. We obtain the the differentiate code of Table 3.5.

Table 3.5 shows the subroutine call `VALIDITY_TEST(x1 - x2, x1d - x2d)`, where `VALIDITY_TEST()` is the subroutine that computes the local β_i and updates the global value β. This computation is carried out by the algorithm presented in Table 3.4. The subroutine has as inputs the test $T1 = x1 - x2 \geq 0$ and its variation $dT1 =$ `x1d`

```
   subroutine F_d(x1,x1d,x2,x2d,y,yd)

   i1d = x1d * x1 + x1 * x1d
   i1 = x1 * x1
   i2d = x2d * x2 + x2 * x2d
   i2 = x2 * x2
   i3d = i1d + i2d
   i3 = i1 + i2
   i4d = i3d * COS(i3)
   i4 = SIN(i3)
   CALL VALIDITY_TEST(x1 - x2, x1d - x2d)
T1 IF (x2 .LT. x1) THEN
      i5d = -x2d
      i5 = 1 - x2
   ELSE
      i5d = -x1d
      i5 = 1 - x1
      i5d = -i5d
      i5 = -1 * i5
   END IF
   i6d = i4d * i5 + i4 * i5d
   i6 = i4 * i5
   yd = (i6d * i3-i6 * i3d) / i3**2
   y = i6 / i3
```

Table 3.5: Tangent Differentiated example code with Validity Analysis.

`- x2d`.

In Table 3.6 we can observe an extract of the results of the first experiment, which is to execute the code of Table 3.5 with inputs $(x1, x2) = [-2, 2][-2, 2]$ when $x1 + x2 == 0$, and following the direction $(x1d, x1d) = (1, 4)$.

Basically the results of Table 3.6 can be separated in two groups, one group is composed by the points before reaching the test $x1 == x2$, for those points the interval of validity has the form $[\infty, 0.*]$, that is, in the given direction the interval is reducing due to proximity to the test. In the opposite direction to the given direction the interval is infinite, because there is no differentiability problem in that direction. Conversely, the second group has the form $[-0.*, \infty]$, because in the opposite direction to the given direction the interval increases as the input gets far from the test, and the values are negative because they are in the opposite direction of the given

x1	**x2**	**yd**	**[gmin**	**gmax]**	**infmin**	**infmax**
0.399999	-0.399999	-3.57733	0.0	0.266666	T	F
0.299999	-0.299999	-3.83849	0.0	0.199999	T	F
0.199999	-0.199999	-3.95735	0.0	0.133333	T	F
0.099999	-0.099999	-3.99533	0.0	0.066666	T	F
-0.000001	0.000001	0.995140	-0.000002	0.0	F	T
-0.100000	0.100000	1.004333	-0.066666	0.0	F	T
-0.200000	0.200000	1.037307	-0.133333	0.0	F	T
-0.300000	0.300000	1.134554	-0.200000	0.0	F	T

Table 3.6: Numerical results of the example, with $(x1d, x2d) = (1, 4)$.

direction.

In Table 3.6 we use the values of infmin and infmax to represent the ∞ bound. This is the case when the bounds has not been updated. We use that variables because there is no such thing as infinite the real number represented by computers. In the rest of the thesis we will use the symbol ∞ in order to simplify the tables.

x1	**x2**	**yd**	**[gmin**	**gmax]**
0.399999	-0.399999	0.746525	-0.399999	∞
0.299999	-0.299999	0.901311	-0.299999	∞
0.199999	-0.199999	0.973350	-0.199999	∞
0.099999	-0.099999	0.997000	-0.099999	∞
-0.000003	0.000003	1.271568	$-\infty$	0.000003
-0.100000	0.100000	0.997000	$-\infty$	0.100000
-0.200000	0.200000	0.973350	$-\infty$	0.200000
-0.300000	0.300000	0.901311	$-\infty$	0.300000

Table 3.7: Numerical results of the example, with $(x1d, x2d) = (1, -1)$.

In Table 3.7 we can observe the same behavior of the results of Table 3.6, but this time, because the direction of differentiation is different, the two groups of results are switched with respect to the test. When the point is getting closer to the test the interval is reduced, and the value is negative because it is obtained the in the opposite direction of to the given input direction. After the test the *gmin* bound is infinite because there is no problems of differentiability in the opposite direction of the given direction.

Figure 3.7 shows a graphical representation of the derivatives of both experiments, including the aforementioned interval of validity. The interval of validity is

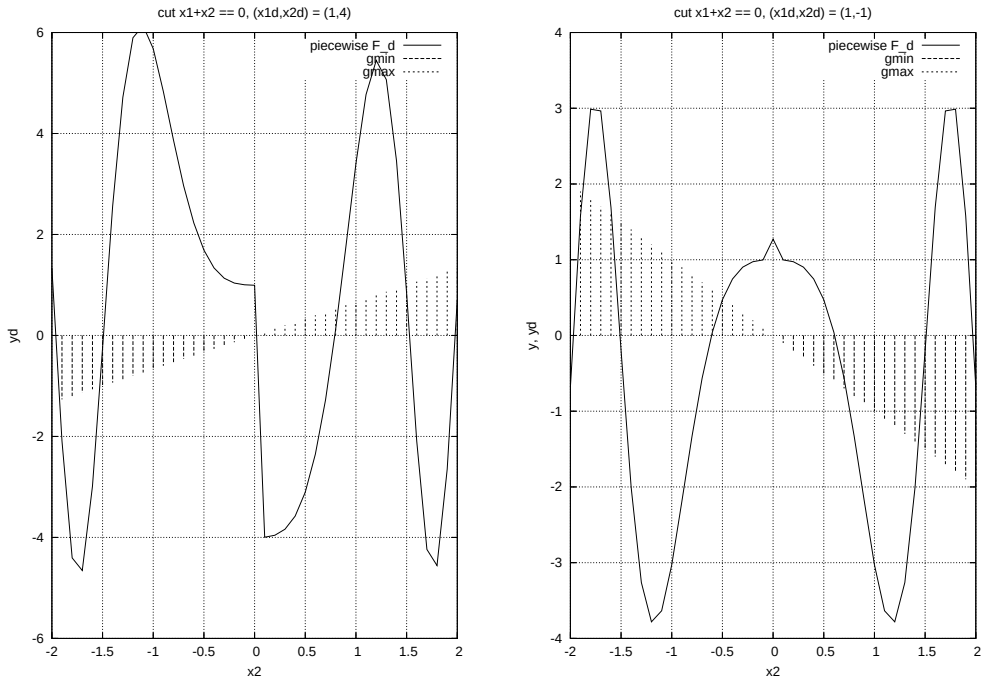


Figure 3.7: Experiments with two directions of the input space, and the computed validity information.

represented only for the boundary which is not infinite. In this case DFP successfully informs the validity information.

3.6.2 Experiments with the Newton Method

The goal of this experiment is to show how the validity information may help a general method, in particular when the method includes piecewise or non-smooth functions [26], which at implementation level are implemented as conditional statements.

The Newton Method is an iterative algorithm to finding approximations to the zeros (or roots) of a real function, as can be defined in the following form:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3.16}$$
$$f \in C^1 : [a,b] -> I\!R$$

where the function $f()$ is differentiable and defined on the interval $[a, b]$. The algorithm starts with an initial $x_0 \in [a, b]$, and iterates with $n \in I\!N$ until the root is found, or some other stopping criterion. The method can also be used to find a local maximum/minimum, in that case the function $f()$ has to be differentiable twice, and what the method solves can be seen as the search for the root of $f'()$. The method to search for a local minimum has the following form:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)} \tag{3.17}$$
$$f \in C^2 : [a, b] -> I\!R$$

The method can be extended to arbitrary dimensions by replacing $f'()$ by the gradient $\nabla f()$, and the second derivative $f''()$ by the Hessian matrix $Hf()$ of $f()$.

We implement Formula 3.17 with the iterative method to search for a local minimum, thus we can use AD to compute $f'()$ and $f''()$ for the implementation of an arbitrary function $f()$. In order to carry out the experiment we implement a function $f()$ as follows:

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`

Table 3.8: Function $f()$ and its tangent differentiated version.

Table 3.8 shows the implementation of function $f()$, and its second derivative $f''()$, both needed to carry out the iterative method. The first and second derivatives are implemented in function `FUNCTION F_D_D()` of right sector of Table 3.8, this implementation was obtained by applying the tangent mode of AD on the code that implements the function $f()$ two times. The implementation of the Newton method itself is presented in Appendix B.1, along with the complete procedure to replicate the results of this section.

In Figure 3.8 we can observe the results of the method on $f()$. The method reaches the solution, and it takes the same number of steps as a hand-made version does, therefore the implementation of the method using AD derivatives is viable and efficient, at least for this kind of functions.

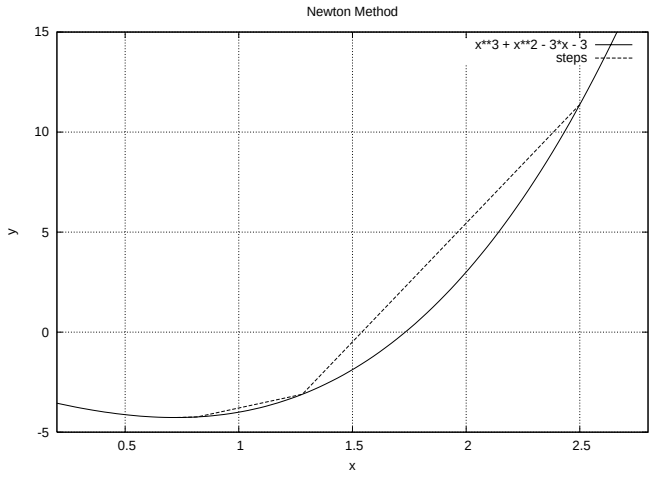


Figure 3.8: Newton method using derivatives generated by AD.

The goal of this section is to experiment with piecewise functions. Therefore we now present cases of piecewise functions which can be used with the Newton method.

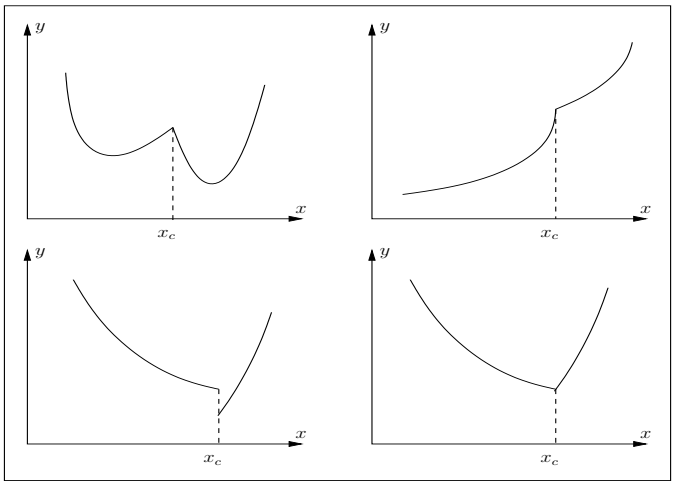


Figure 3.9: Examples of piecewise functions for the Newton method.

The four piecewise functions presented in Figure 3.9 introduce problems for the Newton method. The convergence to the local minimum for the first (top-left) is strongly related to the initial guess, thus the connection between functions becomes an insurmountable barrier, for the second function the fact that it is piecewise only produces a delay in the convergence. In the last two cases, the bottom area of Figure 3.9, the convergence is not clearly achieved due to the drastic difference between the the first derivative of the component functions. It is possible to have a cycle of never

ending iterations. Therefore, we implement a piecewise function *f_piecewise*() which exhibits a similar behavior as the last case (bottom-right) of Figure 3.9. This decision is based on the idea that the validity information could help more than just warning about inconsistent derivatives.

Piecewise Function Code	**Second Order Differentiated Code**

`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`

Table 3.9: Function *f_piecewise*() and its tangent differentiated version.

Table 3.9 shows the implementation of function the *f_piecewise*() and its derivatives, the first and second order derivatives required by the Newton method. The first branch of function the *f_piecewise*() is similar to the function $f()$ of the previous experiment. The Newton method is applied to the piecewise function with the same set-up used with function $f()$, and we obtain the following results.

In Figure 3.10 we can observe that convergence is not reached, and in Table 3.10, which is truncated at iteration 10, we can observe that. This is because the iterative process falls in a loop. This loop is composed of three steps, for instance, the first loop goes from iteration 5 to iteration 7.

We applied DFP to the piecewise function in order to compute the validity information along the Newton method, obtaining the results of in Table 3.11.

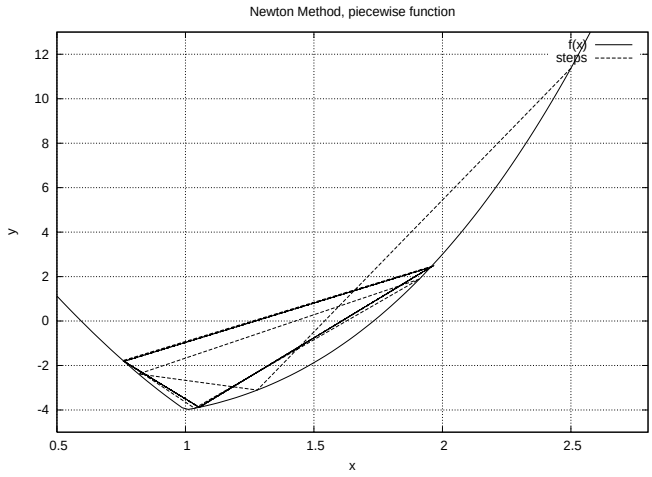


Figure 3.10: Piecewise functions.

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

Table 3.10: Numerical results of the Newton method on piecewise function.

This experiment is specific because the local minimum is at the same time the connection point between the two component functions of $f\_piecewise()$. As a result of this, we can observed in Table 3.10, Figure 3.10 and Table 3.11 convergence is not achieved, and the method shall iterate forever without satisfying the stopping criterion.

Once validity information is computed for every iteration, the interval of validity warns about the proximity to the test, in this case the local minimum. For example, for step 1 the interval of validity is $[-0.27941, \infty]$, that is in the given direction ($xd = 1$) there is no problem of differentiability, on the other hand, the point is very close to the test and to a drastic change in the first derivative value. As we can

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	∞

Table 3.11: Numerical results of Newton method and DFP on piecewise function.

observe in Table 3.11, the step 2 is even closer to the local minimum, but anyway the stopping criterion is not satisfied.

Taking into account the behavior of the Newton method when it deals with a certain type of functions, and taking into account the useful information generated by DFP. First, The above results show that DFP successfully informs the validity information for this iterative method. Second, the validity information may be useful for a user of the Newton method when the objective function is not smooth.

We propose that the validity information may be used not only to warn the user about non-differentiability, but also as feedback to algorithms, specially when those algorithms are dealing with non-smooth functions. For instance, in this case the Newton method can be modified, that is by improving either the stopping criterion or the computation of the next step regarding the validity information.

3.6.3 Experiments with Real-Life Scientific Programs

The goal of this was to test the scalability of DFP. Therefore, we mostly focus in how the method behaves dealing with large codes, in particular from the computational point of view. We conducted a large number of tests on two real-size programs. Basically, the experiments were not focused on the mathematical meaning of the results, because neither the models behind the codes nor the inputs are in our domain of knowledge. Consequently, we handled these as black boxes.

In Table 3.12 we can observe the description of the codes used in the experiments, the information given in the table is mainly related with the size of the codes and the

Program	Application domain	Lines code	# Tests	# Analyzed tests
STICS	Agronomy	27.000	2.682	542
CEA	CFD	19.789	1.864	189

Table 3.12: Real-life program settings.

number of tests within. The column named *analyzed tests* represents the number of test which have influence in the derivatives, thus the validity information is commutated only for them.

DFP proves to be cheap in execution time. The execution time of the tangent differentiated code STICS is 0.529 seconds. The execution time of the tangent differentiated code STICS with the validity deployment is 0.551. Therefore, the overhead of the DFP is just 4%. The provided execution times are average of multiple executions of the codes.

The results require close analysis from the end-user in order to be mathematically useful, but from our point view, the results are promising, in the sense that they are consistent with the predicted behavior, that is with the behavior around the randomly selected input and its derivatives.

3.7 Related Techniques

This section is a selection of techniques from the literature, all these techniques are related to the problem of calculating derivatives when the functions involved have problems of differentiability, in particular when the problem is generated by the control flow. We discuss four approaches, Interval Extension, Ideal Discontinuity, Sub-differentials and Laurent Series.

3.7.1 Interval Extension

The goal of this approach is to deal with non-differentiable functions using interval extensions, in particular with conditional statements. The interval extension $F(X)$ is the interval which encloses the extreme values of the function results, $\{f(x) \mid x \in X\} \subseteq F(X)$, where X is the input domain. Functions with branches are represented as follows:

$$F(X) = \chi(x_s, x_q, x_r) = \begin{cases} x_q & \text{if } x_s < 0 \\ x_r & \text{if } x_s > 0 \\ x_q \underline{\cup} x_r & otherwise \end{cases}$$

Where, x_s is the decision expression, x_q and x_r are the interval evaluations of function of each branch, and $x \underline{\cup} y$ is the interval hull of the interval evaluation of functions x and y.

Because interval extension is applicable when the functions are smooth, a new property is necessary, thus adapting interval extension to non-smooth functions. Basically, if the jacobian $F'(X)$ is bounded, closed and convex set as Lipschitz sets [50], it is possible to define interval extension for every case of branches and non-differentiable functions (as the intrinsic functions describe in Section 3.2.1).

For example, the derivative extension (when $\chi(0^-, x_q, x_r) = \chi(0^+, x_q, x_r)$) is defined as follows:

$$\frac{\partial \chi(x_s, x_q, x_r)}{\partial x_q} = \begin{cases} 1 & \text{if } x_s < 0 \\ 0 & \text{if } x_s > 0 \\ [0,1] & otherwise \end{cases} , \quad \frac{\partial \chi(x_s, x_q, x_r)}{\partial x_r} = \begin{cases} 0 & \text{if } x_s < 0 \\ 1 & \text{if } x_s > 0 \\ [0,1] & otherwise \end{cases}$$

$$, \frac{\partial \chi(x_s, x_q, x_r)}{\partial x_s} = 0$$

This approach was developed to verify solutions of non-linear systems of equations, and for global optimization [41, 42]. The author aims to use these interval extension in the AD context through operator overloading, using the package INTLIB [40]. The approach is related to our work because it presents a solution to the problem introduced by the conditional statements, but Kearfott's approach is only partial because does not cover all possible cases, and it is defined within the interval arithmetic framework, which is not a standard in our field.

3.7.2 Sub-differentials

The sub-differential of a function $f()$ at point x_0 is the set

$$\partial f(x_0) = \{y : f(x) - f(x_0) \geq \langle y, x - x_0 \rangle\} \tag{3.18}$$

For a function of one variable this is the collection of angular coefficients y for which the lines $f(x_0) + y(x - x_0)$ lie under the graph of f, as is represented in Figure 3.11.

The sub-differential is a concept which generalizes the derivative for a convex function, and if f is differentiable at x_0, then $\partial f(x_0) = f'(x_0)$. [36]

The sub-differential concept belongs to the *convex calculus*, which is widely used in convex analysis, and therefore also in optimization research [20, 36]. We have

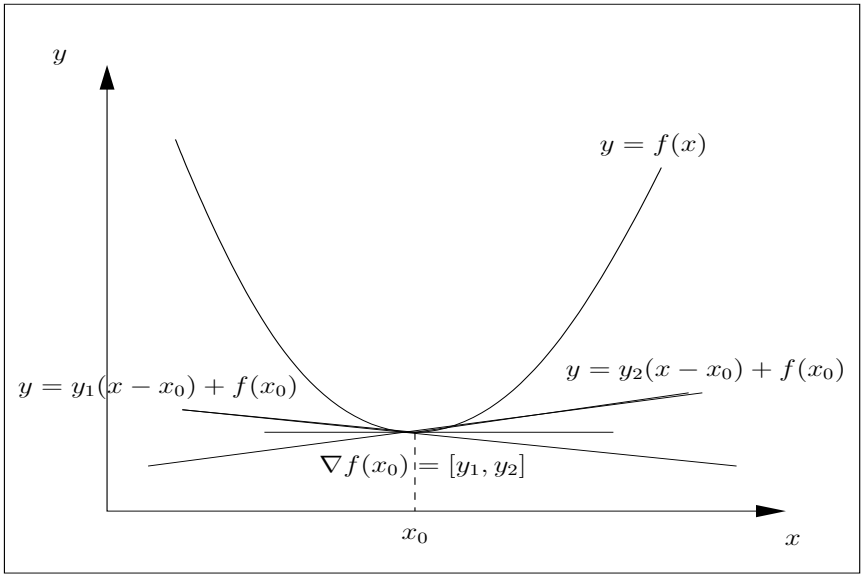


Figure 3.11: Sub-differential example.

no information regarding an implementation of the sub-differential approach within the AD community. However, we consider the approach connected with our work in the sense that they both look to overcome the problem of non-differentiability, particularly non-differentiability in certain points of the input space. Unfortunately, this approach has three problems. First, it is computational expensive, due to the several executions required to compute Formula 3.18. This formula computes an interval, this interval represents the derivative. As a result, the second problem arises, even if the user knows exactly the input which produces the non-differentiability, the returned derivative for that input will be an interval. This probably requires the utilization of the interval arithmetic, which change completely the framework of work. The third problem comes from the fact that this technique requires certain mathematical conditions (convexity) in order to be applied correctly. Therefore, the range of possible applications is limited for this technique.

3.7.3 Laurent Series

This method deals with functions with known troublesome inputs, as the ones listed in Section 3.2.1. The method is based on Laurent series, which original use is in complex analysis. Laurent series represents a function as a power series which includes terms of negative degree. Usually, this method allows to express complex functions in cases where a Taylor series expansion cannot be applied.

In the AD context, Laurent series are one-side expansions of problematic func-

tions, like $|x|$ and $x^2 log(x)$ at the origin. The method requires that the user's input be a point x_0 for the evaluation and differentiation, and also a direction x_1. With this input the directional differentiation can be performed in an effective way.

However the problem arises when the method obtains roots with an uncertain sign, i.e. when the number of significant terms may vary from intermediate to intermediate steps. The Laurent series can be truncated, thus becoming a Taylor polynomial. Hence, the method is applicable as long as we deal with the latter [27].

3.8 Conclusions and Future Work

The question of derivatives being valid only in a certain domain is a crucial problem of AD. If derivatives returned by AD are used outside their domain of validity, this can result in errors that are very hard to detect. AD tools must be able to detect this kind of situation.

The origin of the problem is due to two aspects: problematic intrinsic functions and changes in the flow control. Fortunately, the first kind of problem can be transformed into the second kind of problems. Thus we only have to deal with one kind problems: the changes in the control flow.

We proposed two methods to tackle the problem of non-differentiability in programs differentiated with Automatic Differentiation. Both rely in analyze every conditional statements, in order to determine which is the neighborhood from which we can obtain reliable derivatives. This safe neighborhood was called domain of validity.

The first method we proposed HSBP allow us to compute the domain of validity for a given input with an acceptable precision. The method computes the validity information for every conditional statement (constraint) and then propagates the information, to do that it uses the reverse mode of AD.

Unfortunately, even although HSPB provide us with a complete description of the domain of validity, the cost of this method is prohibitively high, both in execution time and memory consumption. The main problem to apply the method is the size of the set of constraints.

Several alternatives to reduce the cost of the method were presented. Basically, these alternatives are: to select the most relevant constraints, and to change the solution space representation. Unfortunately, we explained that the above alternatives have small impact on reduce the computational costs, therefore we look for another method.

The second proposed method DFP computes the validity information following a given direction in input space. Thus the representation of the validity information becomes a interval. If the input remains in this interval, the returned derivatives have no problem of differentiability. The implementation of this method is based on the tangent mode of AD. The computational cost of DFP is marginal (4%) with respect the computational cost of the tangent mode. That small overhead is added to the computation of derivatives when DFP analyze every test within the code, but only for certain directions in the input space.

We have presented method DFP, which proves to be useful, thus it is one possible way to warn the user from abusive use of the derivatives.

The future work for DFP includes to implement a tracking system to determine the conditional statement which has more relevance to the validity information. This system can be implemented as an extra parameter of the subroutine `VALIDITY_TEST`. This parameter can be updated by the subroutine with the information regarding the increase/decrease of the bounds of the validity information.

Further improvements to the method, which would increase the accuracy is to use a higher-order approximation to compute the validity information. This can helps in case when the first order approximation is not enough precise, therefore not helping to avoid the problems that motivate this research.

Finally, we believe that the method can be integrated to a large algorithms/methods. But only to warn the user about possible problems, also to add a new kind of feedback to the algorithm, which can helps to improve the performance of the it.

Chapter 4

Data-Flow Analyses and Checkpointing for the Reverse Mode

Contributions of this chapter:

- We introduce a mathematical specifications for data-flow analyses used by reverse mode AD. This mathematical specifications are based on classical data-flow analyses for a model of reverse mode of AD.

- We formalize an improved AD model with Store-All strategy. This improved model relies on the above adjoint data-flow analyses.

- We extend the above AD model in order to take into account a checkpointing strategy, this strategy consists of systematically checkpoint subroutines calls, thus looking for reduce the memory space consumption.

- We implement the extended AD model in our AD tool TAPENADE. This implementation also add a new functionality that enables the users to specify checkpointing strategies. Accordingly, we perform a thorough study of the effect of checkpointing strategies on real-size scientific programs.

In this chapter we investigate ways to cope with the main problem of the reverse mode of AD. Although the reverse differentiated program structure has a high efficiency in execution time of , this structure is also source of the main problem of the reverse mode of AD. Recalling from Section 2.5.1, the reverse differentiated structure of a program is composed of two main parts, the first part is called the forward sweep and the second part is called the backward sweep. The forward sweep has basically the same instructions as the original program. The purpose of the forward sweep is to compute intermediate values required in the backward sweep. The backward sweep implements the derivatives. During the execution of the forward sweep variables are re-defined, thus changing their intermediate values. The problem arises when the intermediates values are required by the derivatives, but they are not accessible anymore because they have been overwritten.

In this thesis we focus on a strategy to cope with the above problem. This strategy is called Store-All (SA). The SA strategy consists of storing the intermediate variables values in the forward sweep, then restoring them in the backward sweep, this makes the intermediate values accessible to the instructions that implement the derivatives. The problem with this approach is the possibly unacceptable high memory consumption [22]. This is what motivates our research. We have investigated two kind of optimization in order to handle the memory problem. The first kind of optimizations improves the differentiated code at the level of instructions, and is based on data-flow analyses. The second kind of optimizations works on segments of code, this segments have both arbitrary control flow and size.

In the following section, we detail the problem and present the main two strategies to handle it. We give a description of both strategies, the SA strategy and the Restore-All (RA) strategy, where RA consists of recomputing the required variables. A trade-off between both strategies is also introduced.

This chapter is organized as follows. In Section 4.1 we introduce the main problem of the reverse mode of AD. In Section 4.2 we present the classical strategies to cope with the mentioned problem. In Section 4.3 we present a initial formal model for reverse differentiation, which takes into account the SA strategy. Also, in section 4.3 we present the general framework for the sequel of the chapter, this sequel is split between instruction-level and code segments kind of optimization. In Section 4.4 we present our contributions to the instruction-level kind of optimizations. In Section 4.5 we present our contributions to the code segments kind of optimizations. Finally in Section 4.6 we conclude and discuss future work.

4.1 The Memory Consumption Problem of the Reverse Mode

The Reverse mode computes adjoint codes, in particular gradients. Due to the structure of the reverse mode of AD, the computation of that kind of code is very efficient in terms of execution time. But this has the cost of requiring a large amount of intermediate variable values, which may be lost by a re-definition of the variable during the forward sweep. Therefore, the reverse mode has a drawback. Because intermediate variables are re-defined during the forward sweep, the values that those intermediate variables held are not accessible in the backward sweep, when derivatives may require them. For instance, let us consider the following example:

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

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

where program $\overline{P}$ is the reverse differentiated version of program P, and the instructions I_i' implement the derivative of the corresponding instructions I_i.

In Formula 4.2, the reverse derivative of instruction I_2 is composed of two terms, that is $I_2' = (xb = y \times I_2 b \ ; \ yb = x \times I_2 b)$, according with Table 2.4. If we assume that variables x and/or y are re-defined in I_3, then the values of those variables held at the moment of the computation of I_2 are out of reach when they are required by instruction I_2'.

Actually, programs include several re-definition of variables, just imagine variables within loops. Thus the scenario presented by Formula 4.2 is common in industrial/scientific programs. In order to avoid that problem, we may ask the users to modify their codes, but this is not realistic, thus AD models have to overcome this problem in a systematic way. In the next section, we explore the main alternatives to cope with the mentioned problem.

4.1.1 Store-All Strategy vs Recompute-All Strategy

In order to provide the lost intermediate values required by the reverse mode, we mainly have two options: *Recompute-All* (RA) [24] and *Store-All* (SA) [27] strategy.

The RA strategy aims at solving the problem by not storing intermediate values in the forward sweep, but just carrying out the forward sweep which generates the initial state of the backward sweep. Then, when the backward sweep reaches an instruction which needs a certain value out of reach, the RA strategy computes the whole sequence of original instructions required to produce the needed value. In the right area of Figure 4.1 (page 72) we can observe the memory usage of a brute force

implementation of the RA strategy.

Rule	Program	Reverse Differentiated Program
R_0	$P = ()$	$\overline{P} = ()$
R_1	$P = I$	$\overline{P} = I'$
R_n	$P = U \ ; \ I$	$\overline{P} = \bullet \ ; \ U \ ; \ I' \ ; \ \circ \ ; \ \overline{U}$

Table 4.1: RA ruleset

Table 4.1 presents the minimum set of recursive formulas needed to build a reverse differentiated program in RA fashion, where symbol () means the empty set, the U represents the sequence of instructions from the beginning of the program until before a certain instruction I, the symbol $\bullet$ depicts the storage of the input variables values, and the symbol $\circ$ depicts the restoration of the previously stored values.

$$\overline{P_{ra}} = \bullet \ ; \ I_1 \ ; \ I_2 \ ; \ I_3' \ ; \ \circ \ ; \ I_1 \ ; \ I_2' \ ; \ \circ \ ; \ I_1' \tag{4.3}$$

Formula 4.3 shows the reverse differentiated program example (from Formula 4.1), program $\overline{P_{ra}}$ in Formula 4.3 was obtained using the rules presented in Table 4.1, thus implementing the reverse differentiated program in RA fashion. Therefore, in order to make the required values to instruction I_3' accessible, the sequence of instructions $I_1 \ ; \ldots ; \ I_2$ is computed, and so on for the rest of derivative instructions. To ensure that the re-computation of segments will produce the correct values, the input values of program P should be stored one time, and restored as many times as they are required.

RA is demanding in execution time, quadratic with respect to the number of instructions (brute force), because it re-computes the intermediate values every time they are required. The worst case scenario happens when the required values are the last values vanished in the forward sweep, thus forcing the re-computation of most of the original sequences of instructions.

In contrast, the SA strategy consists in storing in a special stack-wise memory structure (*tape*) all the intermediate values that will be required in the backward sweep. In the backward sweep the values from the tape are restored in the reverse order with respect to order of storage. This results in the structure of reverse differentiated programs shown on left area of Figure 4.1 (next page).

Table 4.2 presents the minimum set of recursive formulas needed to build a reverse differentiated program in SA fashion, where the D represents the sequence of instructions after certain instruction until the end of the program, symbol $\bullet_i$ depicts

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

Table 4.2: SA ruleset

the storage of the variables values to be overwritten by instruction I_i, and symbol $\circ_i$ depicts the restoration of the previous stored values.

$$\overline{P_{sa}} = \bullet_1 \ ; \ I_1 \ ; \ \bullet_2 \ ; \ I_2 \ ; \ I_3' \ ; \ \circ_2 \ ; \ I_2' \ ; \ \circ_1 \ ; \ I_1' \tag{4.4}$$

Formula 4.4 is the reverse differentiated version of the program example (from Formula 4.1), it implements the SA strategy. Therefore, before the variables are re-defined their values are stored, and then restored in the backward sweep before the instruction that requires them. For instance, the last instruction of program $\overline{P_{sa}}$ requires values which may be overwritten by I_1, thus before I_1 the values are stored in $\bullet_1$, then restored from $\circ_1$ before I_1'. The same procedure is also implemented for I_2', but it is not the case of I_3' because definitions in I_3 are useless for I_3', then I_3 is not needed at all, therefore there is no need to store values before I_3.

As was presented in the Section 2.6, the RA strategy is linear order with respect to memory consumption and execution time. The worst case scenario occurs when it is necessary to store a large number of values, in that case the tape might grows to an unacceptable size.

After considering Figure 4.1, it is clear that the efforts to improve the RA strategy should be focused in reducing execution time. On the other hand, the drawback of SA is the high memory consumption, as can be observed in Figure 4.1, the peak of memory consumption occurs at about the end of the forward sweep. Comparatively, the peak of memory consumption of the SA strategy is far bigger than the one of the RA strategy, but the execution time required to compute the program for the SA strategy is shorter than the execution time required by the RA strategy. In order to visualize the strategies performance, we carried out experiments with brute force versions of both strategies.

4.1.2 Experimental Measurements

In order to compare both strategies, we present experimental measurements made on a small straight-line example. The selected code does not include subroutine calls. The code is small because we have no access to a tool that produces RA differentiated code. Therefore, the transformation needed to implement the RA strategy version of

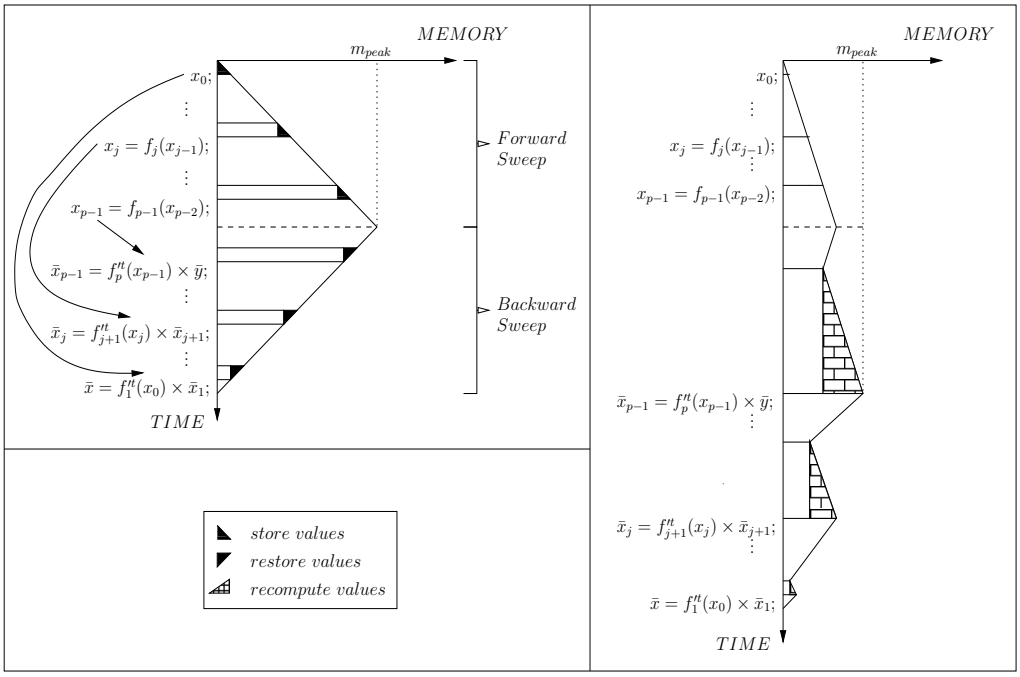


Figure 4.1: Plot of RA and SA strategies, where the horizontal axis represents the amount of values currently on the stack.

the code were hand-made, the results are numerically consistent with the one obtained with the SA strategy version of the code.

Experiment	Execution Time [s]	Memory [bytes]
RA	51.1	8
RA v1	49.2	16
RA v2	48.7	16
RA v3	49.4	12
SA	43.7	28
SA v1	44.3	20
SA v2	44.8	16
SA v3	45.2	16

Table 4.3: Experiments RA vs SA

In Table 4.3 we can observe the two groups of results. First, the results of the RA strategy and its variations, where these variations replace re-computations by storage of values. Second, the results of SA and its variations, where these variations replace an intermediate variable value storage by a value re-computation.

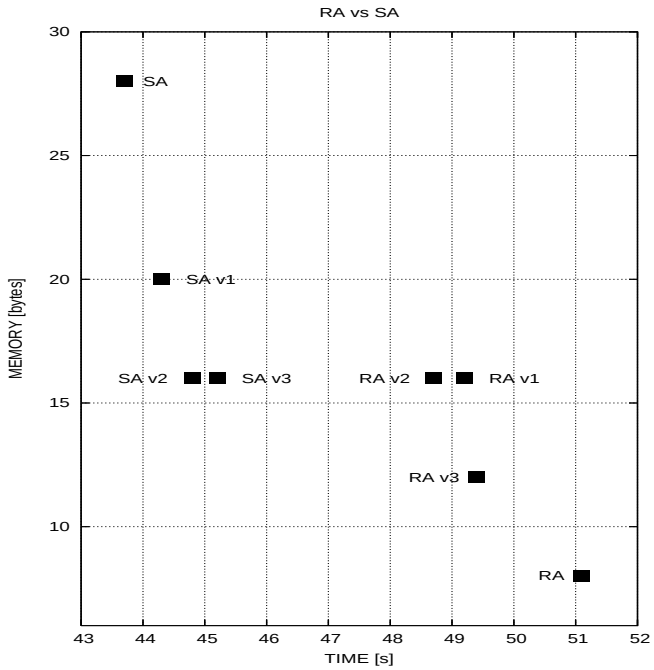


Figure 4.2: Experimental results. Graphical distribution in a memory vs time.

Figure 4.2 shows the graphical distribution of the results presented in Table 4.3. As we can observe, the extreme points are the result obtained by the application of the strategies in their "basic" form. The rest of the results are local optimizations based on trading storage and re-computation.

We propose that in general codes, local optimizations follow the pattern of Figure 4.3, which is based on our experimental results. Nonetheless, we admit that the pattern may be changed by some features of codes no considered in the experiments. The results of Table 4.3, graphically represented in Figure 4.2 and 4.3, help us to clearly realize the trade-off between the recomputing and storing. While SA outperform RA in execution time, RA outperform SA in memory space.

4.1.3 The Store-All Strategy and Memory Constraint

After evaluating the previous experiments and taking into account the positive experience of the TROPICS team about the RA strategy, we decided to study the SA strategy in depth. We believe that it is a very promising strategy starting from the fact that in execution time it performs better than the RA strategy. On the other hand, memory consumption may be improved and that will be our focus.

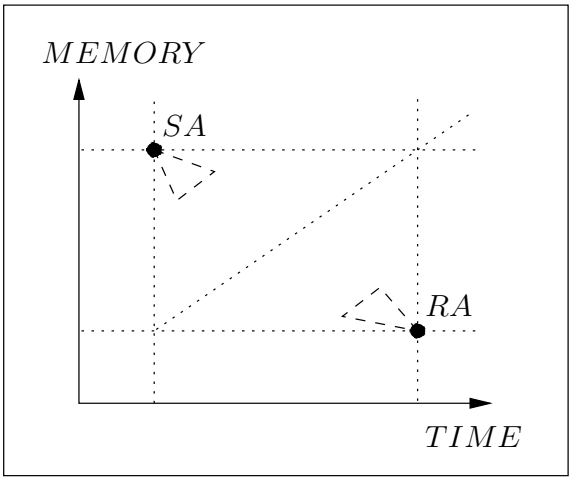


Figure 4.3: Graphical projection of local optimization for both, SA and RA strategies.

We believe that the memory constraint is the most pressing constraint. For instance in the hardware industry, CPU technology moves faster than RAM technology. The latter is bound by the Moore's Law [45] of integrated circuits (the number of transistors doubling every 18 months) and the former only by a constant ratio of growth of about 10% increase. Thus, memory space is the constraint we choose to address.

Furthermore, Due to the relationship between the RA and SA strategies, our developments in SA strategy in order to improve the memory consumption overhead, will be adaptable to RA strategy. This will produce improvements in execution time.

In the next section, we present the classical strategies within SA strategy, which is the starting point of our research.

4.2 Classical Strategies for the Store-All Approach

To control the memory problem produced by the storing of intermediates values, the store-all strategy can be improved in two main directions. First, by refining the data-flow analyses in order to reduce the number of values to store, and the instructions to generate. These data-flow analyses are instruction level kind of improvement, thus we called them fine-grain strategies. Second, deactivate the store-all strategy (without activating recompute-all) for chosen segments of the code, therefore allowing us to spare memory space. Because the scope of this strategy are segments of code, we call this a coarse-grain strategy.

The goal of this section is to address the above two main directions of improvements. Consequently, in the following we present the classical fine-grain strategies used to cope with the memory problem of the reverse mode. This will be followed by the same exercise for the coarse-grain strategy.

4.2.1 Fine-Grain Strategies

Data-Flow Analyses

Data-flow analyses are static, because they work at compilation time and without any runtime information. These analyses are conservative in terms of the results, thus avoiding cases where the result of the analysis is uncertain, if this happens the worst case is assumed. Another hazard to the data-flow analyses is the combinatorial explosion problem. To handle it, usually the data-flow analyses are designed as a hierarchical model. In this kind of model two sweeps through the code are performed, the first sweep is bottom-up and computes local synthesized information , thus this information is independent of the rest of the program (context free). Conversely, the second sweep is top-down and context dependent, thus propagating the synthesized information of the first sweep through the program.

The above characteristics and desired features are the framework for all the data-flow analyses presented in this thesis. Also it is important to notice that the data-flow analyses equations are solved using fixed point iteration. Although for some particular cases, it is possible to solve the equations without fixed point, but this is a theoretical remark because in practice all implemented algorithms do fixed point iteration.

The traditional data-flow analyses are focused on generic code, extensions to these analyses that take into account the particularities of the AD generated code were introduced in [17, 47, 31], where *Activity* analysis and *To Be Recorded* (TBR) were defined as:

- Activity analysis [31]:

  The optimization of differentiated code by activity analysis can be considered as an specific partial evaluation kind of optimization, because from all possible paths between the input and output variables only some of them hold differentiable influence, and so the variables along the paths. Not all the variables have differentiable influence, the ones which have this influence are called active variables, the analysis that determine this state of the variables is called activity analysis. This analysis is general in the sense that is used by all modes of AD. In order to determine the set of active variables in a piece of code we have to

use the dependency relationship. A variable is active at some place in the code if the following two conditions are satisfied:

- the variable depends in differentiable way on an *independent variable*,
- at the same time a *dependent variable* depends in differentiable way on the variable.

Among the output variables of the code, the ones for which the derivatives are requested are called dependent variables. The independent variables are input variables with respect to which the dependent output must be differentiated. Graphically, a variable is active if it belongs to at least one path of the dependency graph of a program, where that path starts from an independent variable, and the path ends at a dependent variable.

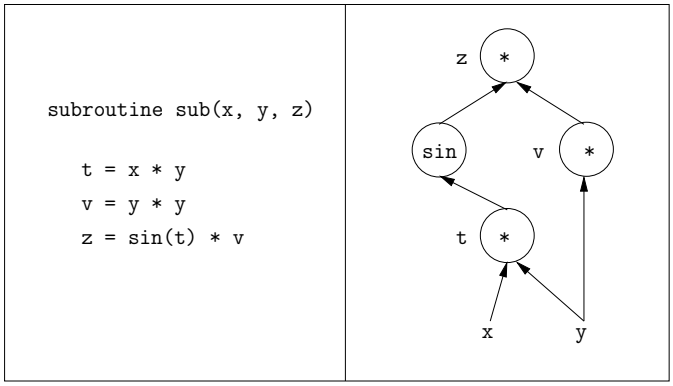


Figure 4.4: Example dependency graph.

If the code in Figure 4.4 is differentiated with respect to independent variable `x` and dependent variable `z`, the variable `t` is active, because variable `t` depends on the independent variable `x` and at the same time, dependent variable `z` depends on it. In contrast, variable `v` is not an active variable because depends on `y` which is not an independent variable. Alternatively, if the code in Figure 4.4 is differentiated with respect to independent variable `y` and dependent variable `z`, variables `t` and `v` are active, because both variables depends on independent variable `y` and dependent variable `z` depends on them.

In order to detect active variables, the activity analysis runs three analysis. The first analysis is called *differentiable dependency analysis*, it computes for basic blocks the differential dependency of every pair of variables. For instance, the dependency across a piece of code A is defined by the following data-flow

equation:

$$\mathbf{Dep}(A) = \{(v_o, v_i) \in \text{Outputs}(A) \times \text{Inputs}(A) \mid v_o \text{ depends on } v_i\}$$

where the combinator $\otimes$ is defined as

$$V \otimes \mathbf{Dep} = \{x \mid \exists y \in V \mid (y, x) \in \mathbf{Dep}\} \tag{4.5}$$

Once the above analysis is finished, and the information about dependencies is synthesized, the second and third analysis are executed. The second analysis is called *varied*, it computes the set of variables that possibly depend on some independent input. The third analysis is called *useful*, it computes the set of variables on which some dependent output possibly depends. The intersection of these two sets determine which variables are active.

- TBR [17, 47, 31]:

  In the forward sweep of the reverse mode variables may be overwritten, and this can happens several times, thus variables may hold different intermediate values. Some of these intermediate variables values may be required in order to compute the derivatives in the backward sweep. Unfortunately, the required intermediate values are not accessible when they are needed. In order to provide these required values, the TBR analysis determines which intermediate values must be stored in the tape during the forward sweep, thus making these values accessible to the derivatives in the backward sweep by restoring them from the tape. The TBR analysis is specific of the reverse mode of AD.

  TBR is composed of two steps. The first step is a bottom-up analysis that synthesizes two sets of variables, the variables that will be used in the adjoint code (**Req()**) and the variables killed in the forward sweep. The second step is a top-down analysis which using the previous two sets of variables, determines the variables values that must to be stored. When a required variable is overwritten, a stack management subroutine `PUSH` is inserted to push the value to the tape, and the overwritten variable is deleted from the set of required variables. The `PUSH` is inserted just before the instruction that overwrites the variable. In order to restore the values from the stack in the backward sweep, a stack management subroutine `POP` is inserted just after the instruction that implemented the derivative of the instruction that overwrites the variable.

  The general data-flow equations needed to compute the second step of TBR analysis for the basic block B of Figure 4.5 is defined as:

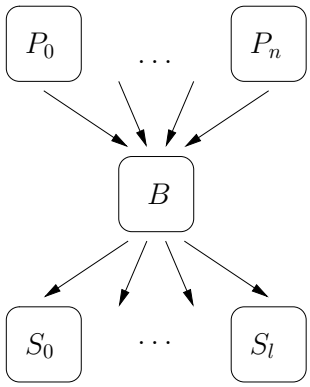


Figure 4.5: Predecessor (P_i) and successor (S_i) blocks of basic block B.

$$\mathbf{InReq(B)} = \bigcup_{i=0}^{n} \mathbf{OutReq}(P_i)$$

$$\mathbf{OutReq(B)} = (\mathbf{InReq(B)} \setminus \mathbf{Kill(B)}) \cup \mathbf{Req(B)} \tag{4.6}$$

where the first equation (**InReq**) accumulates the effect of all the predecessors of block B. The second equation (**OutReq**) takes into account that some variables are overwritten, when this happens instructions `PUSH` are inserted, and the overwritten variables become not required. It may happen that the variables becomes required later during the block due to the **Req** analysis. This happens if the variables are required by another instruction in the adjoint version of the block (in the following example of Table 4.4, this happens for instructions `i1, i2 and i5`).

The **Kill** analysis used in Formula 4.6 and in the rest of this work, is defined as the set of variables whose value are completely overwritten inside a segment of code. In general for two successive segments of code A and B we take the conservative under-approximation:

$$\mathbf{Kill}(A; B) = \mathbf{Kill}(A) \cup \mathbf{Kill}(B).$$

Recomputing versus Storing

Some values required by the backward sweep can be re-computed just by re-executing one or two original instructions. In that case to store the value when forward sweep, or to re-compute it in the backward sweep are both valid options to follow. But both require precise analysis in order to determine the fine trade-off.

Example Code	Data-flow Analysis Computation
`Block 1 (B1):`	InReq(`B1`) = {`x, y`}
`i1 x = x / y`	a `PUSH(x)` is inserted before `i1` in forward sweep, because variable `x` ∈ **InReq**(`B1`) and it is killed
`i2 y = 10 * x` `i3 w = COS(y)`	a `PUSH(y)` is inserted before `i2` in forward sweep, because variable `y` ∈ **InReq**(`B1`) and it is killed **Kill**(`B1`) = {`x, y, w`} **Req**(`B1`) = {`x, y`} ⇒ **OutReq**(`B1`) = ({`x, y`} \ {`w, x, y`}) ∪ {`x, y`} = {`x, y`}
`Block 2 (B2):`	**InReq**(`B2`) = **OutReq**(`B1`)
`i4 z = x * SIN(x)`	
`i5 y = x * z` `i6 w = w * COS(x)`	a `PUSH(y)` is inserted before `i5` in forward sweep, because variable `y` ∈ **InReq**(`B2`) and it is killed **Kill**(`B2`) = {`w, y, z`} **Req**(`B2`) = {`w, x, z`} ⇒ **OutReq**(`B2`) = ({`x, y`} \ {`w, y, z`}) ∪ {`w, x, z`} = {`w, x, z`}

Table 4.4: TBR analysis example.

4.2.2 Coarse-Grain Strategies

Checkpointing

The mechanism which deactivates the store-all strategy for certain chosen segments, is called *checkpointing*. Checkpointing exploits a trade-off between storing and re-computing. It has two consequences on the behavior of the reverse differentiated program:

1. when the forward sweep reaches a chosen segment (called *checkpointed segment*), a sufficient set of values (called a *snapshot*) must be stored. The snapshot allows to re-execute the checkpointed segment in the backward sweep with the correct context. During the forward sweep execution of a checkpointed segment, the store-all strategy is deactivated.

2. when the snapshot is restored during the backward sweep, the checkpointed segment is re-executed, but this time the store-all strategy is activated. As a result, the rest of the backward sweep is executed as usual.

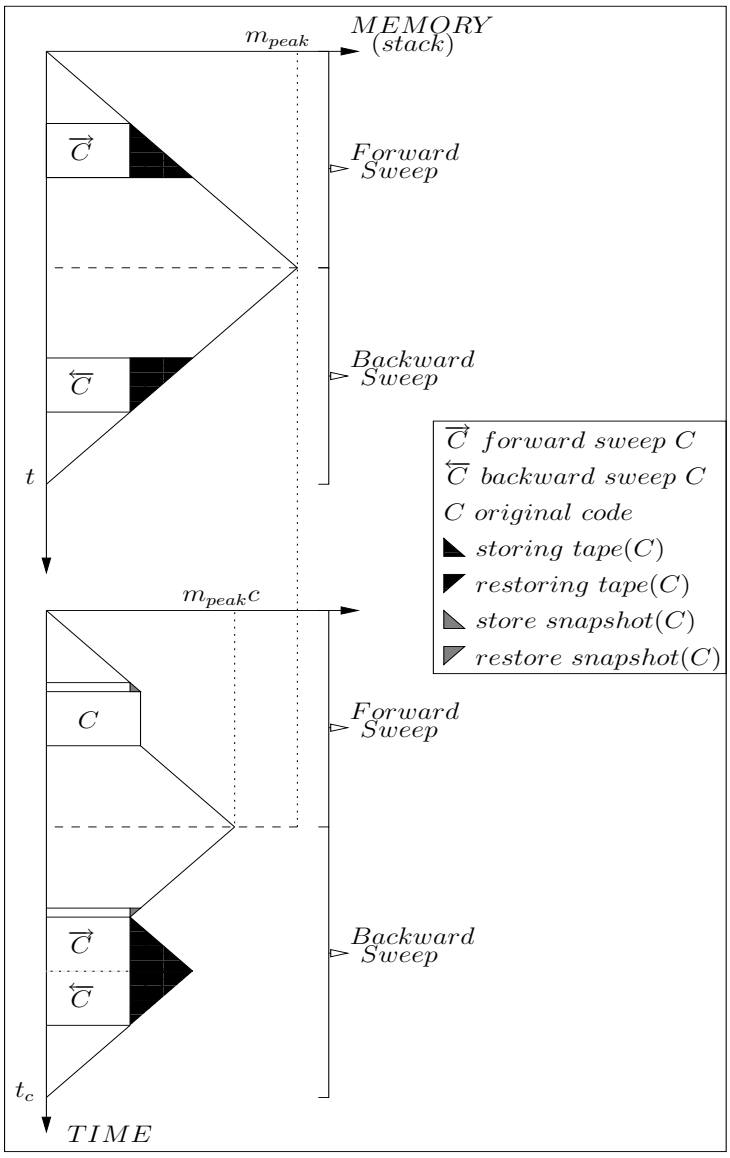


Figure 4.6: Checkpointing on Reverse Mode AD.

Recall that the set of variables values stored for a segment during the forward sweep (of SA strategy) is called the *tape*. The checkpointing mechanism is profitable if the size of the snapshot is smaller than the size of the tape for any checkpointed segment. Under this assumption the mechanism is profitable in memory space. This

is because storing the snapshot barely increases the size of the stack, thus the memory consumption peak of the reverse differentiated program with checkpointing is $(tape - snapshot)$ smaller than a reverse differentiated program without checkpointing. In contrast, the mechanism is not profitable in execution time, due to a double execution of the checkpointing segment.

In order to define a checkpointing strategy two elements have to be specified. The first element is the selection of checkpoint placements. The second element is to define which variables values should belongs to the snapshots.

Assumption Regarding the Tape and the Snapshot

As we mentioned above, we assume that the tape is bigger than the snapshot. This assumption is reasonable because in most cases the checkpointed segment and the size of its tape are large. This is because the size of the tape grows linear with respect to the number of instructions of the checkpointed segment. In contrast, the snapshot experiences a logarithmic rate of growth with respect the number of instructions. For instance, the set of input values for a large subroutine might be small. Finally, experimental observations confirm the assumption, these observations are part of the experiments of the end of the chapter.

For example in Figure 4.6, we assume that $tape(C) > snp(C)$. Consequently we see that $m_{peak}c$ is smaller than m_{peak}, because in the checkpointed case, $tape(C)$ is not required by the first forward sweep of segment C. Conversely, we see that t_c is larger than t, because in the no-checkpointing case the subroutines are executed only one time, and as we can observe in the checkpointing case the segment C is executed twice (C and $\overrightarrow{C}$).

Checkpoint Placements

The checkpointed segments can be placed in arbitrary ways, or through a systematic scheme, which is called a checkpoint strategy. In literature the Griewank [25] strategy is well known, this strategy is optimal, but only for a particular case (loop with fixed number of steps). In the general case, the runtime behavior of the program is unknown. For this general case the optimal checkpointing strategy has not been found yet.

Next, we present the two classical checkpointing strategies:

1. Optimal Checkpointing strategy for a fixed number of steps:

   This strategy fixes the size of snapshots, thus it is focused on deciding the optimal checkpoint placement, and the number of them. In order to accomplish

this, the strategy uses recursive functions that select which computational steps are suitable to be checkpointed. This strategy is also known as *binomial partitioning*. This is because the key point of the strategy is to split the range of steps, thus allowing recursion on the sub-ranges.

The following equations present an upper bound in execution time and memory consumption of the reverse mode computation under this checkpointing strategy:

$$T_d \equiv G + (d + r)R \tag{4.7}$$

$$W_t \equiv \hat{W} + \bar{W} + tW \tag{4.8}$$

Formula 4.7 is an upper bound of the size of the tape (memory consumption) used in the reverse mode. In this formula, d is the number of snapshots, r is the size of the steps, R is the size of the state of variables, and G is the maximal size of the state of variables. Basically this formula is composed of the multiplication of the depth of recursion $(d + r)$ and R.

Formula 4.8 is the upper bound of the execution of the reverse mode. In this formula t is the number of extra computational steps, $\hat{W}$ is the execution time needed to compute the forward sweep, $\bar{W}$ is the execution time needed to compute the backward sweep, and W is the execution time of the original program. Both spatial and temporal complexity order of magnitude are bounded by logarithmic expression. For example, if $d = r$ the complexity of the strategy is $log(T/rR)$, where T is the size of the tape. Also, an interesting trade-off is presented in [25] between the size of the snapshots and the number of checkpoints.

This strategy has been implemented by Griewank and Walther [29], but using simplified algorithms. This is because the original version was very expensive at the computational level (as it included too many recursions), and very particular about the kind of problems that could handle.

An promising extension to this strategy was introduced by Sternberg [57]. In that work, the number of steps is given but the size of the snapshots change depending on where the checkpoints are placed. The strategy uses two kind of snapshots, one called *thin*, the other called *fat*. Both kind of snapshots are related; the size of the fat snapshot is thrice the size of the thin snapshots. Authors introduce the idea of nesting checkpointing in their framework. Also, heuristics are presented to improve performance for certain cases of unknown runtime behavior. As a result, they achieved sub-optimal performance for those cases.

2. Checkpointing subroutines calls:

   A strategy that is more simple and easy to implement, although not optimal, is to systematically place the checkpoints before each subroutine call. This strategy is used in our AD tool TAPENADE. It is possible that some checkpoints are not necessary or of little use. This is one of the reasons why the strategy is not optimal. The other reason is that sometime the assumption (snapshot ¡ tape) behind the checkpointing strategy is not valid, as we will show in the experimental results.

   In the above strategy, the size of the snapshot is defined by an equation which relies on data-flow analysis, thus the size of the snapshot for each subroutine call is different. This is because it depends on the internals of the checkpointed subroutine.

Snapshot Definition

The snapshot is the minimal set of variables required to allow the checkpointed segment to be re-computed, thus producing the values which are required in the backward sweep. In order to determine the snapshot we introduce the following conservative definition, this definition is conservative because provides the required set of variables, but occasionally it may also include some not required variables, which, in any case, does not corrupt the computation, but it does not help the memory space performance of the program.

For an arbitrary program $P = S\ ;\ D$, where S is a subroutine, and the checkpoints are placed before the subroutine calls. The differentiated version of program P is as follows,

$$\overline{P} = \texttt{PUSH}(\mathbf{Snp}(S, D))\ ;\ S\ ;\ \overline{D}\ ;\ \texttt{POP}(\mathbf{Snp}(S, D))\ ;\ \overline{S} \tag{4.9}$$

The definition of the snapshot formula used in Formula 4.9 is:

$$\mathbf{Snp}(S, D) = \mathbf{Use}(S) \cap (\mathbf{Out}(S) \cup \mathbf{Out}(\overline{D})) \tag{4.10}$$

In Formula 4.10, the set $\mathbf{Use}(S)$ is some upper bound of the amount of information we need to store, because this set contains the values necessary for the backward sweep of S. But that set may include to many unnecessary variables values. Therefore, we can refine the snapshot formula by detecting the variables which are overwritten in S ($\mathbf{Out}(S)$), also detecting the variables overwritten in the downstream of $\overline{P}$ ($\mathbf{Out}(\overline{D})$).

Table 4.5 shows the computation of the snapshot for subroutine `SUB`. The local variable `u` of subroutine `SUB` is not considered in the computation of the analyses,

Example Code	**Snaphot Computation**

`SUBROUTINE EXAMPLE(x, y, z)`
 	
`w = COS(y)`
`z = w * SIN(x)`
`CALL SUB(w, x, y, z)`
`y = z * w`
`z = y**2 + x * COS(z)`
`END`
 	
`SUBROUTINE SUB(w, x, y, z)`
 	
`u = COS(w) + y + x`
`x = z / u`
`END`
	$\mathbf{Use}(\texttt{SUB}) = \{\texttt{w, x, y, z}\}$
	$\mathbf{Out}(\texttt{SUB}) = \{\texttt{x}\}$
	$\mathbf{Out}(\overline{D}) = \{\texttt{y, z}\}$
	
	$\mathbf{Snp}(\texttt{SUB}) = \{\texttt{w, x, y, z}\} \cap (\{\texttt{x}\} \cup \{\texttt{y, z}\})$
	$\mathbf{Snp}(\texttt{SUB}) = \{\texttt{w, x, y, z}\} \cap \{\texttt{x, y, z}\}$
	$\mathbf{Snp}(\texttt{SUB}) = \{\texttt{x, y, z}\}$

Table 4.5: Computation of snapshot example.

because for purpose of the snapshot computation this variable has no influence, due to its status of local variable of the checkpointed segment.

In the sequel of this section, we present the formulas of **Use** and **Out** analysis, both are required in Formula 4.10. Also, we present the **Live** analysis. These formulas are used in the rest of the chapter, in particular to derive some of the improved data-flow analysis. The following analyses are presented for successive segments of code, because in the sequel, i.e. the improved analyses, the analyses are also presented for that kind of code. The derivation of the general rules for these analyses is direct.

The set of variables whose value at the beginning of Z is read inside Z is denoted by $\mathbf{Use}(Z)$. For instance, for two successive segments of code A and B, the variables killed by A hide the variables read by B, so that:

$$\mathbf{Use}(A;B) = \mathbf{Use}(A) \cup (\mathbf{Use}(B) \setminus \mathbf{Kill}(A)). \tag{4.11}$$

The set of variables whose value at the beginning of the segment of code A is overwritten or partly overwritten inside A, during some possible execution of A, is denoted by $\mathbf{Out}(A)$. For example, for two successive segments of code A and B:

$$\mathbf{Out}(A;B) = \mathbf{Out}(A) \cup \mathbf{Out}(B). \tag{4.12}$$

If the **Out** analysis is applied to a segment of code when the stack is used, the variables values in the stack are not considered by the analysis. This is because from

the point of view of the analysis these variables are globally unmodified. The **Out** analysis is defined as follows:

$$\mathbf{Out}(\texttt{PUSH(v)};\ A;\ \texttt{POP(v)}) = \mathbf{Out}(A) \setminus \{\texttt{v}\}. \tag{4.13}$$

Finally, the **Live** analysis is defined as the set of live variables at the beginning of the tail segment of a program, where the variables values influence the results of the program. By definition all program results are live. We define $\mathbf{Live}([]) = \emptyset$. For two successive segments of code A and B, where B is the tail of the program, the live variables of segment B leads to live variables just before A through the dependence across A, defined in Formula 4.5. As a result, the formula **Live** for two successive segments is:

$$\mathbf{Live}(A; B) = \mathbf{Live}(B) \otimes \mathbf{Dep}(A) \tag{4.14}$$

In the next section, we use the presented analysis to state the formal model of the reverse mode of AD.

4.3 A Formal Model of Store-All Reverse Mode of AD

Using the above defined analysis we introduce an formal AD model for SA strategy. For a given program $P = I\ ;\ D$, where I is an instruction, and D represents the sequence of instructions called *downstream*, where this sequence of instructions goes from after instruction I to the end of the program. The reverse differentiated program $\overline{P}$ has the following form:

$$\overline{P} = \overline{I; D} = \overrightarrow{I}\,; \overline{D}; \overleftarrow{I} = \texttt{PUSH}(\mathbf{Out}(I))\ ;\ I\ ;\ \overline{D}\ ;\ \texttt{POP}(\mathbf{Out}(I))\ ;\ I' \tag{4.15}$$

The structure of program $\overline{P}$ is composed of two parts, the forward sweep:

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

and the backward sweep:

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

The model states that if a variable is overwritten by I, the value of the variable before being overwritten has to be stored in the stack. This set of variables is provided by the **Out** analysis. The action is carried out by `PUSH`. As a result, the value can be restored by `POP` in the backward sweep before being used by I'.

The model can be improved. In the next section, we present the first kind improvements. This is the improved data-flow analyses.

4.4 Contributions to the Fine-grain Strategies

The model described in Formula 4.15 can be improved. We have made three improvements which we discuss in this section. The first improvement is related to the fact that the results of instruction I in Formula 4.15 are only useful for $\overline{D}$. This is because if we have instructions before I, called *upstream* U, the backward sweep of these instructions $\overleftarrow{U;I}$, only require intermediate variables values created before I. As a result, if the results of I are not useful in $\overline{D}$, we can discard instruction I and the associated `PUSH/POP`. In order to detect this behavior we introduce the predicate *Adj-live*$(I, D) = (\mathbf{Out}(I) \cap \mathbf{Live}(\overline{D}) \neq \emptyset)$. Therefore if the predicate is false we can remove the instruction I. To compute the predicate we have first to compute the analysis **Out**, which is already presented, and $\mathbf{Live}(\overline{D})$ which is new. This new analysis is called *Adjoint Liveness*. This analysis can be considered as an particular kind of slicing, because starting from the end of a code segment, systematically discard the not needed instructions, thus it generates a sliced version of the differentiated program.

The second improvement is related to the TBR analysis. The idea is to refine the TBR analysis using the context information of the following differentiated instructions, which include the backward sweep of U, thus we must introduce U as a context into Formula (4.15). We use the notation $\vdash$ to separate U from the part of the program currently being differentiated. We introduce the set of variables used by instructions I' and after, which is $\mathbf{Use}(I'; \overleftarrow{U})$. Taking this into account, the only variables actually `PUSH`'ed and `POP`'ed for instruction I are now the set $(\mathbf{Out}(I) \cap \mathbf{Use}(I'; \overleftarrow{U}))$. Therefore, we have to define $\mathbf{Use}(I'; \overleftarrow{U})$

The last improvement is related to the activity analysis. If the activity analysis is carried out before the above two improvements, the above analyses compute smaller sets of intermediate variables. This is because the activity analysis may discard a good deal of instructions. For instance, some variables can be prove to have always a zero derivative with respect to the independent inputs or dependent outputs. When the variable written by assignment I is inactive, then I' can be removed. When some variable used by assignment I is inactive, I' is simplified.

In the remainder of this section, we discuss these improvements in more detail.

4.4.1 Adjoint Data-flow Analyses

In this section we provide the data-flow equations of three adjoint data-flow analyses. This analyses are called adjoint because they are focused on the differentiated instructions, thus looking to generate these kind of instructions at the cost of the fewer

possible original instructions. The first two adjoint data-flow analyses are introduced in the above section. The third adjoint data-flow analysis is basically a refinement of the **Out** analysis.

Adjoint Liveness Analysis

This analysis looks for the set $\mathbf{Live}(U \vdash \overline{I;D})$, with $\mathbf{Live}(\overline{[]}) = \emptyset$. The analysis computes the necessary differentiated variables. Both $\overline{D}$ and I' write differentiated variables. Therefore $\mathbf{Live}(U \vdash \overline{I;D})$ is the union of the necessary variables required for $\overline{D}$ and I'. The formula $\mathbf{Live}(\overline{I;D})$ is composed of two slices. The first slice does not depend of the context, and it states that the necessary variables due to I' are $\mathbf{Live}(I')$. The second slice is due to $\overline{D}$. This slice states that due the differential dependency across I, using Formula 4.14, the necessary variables due to $\overline{D}$ are $\mathbf{Live}(\overline{D}) \otimes \mathbf{Dep}(I)$. As a result, the composed formula is:

$$\mathbf{Live}(\overline{I;D}) = \mathbf{Live}(I') \cup (\mathbf{Live}(\overline{D}) \otimes \mathbf{Dep}(I)). \tag{4.16}$$

In order to avoid any circularity in the specification of the analysis, we run the Adjoint Liveness analysis that computes $\mathbf{Live}(\overline{Z})$, where $\overline{Z}$ is the reverse differentiated tail of the original program, before TBR analysis that computes $\mathbf{Use}(\overleftarrow{U})$. Formula (4.16) is extended to cope with basic blocks instead of instructions: for any block B followed by a downstream code D

$$\mathbf{Live}(\overline{B;D}) = \mathbf{Live}(\overline{B}) \cup (\mathbf{Live}(\overline{D}) \otimes \mathbf{Dep}(B)).$$

This backward data-flow equation is particularly efficient since $\mathbf{Live}(\overline{B})$ and $\mathbf{Dep}(B)$ can be precomputed.

Refined TBR Analysis

From the classical equation (4.11) of the **Use** analysis, we can write the rules that compute $\mathbf{Use}(I'; \overleftarrow{U})$ and $\mathbf{Use}(\overleftarrow{U})$, yielding a formal specification of the TBR analysis. Since I' only overwrites differentiated variables, and we study here the data-flow properties of the original variables only, $\mathbf{Kill}(I') = \emptyset$. Therefore

$$\mathbf{Use}(I'; \overleftarrow{U}) = \mathbf{Use}(I') \cup \mathbf{Use}(\overleftarrow{U}), \tag{4.17}$$

where $\mathbf{Use}(\overleftarrow{U})$ is defined recursively by:

$$\mathbf{Use}(\overleftarrow{[\,]}) = \mathbf{Use}([\,]) = \emptyset$$

$$\mathbf{Use}(\overleftarrow{U;I}) = \begin{cases} \mathbf{Use}(\texttt{POP}(\mathbf{Out}(I) \cap \mathbf{Use}(I'; \overleftarrow{U})); I'; \overleftarrow{U}) \\ \quad = (\mathbf{Use}(I') \cup \mathbf{Use}(\overleftarrow{U})) \setminus \mathbf{Kill}(I) \quad \textit{if adj-live}(I, D) \\ \\ \mathbf{Use}(I'; \overleftarrow{U}) = \mathbf{Use}(I') \cup \mathbf{Use}(\overleftarrow{U}) \qquad \textit{otherwise} \end{cases} \tag{4.18}$$

This formula runs forward on a flow-graph, or, in other words, this is a forward data-flow equation.

Adjoint Write Analysis

The **Out** must be adapted in order to be consistent with previous improved analyses. This analysis now has to take into account the context provided by the upstream segment of the program. Adjoint Write Analysis computes $\mathbf{Out}(U \vdash \overline{Z})$. If $Z = [\,]$, obviously $\mathbf{Out}(U \vdash [\,]) = \emptyset$. If $Z = I; D$, we distinguish two cases according to *adj-live*(I, D). We also use definition (4.13), i.e. a `PUSH/POP` pair on a variable leaves it unmodified by definition. As a result, we obtain:

$$\mathbf{Out}(U \vdash \overline{I; D}) = \begin{cases} (\mathbf{Out}(I) \cup \mathbf{Out}([U; I] \vdash \overline{D})) \setminus (\mathbf{Kill}(I) \cap \mathbf{Use}(I'; \overleftarrow{U})) \\ \qquad \textit{if adj-live}(I, D) \\ \mathbf{Out}([U; I] \vdash \overline{D}) \qquad \textit{otherwise.} \end{cases} \tag{4.19}$$

We see that $\mathbf{Out}(U \vdash \overline{I; D})$ is always included in $\mathbf{Out}(I; D)$, and this is often strictly due to the `PUSH/POP` pairs. Formula (4.19), as Formula 4.4.1, runs backward on a flow-graph.

4.4.2 Improved Model of the Store-All Reverse Mode of AD

As a result of the improved and new adjoint data-flow analyses the formal model changes to the following, more precise and complex model [33]:

$$\begin{aligned} U \vdash \overline{I; D} \;=\; & [\texttt{PUSH}(\mathbf{Out}(I) \cap \mathbf{Use}(I'; \overleftarrow{U})); I;] \;\; \textit{if adj-live}(I, D) \\ & [U; I] \vdash \overline{D}; \\ & [\texttt{POP}(\mathbf{Out}(I) \cap \mathbf{Use}(I'; \overleftarrow{U}));] \;\; \textit{if adj-live}(I, D) \\ & I' \end{aligned} \tag{4.20}$$

Depending on the predicate *adj-live*(I, D), the original instruction I can be discarded, subsequently the `PUSH/POP` can also be eliminated, no matter the result of the $\mathbf{Use}(I'; \overleftarrow{U})$ analysis. This is an important gain in memory, because the stack is not used. Furthermore, it also represents a gain in execution time. This is because the access time to the stack is not really negligible. If the predicate *adj-live*(I, D) is true, then the gain depends on the size of the set of values to store. This gain is the difference between $(\mathbf{Out}(I) \cap \mathbf{Use}(I'; \overleftarrow{U}))$ and $\mathbf{Out}(I)$, which used to be the set of variables to store in Formula 4.15.

4.4.3 Experimental Measurements

We measured the benefits of Adjoint Liveness and Adjoint Write analyses on three large applications that we use as validation tests. The results strongly depend on the actual codes, thus Table 4.6 shows the general description of the scientific codes used in our experiments. Notice that the same codes are used at the end of the chapter, where they are tested using different checkpointing strategies.

Code name	Number of lines	Application domain
STICS	27 000	Agronomy
UNS2D	2 700	CFD
SONICBOOM	21 000	CFD

Table 4.6: Scientific codes description.

We can observe in table 4.7, that the results are positive in almost all the experiments. This is because the speedup ranges between 7% and 18%, and the improvement in memory between 0% and 49%. This last result requires extra explanation. The `STICS` code is so large that it makes a heavy use of the swap space in the reverse mode. This causes the huge slowdown of the reverse mode. Therefore, it is even more important to spare 49% in memory, which we achieve thanks to the Adjoint Write analysis.

Experiment	Time		Memory	
Description	Total [s]	**% gain**	Peak [Mb]	**% gain**
Adjoint program STICS[1]	42.60		456	
[1] + Adjoint data-flow analysis	35.70	16	230	49
Adjoint program UNS2D [2]	29.70		260	
[2] + Adjoint data-flow analysis	24.78	16	259	0
Adjoint program SONICBOOM[3]	5.65		10.9	
[3] + Adjoint data-flow analysis	4.62	18	9.4	14

Table 4.7: Time and memory improvements on three large scientific codes.

In Table 4.7 we compare execution times of the reverse differentiated programs, and the improved reverse differentiated programs (both using TBR analysis). The improved version includes the adjoint liveness and adjoint write analysis. We compare the memory consumption under the same set-up. The memory consumption is measured as the maximum size of the tape.

4.5 Contributions to Coarse-grain Strategies

4.5.1 A Formal Model of Store-All Reverse Mode AD with Checkpointing

We are interested in studying the call-graph of the reverse differentiated codes. This is because the call-graph is the most handy way to analyze a program with checkpointed segments of code, specifically when the checkpoints are placed before subroutine calls. We have to extend the formulas of the previous sections in order to capture the influence of these analyses of the call-graph.

Let us now consider the case where the program P contains checkpointed segments. Thus, for an arbitrary program $P = S\ ;\ D$, where the checkpointed segment is a subroutine S. Usually, the predicate *adj-live*(S, D) is true, thus simplifying the notation and losing no generality. As a result, for the special case of the checkpointed segment S, the reverse AD model of Formula 4.20 is replaced by:

$$\begin{aligned} U \vdash \overline{S; D} \ = \ & \texttt{PUSH}(\mathbf{Out}(S) \cap \mathbf{Use}(\overleftarrow{U})); \\ & \texttt{PUSH}(\mathbf{Snp}(U, S, D)); \\ & S; \\ & [U; S] \vdash \overline{D}; \\ & \texttt{POP}(\mathbf{Snp}(U, S, D)); \\ & [\,] \vdash \overline{S}; \\ & \texttt{POP}(\mathbf{Out}(S) \cap \mathbf{Use}(\overleftarrow{U})); \end{aligned} \tag{4.21}$$

A trade-off is presented between the set computed by $\mathbf{Snp}(U, S, D)$ and the set $\mathbf{Out}(S) \cap \mathbf{Use}(\overleftarrow{U})$, which basically is due to the context added by U. For example, putting U instead of [] as the context for the generation of $\overline{S}$ will cost more `PUSH/POP` inside $\overline{S}$, and on the other hand, storing $\mathbf{Out}(S) \cap \mathbf{Use}(\overleftarrow{U})$ becomes unnecessary in (4.21). Exploration of this trade-off is an open problem [16].

The AD model with checkpointing defined in Formula 4.21 is more efficient than the model of Formula 4.9. This is because it takes advantage of the adjoint data-flow analyses, who allow to define a better snapshot formula, as we present in the next section.

4.5.2 Improved Snapshot Definition

In comparison with Formula 4.10, we mainly improve that formula by adding the context information of U which is used by the adjoint data-flow analyses. First, the set of variables $\mathbf{Live}(\overline{S})$, required to run $\overline{S}$, is smaller than $\mathbf{Use}(S)$. Second, we need to restore a variable only if it was modified "in between," i.e. is in the **Out** set of code

sequence $S; \overline{D}$. We take advantage of $\mathbf{Out}(\overline{D})$ being smaller than $\mathbf{Out}(D)$. Therefore, when the predicate *adj-live(S, D)* is true, we define the snapshot as:

$$\mathbf{Snp}(U, S, D) = \mathbf{Live}(\overline{S}) \cap (\mathbf{Out}(S) \cup \mathbf{Out}([U; S] \vdash \overline{D})) \tag{4.22}$$

The set of variables to store by the improved definition of snapshot is smaller than the one computed by Formula 4.10. This because snapshot definition of Formula 4.22 uses the adjoint liveness analysis, which computes a smaller set of variable vales than the set used in Formula 4.10. Also, in Formula 4.22 the set **Out** is computed taking into account the context $[U; S]$. This implies the possibility of reducing the computed set.

4.5.3 Example

The reverse differentiated code of Table 4.8 was obtained (from the original code of Table 4.5) using the classical model of reverse mode of AD. All commented lines are not necessary, this is determine by the improved formal model of Store-All reverse mode of AD with checkpointing. Therefore, in order to determine these improvements, the adjoint data-flow analyses, as well as the improved snapshot formula were used.

In Table 4.8, due to Adjoint Liveness Analysis the last instruction of the forward sweep of subroutines `EXAMPLE_B` and `SUB_B` is discarded, thus allowing to spare the pair `PUSH/POP` that the TBR analysis has inserted. Thanks to the Adjoint Write Analysis and the Adjoint Liveness Analysis we are able to compute a smaller snapshot, thus variable `z` is not anymore stored. Notice that depending on the context of the analyses, the variable `y` may not be part of the **Snp(**`SUB`**)**, this is correct as long as TBR analysis take this into account and insert a `PUSH` just after subroutine `SUB` and before the instruction that overwrites `y`. This alternative is valid because variable `y` $\notin$ **Out(**`SUB`**)**, and the required value of this variable in `SUB_B` will be restored just before the subroutine call.

4.5.4 The Systematic Checkpointing

Checkpointed segments can be chosen in various manners, and can be nested. In AD tools, checkpointing is applied systematically, for instance at subroutine calls or around loop bodies.

Figure 4.7 central section shows the call-graph of a reverse differentiated program using the joint-all mode, where joint-all is the checkpointing strategy of checkpoint each subroutine call. Experience indicates that the joint-all strategy is not optimal, though the optimal situation is not easy to foresee.

```
   SUBROUTINE EXAMPLE_B(x, xb, y, yb, z, zb)

   w = COS(y)
   z = w * SIN(x)
C  CALL PUSHREAL8(z)
   CALL PUSHREAL8(y)
   CALL PUSHREAL8(x)
   CALL SUB(w, x, y, z)
   y = z * w
C  CALL PUSHREAL8(z)
C  z = y**2 + x * COS(z)
   <forward sweep ends, backward sweep begins>
C  CALL POPREAL8(z)
   yb = 2 * y * zb
   xb = COS(z) * zb
   zb = w * yb - x * SIN(z) * zb
   wb = z * yb
   CALL POPREAL8(x)
   CALL POPREAL8(y)
C  CALL POPREAL8(z)
   CALL SUB_B(w, wb, x, xb, y, yb, z, zb)
   wb = wb + SIN(x) * zb
   xb = xb + w * COS(x) * zb
   yb = yb - SIN(y) * wb
   END

   SUBROUTINE SUB_B(w, wb, x, xb, y, yb, z, zb)

   u = COS(w) + y + x
C  CALL PUSHREAL8(x)
C  x = z / u
   <forward sweep ends, backward sweep begins>
C  CALL POPREAL8(x)
   zb = zb + xb / u
   ub = - z * xb / u**2
   wb = wb - SIN(w) * ub
   yb = ub
   xb = ub
   END
```

Table 4.8: Reverse differentiated version of example code of Table 4.5

Before we go further in our analysis, it is important to formalize the process of obtaining the call-graph of differentiated programs. This formalization is important because we use the differentiated call-graph for further applications of the snapshot formula. The formalization is resumed in Table 4.9.

One can think of deactivate the checkpointing mechanism for certain segments of

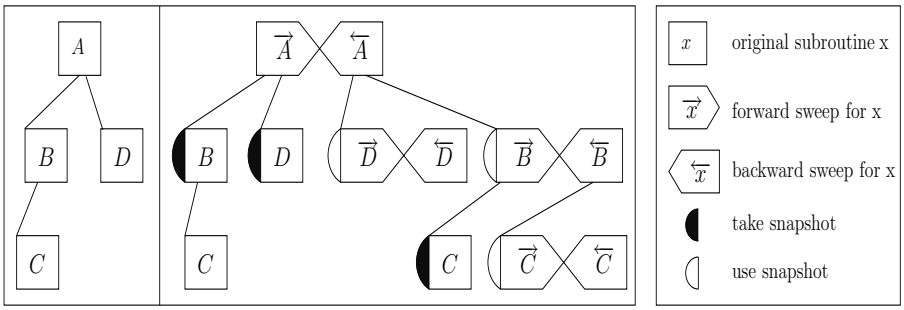


Figure 4.7: Checkpointing on all calls in Reverse Mode AD (joint-all mode).

Symbols	
S = subroutine	– = a call
• = memory write	∘ = memory read
T = tree of subroutines	$T^\star$ = multiple tree
$S^{\rightarrow}$ = forward sweep of S	$^{\leftarrow}$S = backward sweep of S

Rules	
Rule	Description
RAD(S) = $S^{\rightarrow}\ {}^{\leftarrow}S$	Reverse A.D. over S
RAD(S-T^*) = $S^{\rightarrow} - TS(T^*)\ {}^{\leftarrow}S - RADS(T^*)$	RAD over S call T^*
RADS(()) = ()	RAD with Snapshot over ()
RADS(S) = $\circ S^{\rightarrow}\ {}^{\leftarrow}S$	RADS over S
RADS(TT^*) = RADS(T^*)RADS(T)	RADS over a T and T^*
RADS(S-T^*) = $\circ S^{\rightarrow} - TS(T^*)\ {}^{\leftarrow}S - RADS(T^*)$	RADS over S call T^*
TS(()) = ()	Take Snapshot over ()
TS(S) = $\bullet S$	TS over S
TS(TT^*) = TS(T)TS(T^*)	TS over a T and T^*
TS(S-T^*) = $\bullet S - T^*$	TS over S call T^*

Table 4.9: From original call-graph to reverse differentiated with checkpointing call-graph.

code, this is called *split mode*. But that feature was not part of the tool at hand, so we added it to the tool. In split mode the forward sweep and the backward sweep are implemented separately, and do not follow each other during execution. Therefore, no snapshot is required, but this forces to store more intermediate values in the tape. This is because the local variables values of the forward sweep of the no-checkpointed segment are unreachable for the corresponding backward sweep. As a result, they have to be stored. Figure 4.8 shows the other classical alternative, which is no-checkpoint for each subroutine, this alternative is called the *Split-All* strategy.

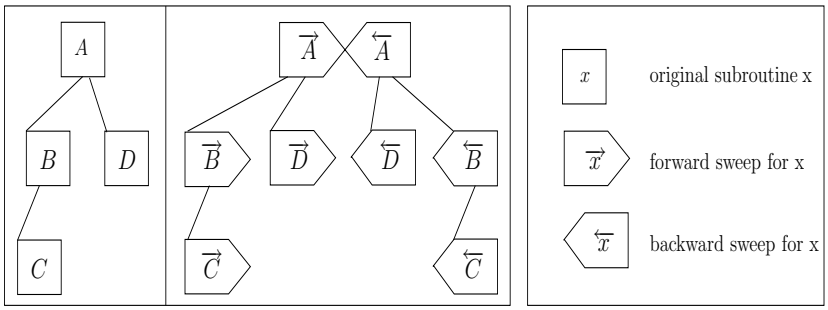


Figure 4.8: No Checkpointing in Reverse Mode AD (split-all mode).

The advantage of split mode is that the subroutines are executed just one time (Figure 4.8), then savings in execution time are important.

Split-all and joint-all are two extreme strategies. It is worth trying hybrid cases, and we present a couple of cases in Figure 4.9. The first strategy *hybrid1*, implements the joint mode for all subroutines except for subroutine D.

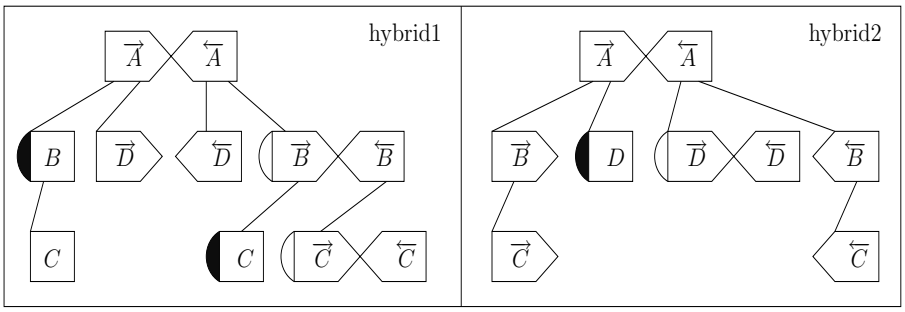


Figure 4.9: Hybrid approach (split-joint)

Conversely, the second strategy: *hybrid2*, implements the split mode for all subroutines except for subroutine D, which is checkpointed.

Simulation of Hybrid Strategies Assuming *Snapshot* $<$ *tape*

In order to have a more precise idea of the aforementioned trade-off we have to simulate the performance of mentioned cases for two motivating scenarios. We assume that all the subroutines have the same snapshot size, and have the same tape size, but snapshot and tape have different sizes. Also, we assume that each subroutine has the same execution time. The original call-graph is the one given in the left section of Figure 4.7 and the differentiated ones are given in Figures 4.7, 4.8, 4.9.

For the first scenario, we set the memory size of the snapshot to 6 and the memory size of the tape to 10. This setup corresponds to the usual assumption that the tape is bigger than the snapshot for subroutines.

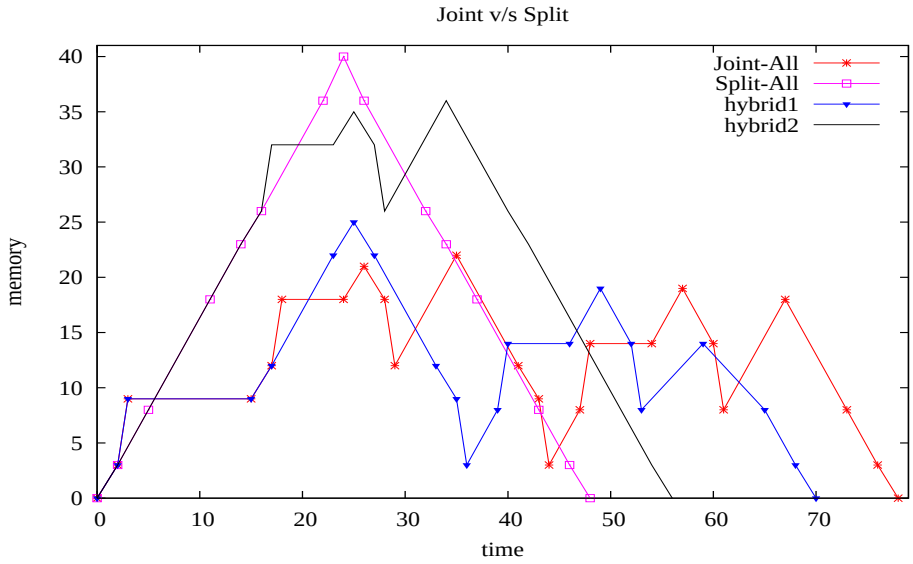


Figure 4.10: Simulation results, tape = 10, snapshot = 6.

Figure 4.10 shows the behavior of the four checkpointing strategies previously mentioned. As we expected, the curve that represents the joint configuration shows the fewest consumption of memory but the largest execution time. Conversely, the curve that represents the split mode has the highest peak of memory consumption but the shortest execution time. Hybrid strategies range between these two extremes.

Simulation of Hybrid Strategies Assuming *Snapshot > tape*

This previous scenario assumed that the tape is bigger than the snapshot. However, this assumption is not always valid. Therefore, we present a second simulation where we assume that the tape cost 6 in memory, and the snapshot costs more, e.g. 10.

Figure 4.11 shows that joint and split modes are not the extreme of the trade-off anymore. In fact, the extreme bounds in memory consumption correspond to the hybrid modes. Another interesting fact from the second simulation is that the maximum peak of memory consumption is smaller than the one of the first simulation. This is not surprising as the snapshot is bigger than the tape but far less used. In this scenario, the advantage of checkpointing is less obvious because of the costs of

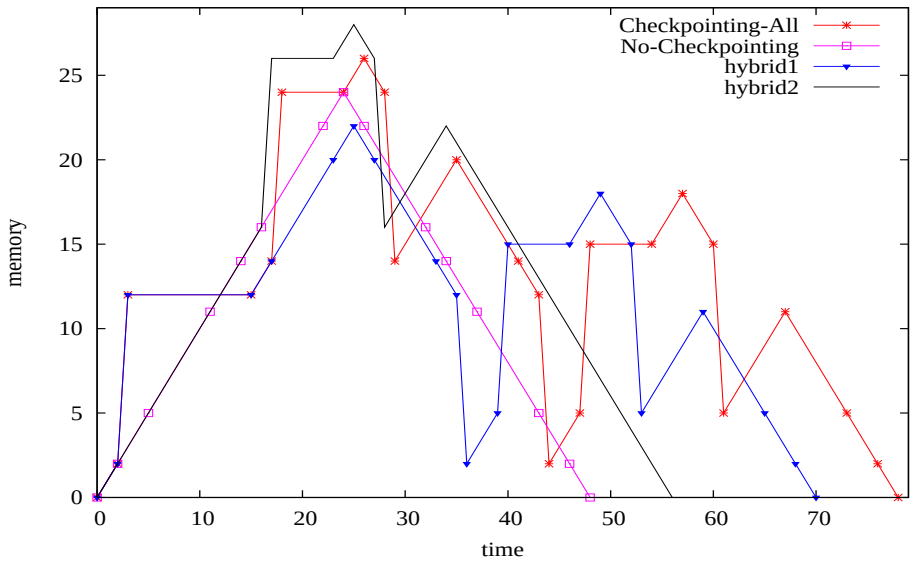


Figure 4.11: Generic Numerical Results, tape = 6, snapshot = 10.

snapshots, therefore the split-all mode is nearly the best in every respect.

We have shown in Figure 4.10 and Figure 4.11 that the hybrid strategies are appealing alternatives to the basic strategies. It would be very useful to carry some experiments with the basic and hybrid strategies. This will give us an idea of the best strategies regarding industrial size code. We consider this experimentation the first step to an optimal checkpointing strategy.

4.5.5 Implementation

Along with the modification of the analyses, the generation of the differentiated program must also be adapted. The AD model defined in Section 4.5.1 shows that the joint mode runs the backward sweep of C immediately after the forward sweep C. When C is a subroutine, the subroutines $\overrightarrow{C}$ and $\overleftarrow{C}$ can be easily be merged into a single subroutine $\bar{C}$. As a consequence, local variables of C (and of $\overrightarrow{C}$) are still in scope when $\overleftarrow{C}$ starts, and therefore preserve their values.

Implementation of the Split Mode in tapenade

The mentioned feature is no longer possible in split mode, since subroutine $\overrightarrow{C}$ and $\overleftarrow{C}$ must be separated. Consequently, local variables of $\overrightarrow{C}$ must be stored before they vanish and restored when $\overleftarrow{C}$ starts. This was addressed in the implementation by adding a new analysis. This analysis finds the local variables that are necessary for

the backward sweep, so that they are `PUSH`'ed to the end of the forward sweep and `POP`'ed at the beginning of the backward sweep.

We make the choice of generalization versus specialization, by allowing for only one split mode per subroutine. Even then, this requires care in naming the subroutines. We need to create up to four names (original, forward sweep, backward sweep and reverse differentiated) when split and joint strategies are combined. This problem is technical, but it has implications within the whole way that TAPENADE handles differentiated elements.

The split strategy is driven by the user by means of a directive (`C$AD NOCHECKPOINT`) which is placed just before the subroutine call, or through a command line option (`-split "[name of subroutines]"`). The introduction of directives is a novel feature for TAPENADE.

We give a tutorial on checkpointing and AD tool TAPENADE in Appendix A.1. The tutorial includes snippets of code from our pool of validation codes. Some of these codes are used in the next section.

4.5.6 Experimental Observation of Problems and Results

Example Codes

We applied the split mode to certain subroutine calls, looking for experimental confirmation of the intuitions from Section 4.5. In particular, we want to show the interest of letting the user drive the checkpointing strategy.

The subroutines chosen to be split were the ones that best illustrate the memory and run-time trade-off. The criteria to choose subroutines rely on two values, which can be obtained by studying the reverse generated code. These values are: the size of the snapshot and the size of the tape. The implementation of both values is based on `PUSH` calls, thus the comparison between these values is straightforward. For instance, we can observe Table 4.11 (page 96).

STICS

STICS is an agronomy modeling program. It has 21.010 lines of code (LOC), and 46.921 LOC were generated to implement the reverse differentiated subroutines.

In STICS code, we introduce three levels of nested loops around subroutine ONEBIGLOOP because this code simulates an unsteady process over 400 time steps. These nested loops are a manual modification that allow us to perform checkpointing

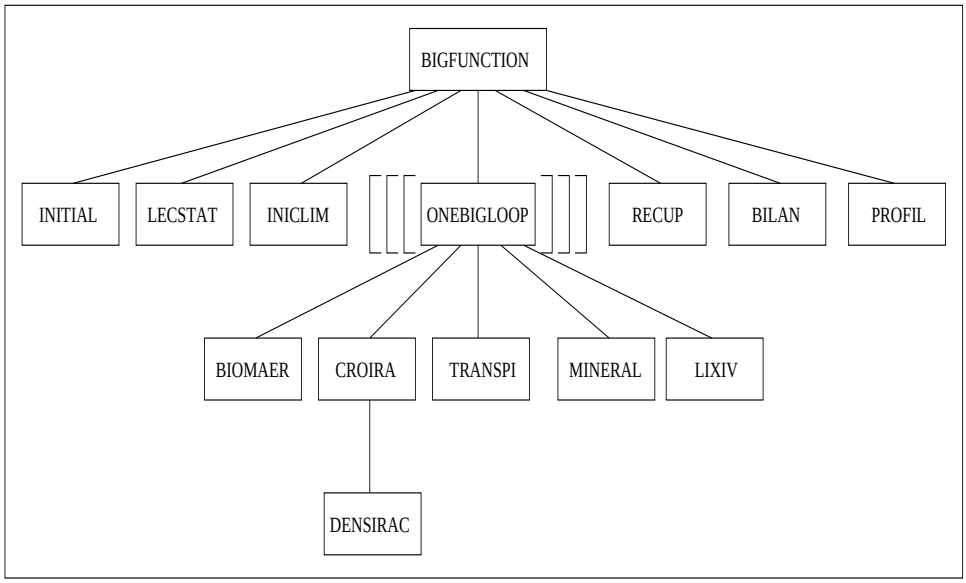


Figure 4.12: STICS Call-graph.

on various groups of time steps.

The performance of STICS is depicted in Table 4.10, specifically in execution time, shows how inefficient the joint-all strategy can be. In this case, the re-execution of the subroutines due to the checkpointing mechanism is the main responsible for this performance drop. This fact confirms our statement about the sub-optimality of joint-all strategy.

For this experiment, the default (Split-All) strategy applied by TAPENADE gave very bad results in time, with a slowdown factor of about 100 from the original code to the reverse differentiated code. We made some measurements of the tape sizes compared to the snapshot sizes, and we found out that tape was much smaller than snapshot for subroutines DENSIRAC, CROIRA and ONEBIGLOOP. This is a special case of the situation of Figure 4.11 and is reflected on the experimental figures of Table 4.10. We see that split mode on these three subroutines gain execution time at no memory cost. Combined split mode on the three subroutines (experiment 09) gives an even better result.

The enormous gain in execution time makes the differentiated/original ratio go down to about 7, which is what AD tools generally claim. In the STICS experiment, the execution time of the Split-All version did not come from the duplicate executions due to checkpointing but rather from the time needed to `PUSH` and `POP` these very large snapshots. This suggests that a complete model to study optimal checkpointing

Experiment		Time		Memory	
Id	Description	Total [s]	**% gain**	Peak [Mb]	**% gain**
01	Joint-All strategy	38.56		229.23	
02	split mode BIOMAER	36.15	6.3	229.23	0.0
03	split mode MINERAL	35.78	7.2	229.28	0.0
04	split mode DENSIRAC	30.02	22.1	229.23	0.0
05	split mode CROIRA	24.45	36.6	229.23	0.0
06	split mode ONEBIGLOOP	23.75	38.4	229.75	-0.2
07	04 and 05	16.79	56.5	229.23	0.0
08	04 and 06	15.64	59.4	229.75	-0.2
08	05 and 06	11.71	69.6	206.81	9.8
09	04, 05 and 06	3.93	89.8	149.11	34.9
09	03, 04, 05 and 06	3.92	89.8	149.11	34.9
09	split all the above subroutines	3.90	89.9	149.11	34.9

Table 4.10: Memory and time performance for STICS.

strategies should definitely take into account the time spent for tape and snapshots operations.

Practically, in the STICS example there is no doubt DENSIRAC, CROIRA and ONEBIGLOOP should be differentiated in split mode. In addition, one can differentiate additional subroutines in split mode, (e.g. MINERAL), but the additional execution time gain is marginal.

UNS2D

UNS2D is a CFD solver. It has 2.055 LOC. The reverse differentiated version has 2.200 LOC.

On figures 4.13 and 4.12, loops are denoted by square brackets. For instance, on Figure 4.13 we have two loops, one which involves from subroutine PASDTL to subroutine QUAIND, and a second one which includes all INBIGFUNC's subroutines. These loops are the segment of the program that consumes most of memory and time.

In Table 4.11 we can observe the required information to determine the checkpoints placements. This is because we can use these information to decide for each subroutine its checkpoint status. The profiling information was obtained using an extension of TAPENADE, which generates differentiated code with the necessary instructions to carry out the computation of the profiling information in runtime.

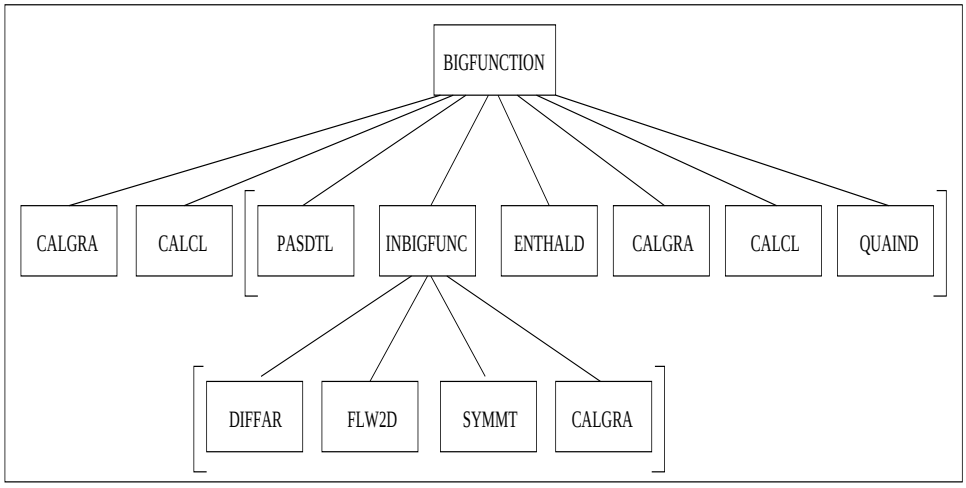


Figure 4.13: UNS2D Call-graph.

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

Table 4.11: Profiling information about snapshots, tapes and number of calls to subroutines of UNS2D.

The first four experiments 02 - 05 of Table 4.12 report gain both in time and memory, reminding us of the case where $tape < snp$ (Figure 4.11). This is indeed what we observe when we measure the actual sizes of tape and snapshot for the subroutines in question. Therefore, when each of CALGRA/CALCL/QUAIND ENTHALD are split the program saves memory for the snapshot without using as much for the tape. At the same time it saves time because the subroutine is not executed twice. Notice that the case of ENTHALD is somewhat different, the research is open for this case.

Experiment		Time		Memory	
Id	Description	Total [s]	**% gain**	Peak [Mb]	**% gain**
01	Joint-All strategy	41.69		184.69	
02	split mode CALCL on all call placements	37.66	9.7	167.53	9.3
03	split mode QUAIND	37.54	9.9	162.13	12.2
04	split mode CALGRA on all call placements	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 and 05	33.95	18.6	163.20	11.6
08	03 and 06	31.75	23.8	446.82	-141.9
09	02, 04 and 05	35.81	14.1	174.45	5.5
10	02, 05 and 06	35.49	14.8	533.23	-188.7
11	02, 03, 04 and 05	38.50	7.6	184.45	0.13
12	02, 04, 05 and 06	30.92	25.8	408.88	-121.4
13	split mode on all above subroutines	32.67	21.6	443.56	-140.2

Table 4.12: Memory and time performance for UNS2D.

Experiment 06 exhibits a gain in time at the cost of a larger memory use. As we suspected from the simulations on Figure 4.10, this corresponds to the case where $snp < tape$. This is also what people had in mind when checkpointing was first proposed, and in this situation checkpointing is really a time/memory trade-off. Therefore checkpointing INBIGFUNC (in other words the joint mode) is a wise choice when memory size is limited.

Experiments 07 - 13 can be separated in two sets: whether INBIGFUNC is checkpointed (08, 10, 12 and 13) or not (07, 09 and 11). The separation criterion underlines the relative weight of the subroutine INBIGFUNC.

Experiments 07, 09 and 11 shows a remarkable behavior on the execution time performance. The execution time savings of combined split mode subroutines should accumulate, regarding what we observe in Figures 4.10 and 4.11, but surprising the execution time performance for these experiments show the opposite. In particular, the experiment 11's execution time saving (3.18s) is smaller than the execution time savings (4.03s, 4.15s, 5.03s and 7.36s) of everyone of the subroutines which compose the experiment itself. This behavior lead us to do further experimentation and analysis in order to make it consistent with the checkpointing model described.

Finally, we advice to use the split mode strategy given by the experiment 12 in case of execution time savings demands. On the other hand, experiments 03 and 05 allow memory savings up to 12%.

SONICBOOM

SONICBOOM is a part of a CFD solver which computes the residual of a state equation. It has 14.263 LOC, but only 818 LOC to be differentiated, generating 2.987 LOC of derivative subroutines.

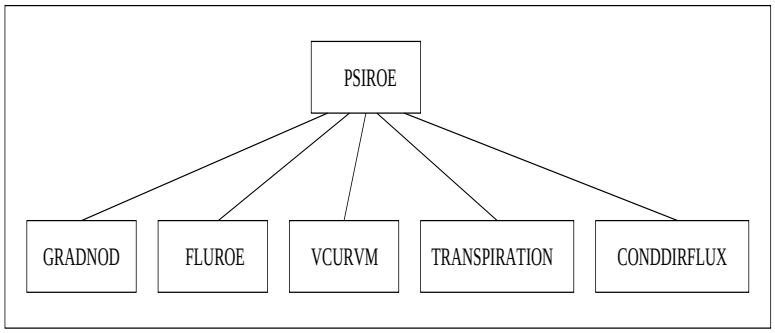


Figure 4.14: SONICBOOM Call-graph.

The first group of experiments, 02 - 04 from Table 4.13, shows gain in execution time, because the subroutines are executed only one time. There is no gain in memory because the size of the snapshot and the tape are very close.

The experiments where GRADNOD is involved exhibit the largest gain in execution time. This is related to the fact that GRADNOD accounts for the largest part of the computation, and since the tape size grows in step with the number of executed instructions, *tape*(GRADNOD) is much larger than *snp*(GRADNOD). For the other subroutines in this experiment, we also have $tape < snp$, but to a smaller extent. Therefore, everything behaves like in the classical case of Figure 4.10. In particular, there is no subroutine for which the split mode would give a gain in a memory consumption.

It is worth noticing that the effect of the split mode is really an increase in memory traffic rather than in memory peak size. For example splitting CONDDIRFLUX certainly results in a higher memory traffic, but the increase of the local memory peak is hidden by the main memory peak which occurs just after $\overrightarrow{\text{GRADNOD}}$. We are currently carrying new experiments and developing refined models that include this memory traffic.

Experiment		Time		Memory	
Id	Description	Total [s]	**% gain**	Peak [Mb]	**% gain**
01	Joint-All strategy	0.2900		10.84	
02	split mode VCURNVM	0.2725	6.0	10.84	0.0
03	split mode CONDDIRFLUX	0.2699	6.9	10.84	0.0
04	split mode FLUROE	0.2500	13.8	11.06	-2.0
05	split mode GRADNOD	0.2374	18.1	18.77	-73.1
06	02 and 03	0.2624	9.5	10.84	0.0
07	04 and 05	0.2374	18.1	19.00	-75.2
08	02, 03 and 04	0.2475	14.7	11.08	-2.2
09	02, 03 and 05	0.2360	18.6	18.77	-73.1
10	split mode on all the above subroutines	0.2374	18.1	19.00	-75.2

Table 4.13: Memory and time performance for SONICBOOM.

Practically for this experiment, our advice would be to run subroutines FLUROE, VCURVM and CONDDIRFLUX (experiment 08) in split mode in any case, and this already gives a 14.7% improvement in time at virtually no cost in memory. In the case where memory size is not strictly limited, then it is advisable to run GRADNOD in split mode too, which gives an additional gain in time at the cost of a large increase in memory peak.

4.5.7 Discussion

Related works on optimal checkpointing have been conducted mostly on the model case of loops of fixed-size iterations. Only in the particular sub-case where the number of iterations in known in advance was an optimal scheme found mathematically [25]. This leads to the treeverse/revolve [29] tool for an automatic application of this scheme. We are not aware of optimal checkpointing schemes for the case of an arbitrary call-tree or call-graph.

Notice that checkpointing is not the only way to improve the performance of the reverse mode of AD. Local optimizations can reduce the computation cost of the derivatives by re-ordering the sub-expressions inside derivatives [2]. Other optimizations implement a fine-grain time/memory trade-off by storing expensive sub-expressions that occur several times in the derivatives.

Local optimizations only give a fixed small benefit. For large programs, only nested checkpointing can make reverse differentiated codes actually run without exceeding the memory capacity of the machine. Therefore the study of optimal checkpointing

schemes is an absolute necessity.

4.6 Conclusions and Future Work

The reverse differentiated structure of a program is composed of two main parts, the first part is called the forward sweep and the second part is the called backward sweep. The forward sweep has basically the same instructions of the original program, and it computes the intermediate variables values required in the backward sweep. The backward sweep implements the derivatives. During the execution of the forward sweep, variables are re-defined, thus changing intermediate values. The problem arises when the intermediate values are required by the derivatives, but they are not anymore accessible because they have been overwritten. This is the source of the problem we have addressed in this chapter.

To cope with this problem we have two main strategies. We follow the strategy called store-all, which consists in storing the required values during the forward sweep, in order to restore them in the backward sweep. This solution has a problem: the amount of memory needed to store the values can be unacceptably high.

We aimed to refine our AD reverse mode model to generates the best code possible in terms of memory consumption, without diminishing the efficiency in execution time. To achieve this, we have two main lines of action. The first consists of local optimization of the generated code. To achieve this optimization we improve the existing data-flow analyses, thus introducing what we called adjoint data-flow analyses.

We have described our special-purpose data-flow analyses that are used to improve the performance of the produced codes. The experimental results show that the improvements successfully reduce the peak of memory consumption up to 49%. Also, the execution time is reduced on average up to 16%.

The second line of action in which we worked takes advantage of the trade-off between storage and re-computation, at the level of segments of code. Therefore, this kind of optimization can seen as global. The main technique of this kind of optimization is the mechanism called checkpointing, which spares time or memory depending on the trade between storage and re-computation.

We started from the observation that the strategy consisting in checkpointing each and every subroutine call is in general, although safe from the memory point of view, far from optimal. Both simulations on very small examples, and real experiments on real-life programs show that some subroutines should never be checkpointed, and that others may be checkpointed depending on the available memory.

The great variety of possible situations makes the objective of automatic selection of checkpointing sites very difficult to achieve. It seems therefore reasonable to let the user drive this choice through an adapted user interface. We discussed the developments that we made into the AD tool TAPENADE to add this functionality. This new functionally allowed us to conduct extensive experiments on real codes. This validate our hypotheses on the optimal checkpointing problem. Furthermore, the results suggests the relevant criteria for a future helping tool. These are for each subroutine, its execution time, its tape and snapshot sizes, and the time required by tape `PUSH` and `POP` traffic.

In Table 4.14 we combine the results of both kind of optimizations. Due to the fact that the results were obtained in two different platform, a post-process was required to compose the following table.

Experiment	Time		Memory	
Description	Total [s]	**% gain**	Peak [Mb]	**% gain**
Adjoint program STICS (AP1)	42.60		456	
AP1 + adjoint data-flow analysis (ADF)	35.70	16	230	49
AP1+ ADF + hybrid strategy 09	3.55	90	149	35
Adjoint program UNS2D (AP2)	29.70		260	
AP2 + adjoint data-flow analysis	24.78	16	259	0
AP2 + ADF + hybrid strategy 05	20.32	18	227	12
Adjoint program SONICBOOM (AP3)	5.65		10.9	
AP3 + after adjoint data-flow analysis	4.62	18	9.4	14
AP3 + ADF + hybrid strategy 08	3.97	14	9.6	-2

Table 4.14: Time and memory improvements on three large scientific codes.

In Table 4.14 the mentioned hybrid strategies correspond to the ones in Table 4.10, Table 4.12 and Table 4.13. We can observe in Table 4.14 the improvements of every global strategy (fine-grain and coarse-grain) for our test codes. The combined results are very positive, the most interesting case is STICS, where the gains are on both memory and time are quite promising.

The results are very encouraging, thus the next step is to look for an automatic way of discovering the optimal checkpointing strategy. An automatic strategy to place the checkpoints could be based on execution time profiling of the original program or even of the differentiated code itself. This suggests a process of iterative improvements of the reverse differentiated codes, based on previous runs, much like what is

done in iterative compilation [43].

Finally, both lines of action, local and global are connected, thus interesting trade-off can help us to improve the results, and to refine our AD models.

Chapter 5

Conclusions and Further Research Directions

This thesis has two main parts. The first part is devoted to informing the user about possible incorrect use of differentiated codes generated by AD. We called this problem the validity problem of AD. The second part is focused on reducing the memory consumption of the reverse mode of AD. We called this problem the memory problem of reverse mode of AD.

Consequently, in the following we first detail the work we performed to address the first problem. This will be followed by the same exercise for the second problem.

5.1 Summary and Conclusions

5.1.1 The Validity Problem

Automatic Differentiation (AD) tools assume differentiability of the function implemented by the given program. This assumption is fundamental, because the underlying mechanism of AD is the systematic application of the chain rule, which assumes differentiability of every component function. However, sometimes functions are composed by non-smooth elementary functions, which may lead to loss of the global differentiability. Another source of wrong derivatives are switches in the control flow, mainly coming from conditional statements. These switches make most programs only piecewise differentiable. In these cases, sometimes the derivatives close to switches are inconsistent, because they are computed by different sets of instructions. Furthermore, differentiated programs may return derivatives where the implemented function is not differentiable. Unfortunately, the everyday use of AD overlooks those problems, thus problems that should be essential become a matter of concern only when results are not what was expected.

We proposed two methods to tackle the problem of non-differentiability in programs differentiated with Automatic Differentiation. The first method we proposed HSBP allow us to compute the domain of validity for a given input with an acceptable precision. Unfortunately, even although HSBP provide us with a complete description of the domain of validity, the cost of this method is prohibitively high, both in execution time and memory consumption.

The main problem to apply this method is the size of the set of constraints. To cope with that problem we proposed several alternatives. The alternatives include: automatic/manual drop of constraints and change of solution space representation. Unfortunately, none of alternatives prove to reduce significantly the cost of the method. Therefore, we found it is difficult to reduce the cost enough to make this method practical.

The second method we proposed DFP, it compute intervals following a given direction of input data. In these intervals, the returned derivatives have no problem of differentiability. The computational cost of the new mode is marginal (4%) with respect to the computational cost of HSBP. That small overhead is added to the computation of derivatives when DFP is implemented for every test within the code, but only for certain directions in the input space.

The only drawback of DFP is that the information returned by the method is partial, because this information is computed regarding a certain direction in the input space. It is possible to obtain an approximation to the complete information, but that add an extra cost to the method.

The second method can be seen as the other extreme of the spectrum. Execution time and memory usage is low, but the computed domain of validity just informs about a specific direction of derivation in the input space. Nonetheless, the method has proven to be useful as we have shown in our experiments.

DFP is easy to use and is fully implemented in the AD tool TAPENADE. Due to its low cost, we believe that is an efficient way to deal with the validity problems of differentiated codes.

5.1.2 The Memory Problem of The Reverse Mode

Reverse mode computes adjoint codes, in particular gradients. Due to the structure of the reverse mode of AD, the computation of that kind code is very efficient in terms of runtime, but has a drawback. This is because certain intermediate variables are re-defined during the forward sweep, the values that those intermediate variables held are not accessible to compute the derivatives in the reverse sweep.

In order to provide the intermediate values required by the reverse mode, we have mainly two options: Recompute-All (RA) [24] and Store-All (SA) [27] strategy. In this thesis we focus on SA strategy. We have investigated two kind of optimizations for SA strategy. The first kind is the fine-grain optimization, this is based on data-flow analysis. This analysis runs on a static representation of the complete program, and the improvements are visible at instructions level. The second is the coarse-grain optimization, this optimization consists mainly in the checkpointing mechanism, which involves large segments of code.

In order to compute the smaller sets of variable to store/restore, and the fewer number of instructions to generate, we have improved the existing adjoint data-flow analyzes. We add also a new data-flow analysis, this analysis computes the precise set of instructions which are really required to compute the derivatives in the reverse mode. Using these analysis we were able to introduce a refined model for reverse mode of AD. These improved analyzes and AD model are currently implemented in our AD tool TAPENADE.

The adjoint data-flow analyses perform efficiently, they reach up to 49% of improvement in memory consumption, and with an average execution time of 16%. Data-flow analyzes are fundamental in the generation of efficient differentiated code. Particularly, the adjoint data-flow analyzes are the main tool behind the generation of the best backward sweep code. Furthermore, the analyzes allow to compute the smaller sets of values to store during the forward sweep, and even to reduce the number of instructions of the forward sweep by indicating which original instructions are not necessary to compute the derivatives.

Checkpointing is an appealing way to exploit the trade-off between storing and recomputing. A checkpointing strategy is composed of two elements, the snapshot and the checkpoints placement. We introduce a formal equation to compute the snapshot based on the above analysis. This formalization and the fact that the checkpointing change the structure of the differentiated programs force us to adapt the AD model for reverse mode. Therefore, we present an extended AD model which includes checkpointing. This model place the checkpoints systematically before the subroutines calls. This strategy is sub-optimal, and due to the fact that for some segments of code is better not use checkpoint, we introduce the possibility to the user to select the checkpointing placement. All the contributions mentioned are currently implemented in our AD tool. That allow us to run comprehensive experiments with real-size scientific codes.

Data-flow analyzes and Checkpointing are related, and improvements in the former has a consequently positive impact in the latter. The relationship between them

comes from the fact that definition of snapshot is based on data-flow analyzes.

Checkpointing has a good performance, achieving up to 35% decrease in peak memory consumption, and up to 90% decrease in execution time. Therefore, a good checkpointing strategy can improve the performance of reverse differentiated codes.

5.2 Contributions

5.2.1 The Validity Problem

- We formalize a general approach to the validity problem. The approach is based on analyze the variation of each conditional statement.
- We present the first specialization of the general approach. This is called HSPB and relies on the reverse mode of AD.
- We introduce alternatives to reduce the size of the set of constraints.
- We propose a method to automatically drop useless constraints.
- We introduce alternatives to change the representation of constraints.
- We discuss the costs of the alternatives. As a results we present a trade-off between computational costs and accuracy of the representation.
- We present the second specialization of the general approach. This is called DFP and relies on the forward mode of AD.
- We compare DFP and HSBP. And we propose a tactic that allow DFP generate the same kind of solution space as HSBP.
- We present and discuss alternatives in order to reduce the computational cost of DFP, particularly in execution time.
- We present an implementation based on DFP. The implementation can be seen as an extension to the AD tool TAPENADE.

5.2.2 The Memory Problem of the Reverse Mode

- We introduce and compare the two main alternatives to cope with the memory problem of reverse mode of AD.
- We present in detail the Store-All strategy.

- We improve existing data-flow analysis (Activity analysis and TBR) for adjoint code.
- We add a new data-flow analysis (Adjoint-Liveness analysis).
- We present an improved AD model for Store-all, using these analysis.
- We present the checkpointing technique. This technique allow two parameters to be set, the snapshot and the placement of the checkpoints.
- We formalize the snapshot using data-flow analysis.
- We adapt the AD model in order to take into account the Checkpointing strategy.
- We implement the above AD model in our AD tool TAPENADE. This implementation modifies the existing model, and also allows the user to select the checkpointing placements.
- We present and compare checkpointing strategies.

5.3 Future Research Directions

Like all research this work is not finished. We have a number of possible roads for future work. We discuss them here.

5.3.1 The Validity Problem

It is possible to rescue the method to automatically drop constraints. This is to improving the cost of the computation of the ranking of relevance, which is the core stage of that method.

It is possible to increase the accuracy of the solution space of the general approach, thus also for DFP and HSBP, if we use a second order approximation. This will be useful for certain cases, where the first order approximation falls short.

Introduce the results to the AD community in order for the method to be used. Also, showing the usefulness of the method we can encourage the necessary feedback that can helps us to improve the method and make it robust.

The method DFP can be integrated with larger mathematical methods or algorithms, specifically those methods which deal with non-smoothness. DFP can provide useful information for those methods, information that can be regarded, for instance, as feedback for iterative methods.

5.3.2 The Memory Problem of the Reverse Mode

Data-flow analyses can be further refined. Specially promising is the array region analysis adapted to the AD context. Another promising extension to the fine-grain strategies is the possibility of combine storing/recomputing/inverting, which requires a new set of data-flow analyses.

The big open problem of checkpointing is to find an optimal checkpointing strategy for the general case, so far we only have hints about how the checkpoints can be placed, but a consistent strategy is yet to come.

Finally, but no less important, is the further research about the trade-off between the size of the snapshots and the tape of the rest of the code. We can foresee a triple trade-off snapshot/tape/checkpoints placements, which opens a wide range of new strategies to explore in order to optimize the reverse mode of AD.

5.4 Concluding Remarks

In this thesis we have cover two relevant aspects of AD, each one of them have relevance because they are related to fundamental features of AD.

The first aspect is related to the reliability of the derivatives, thanks to our research the users who doubts about the differentiability of their implemented model have the possibility of check their inputs and run their code without hesitation.

The second aspect is related to the efficiency of the reverse differentiated code. We looked for ways to allow the user run their programs without fear to fall short in memory space. This is a big concern when the original programs are large or the internals of the original programs include the usage of gigantic data structures. Thanks to our developments the user is in position to tune the checkpointing strategy which may lead to important saving in memory space as well as in execution time.

We believe that the work is not done yet, and there are open problems that should be address, but the current status of the AD tools, in particular our TAPENADE, provide the user with a powerful tool that actually can help them with their implemented models, allow them to achieve a good performance.

Bibliography

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

[2] A. Griewank and U. Naumann. Accumulating Jacobians as Chained Sparse Matrix Products. *Math. Prog.*, 3(95):555–571, 2003.

[3] http://www.autodiff.com, 2006.

[4] C. Bendtsen and Ole Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, 1996.

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

[6] C. H. Bischof and H. M. Bücker. Computing derivatives of computer programs. In J. Grotendorst, editor, *Modern Methods and Algorithms of Quantum Chemistry: Proceedings, Second Edition*, volume 3 of *NIC Series*, pages 315–327. NIC-Directors, 2000.

[7] Christian H. Bischof, H. Martin Bücker, Wolfgang Marquardt, Monika Petera, and Jutta Wyes. Transforming equation-based models in process engineering. 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.

[8] Christian H. Bischof, H. Martin Bücker, and Andre Vehreschild. A macro language for derivative definition in ADiMat. 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.

[9] Christian H. Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[10] Christian H. Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[11] Alan Carle and Mike Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Department of Computational and Applied Mathematics, Rice University, 2000.

[12] William Castaings, Denis Dartus, Marc Honnorat, François-Xavier Le Dimet, Youssef Loukili, and Jérôme Monnier. Automatic differentiation: A tool for variational data assimilation and adjoint sensitivity analysis for flood modeling. 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] Bruce Christianson and Maurice Cox. Automatic propagation of uncertainties. 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, pages 47–58. Springer, 2005.

[14] The Fortran Company. Fortran 77 standard, 1995.

[15] F. Courty, A. Dervieux, B. Koobus, and L. Hascoët. Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. *Optimization Methods and Software*, 18(5):615–627, 2003.

[16] 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.

[17] 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.

[18] M. Fréchet. Sur la notion de diffrentielle. *C.R. Acad. Sci. Paris*, 152:845–847, 1050–1051, 1911.

[19] M. Fréchet. Sur la notion de diffrentielle totale. *Nouvelles Ann. Math. Sr. 4*, 12:385–403, 433–449, 1912.

[20] R.V. Gamkrelidze. *Analysis II.* Number 14 in Encyclopaedia of Mathematical Sciences. Springer-Verlag, 1987.

[21] R. Gâteaux. Sur les fonctionnelles continues et les fonctionnelles analytiques. *C.R. Acad. Sci. Paris Sr. I Math.*, 157:325–327, 1913.

[22] 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.

[23] R. Giering, T. Kaminski, R. Todling, R. Errico, R. Gelaro, and N. Winslow. Generating tangent linear and adjoint versions of NASA/GMAO's Fortran-90 global weather forecast model. 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.

[24] Ralf Giering. *Tangent Linear and Adjoint Model Compiler, Users Manual*. Center for Global Change Sciences, Department of Earth, Atmospheric, and Planetary Science, MIT, Cambridge, MA, December 1997.

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

[26] 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.

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

[28] Andreas Griewank, David Juedes, H. Mitev, Jean Utke, Olaf Vogel, and Andrea Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Institute of Scientific Computing, Technical University Dresden, 1999. Updated version of the paper published in *ACM Trans. Math. Software* 22, 1996, 131–167.

[29] 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.

[30] Frank P. Hart, Nikhil Kriplani, Sonali R. Luniya, Carlos E. Christoffersen, and Michael B. Steer. Streamlined circuit device model development with FREEDA® and ADOL-C. 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.

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

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

[33] 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.

[34] 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.

[35] Eric Hassold and André Galligo. Automatic differentiation applied to convex optimization. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 287–297. SIAM, Philadelphia, PA, 1996.

[36] Lemaréchal C. Hiriart-Urruty J., editor. *Convex Analysis and Minimization Algorithms I y II.* A Series of Comprehensive Studies in Mathematics. Springer-Verlag, New York, NY, 1991.

[37] Masao Iri. History of automatic differentiation and rounding estimation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 1–16. SIAM, Philadelphia, PA, 1991.

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

[39] Kyung-Wook Jee, Daniel L. McShan, and Benedick A. Fraass. Implementation of automatic differentiation tools for multicriteria IMRT optimization. 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.

[40] R. B. Kearfott, M. Dawande, K.S. Du, and C.Y. Hu. Algorithm 737: Intlib, a portable fortran 77 interval standard function library. *ACM Trans. Math. Software*, 20 (4):447459, 1994.

[41] R. Baker Kearfott. Treating non-smooth functions as smooth functions in global optimization and nonlinear systems solvers. In Götz Alefeld, Andreas Frommer, and Bruno Lang, editors, *Scientific Computing and Validated Numerics, Mathematical Research*, pages 160–172. Akademie Verlag, Berlin, 1995.

[42] R. Baker Kearfott. Interval extensions of non-smooth functions for global optimization and nonlinear systems solvers. *Computing*, 1996. Accepted for publication.

[43] 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.

[44] Koichi Kubota. Computation of matrix permanent with automatic differentiation. 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.

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

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

[47] 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.

[48] Uwe Naumann. *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs*. PhD thesis, Technical University of Dresden, December 1999.

[49] Uwe Naumann and Jan Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. 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.

[50] A. Neumaier. Interval methods for systems of equations. In et all A. Neumaier, G.-C. Rota, editor, *Encyclopedia of Mathematics and its Applications*, pages 170–215. Cambridge University Press, Cambridge, England, 1990.

[51] Derya B. Özyurt and Paul I. Barton. Application of targeted automatic differentiation to large-scale dynamic optimization. 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.

[52] Eric Phipps, Richard Casey, and John Guckenheimer. Periodic orbits of hybrid systems and parameter estimation via AD. 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.

[53] 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.

[54] J. Goffin S. Elhedhli and J. Vial. Nondifferentiable optimization: Cutting plane methods. In C.A. Floudas and P.M. Pardalos, editors, *Encyclopedia of Optimization*, volume 4, pages 40–45. Kluwer Academic, 2001.

[55] Dmitri Shiriaev. ADOL–F automatic differentiation of Fortran codes. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 375–384. SIAM, Philadelphia, PA, 1996.

[56] Pavel Snopok, Carol Johnstone, and Martin Berz. Simulation and optimization of the Tevatron accelerator. 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.

[57] 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.

[58] Mohamed Tadjouddine, Shaun A. Forth, and Andy J. Keane. Adjoint differentiation of a structural dynamics solver. 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.

[59] Michael Ulbrich and Stefan Ulbrich. Automatic differentiation: A structure-exploiting forward mode with almost optimal complexity for Kantorovic trees. In Herbert C. Fischer, B. Riedmueller, and S. Schaeffler, editors, *Applied Mathematics and Parallel Computing, Festschrift for Klaus Ritter*, pages 327–357. Physica-Verlag, Berlin, 1996.

[60] Jean Utke. OPENAD: Algorithm implementation user guide. Technical Memorandum ANL/MCS–TM–274, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 2004.

[61] L. Vandenberghe and S. Boyd. Semidefinite programming. *SIAM Review*, 38 (1):49–95, 1996.

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

[63] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

[64] 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.

Appendix A

A.1 Addendum to tapenade Tutorial

This tutorial is focused on the utilization of the new features of TAPENADE. These new features were developed during this Thesis. To access a complete tutorial, please connect to http://www-sop.inria.fr/tropics/.

A.1.1 Requirements

In order to perform the experiments without any initial problem, the following requirements must be fulfill.

- TAPENADE version 2.1 or 2.2 (download it from ftp://ftp-sop.inria.fr/tropics/tapenade/)
- Programs to be differentiated must be Fortran77/90 (F77/90) source code.
- Some F77/90 compiler.

A.1.2 Domain of Validity

Procedure

- Once the program to be differentiated is available, it must be compiled with the compiler of choice and executed. This is just to be sure we are in safe ground, and the program has no bugs.
- Define the variables of your interest (dependent and independent).
- We call TAPENADE with all the necessary parameters. If there is any doubt, the TAPENADE tutorial should be re-read. In this particular tutorial we care only about the TAPENADE switch -dV (for tangent validity mode).
- Add the following lines to every subroutine inside your code that should be accessed to update the validity information:

```
COMMON /validity_test_common/ gmin, gmax, infmin, infmax
REAL gmin, gmax
LOGICAL infmin, infmax
```

That piece of code give you access to the boundaries of the safe neighborhood, where gmin is the lower bound of the interval and gmax is the upper bound, if infmin or infmax is true, then the interval has a infinite lower or upper bound.

- Link the following object file *validityTest.o* (ELF 32-bit LSB relocatable, Intel 80386) to the rest of your binary objects. This file is located in the directory ADFirstAidKit, which is part of the distribution package of TAPENADE.
- Compile the code.
- Executing the program provide the runtime information about differential validity of the domain.

Example of Application

- Source Code of the Example.

```
SUBROUTINE SUB1(x, y, o1)
REAL x, y, o1

x = y * x
o1 = x * x + y * y
IF ( o1 > 190 ) THEN
   o1 = -o1*o1/2
ELSE
   o1 = o1*o1*20
ENDIF
END
```

- Calling TAPENADE with the selected parameters, and the option (-dV)

```
> tapenade -dV -head sub1 -vars "x y" -outvars "o1"
-o example1 example1.f

Tapenade - Version 2.2 (r1229) - Tue Jun 27 15:24:32 2006
@@ Creating ./example1_dva.f
job is complete
```

The differentiated code is as follows,

```
C        Generated by TAPENADE     (INRIA, Tropics team)
C  Tapenade - Version 2.2 (r1229) - Tue Jun 27 15:24:32 2006
C
C  Differentiation of sub1 in forward (tangent) validated mode:
C   variations  of output variables: x o1
C   with respect to input variables: x y
      SUBROUTINE SUB1_DVA(x, xd, y, yd, o1, o1d)
      IMPLICIT NONE
      REAL o1, o1d, x, xd, y, yd
C
      xd = yd*x + y*xd
      x = y*x
      o1d = xd*x + x*xd + yd*y + y*yd
      o1 = x*x + y*y
      CALL VALIDITY_TEST(o1 - 190, o1d)
      IF (o1 .GT. 190) THEN
        o1d = -((o1d*o1+o1*o1d)/2)
        o1 = -(o1*o1/2)
        xd = 0.0
      ELSE
        o1d = 20*(o1d*o1+o1*o1d)
        o1 = o1*o1*20
        xd = 0.0
      END IF
      END
```

- Adding the validity enabling code.
  In this case, the initialization of the block of data is placed in the program which calls to subroutine `SUB1_DVA()`. For instance, the block of data is added to small driver program.

- Results.
  After compiling the driver along with files example1_dva and validityTest. The differentiated program execution generates the following results:

  In Table A.1, we can see how the interval change around the critical point (where the test switch). Before the switch, the interval is $[-\infty, 0.005]$, which means that derivate is close to a discontinuity in the given direction, but in the opposite direction is $-\infty$, this means that there is no differential problem. Conversely, after the test, the interval is $[0.004, \infty]$, because the critical point lay behind the evaluated point in the given direction.

x	y	o1d	[gmin	gmax]
3.62	3.62	1456628.2	$-\infty$	0.026
3.63	3.63	1484149.2	$-\infty$	0.016
3.64	3.64	1512117.1	$-\infty$	0.005
3.65	3.65	-38513.4	-0.004	∞
3.66	3.66	-39235.4	-0.014	∞
3.67	3.67	-39969.0	-0.023	∞

Table A.1: Results from validated code of the example with direction (xd,yd) = (1,1).

A.1.3 Checkpointing

Procedure

- Differentiate in reverse mode a program with TAPENADE. By default all subroutines will be checkpointed.
- Analyze the differentiate code looking for large snapshots, tape and other important features of the code, for instance, loops.
- Chose the more appealing subroutines to be no checkpointed. Mark these subroutines inserting the directive ("`C$AD NOCHECKPOINT`") just before the subroutine call, or using the command line option ("-split [name of subroutine]"). Then, Re-execute TAPENADE with the corresponding parameters.
- Compare the results, if the results are unsatisfactory, repeat the process changing the checkpoint placements.

Example of Application

- We use a segment of the scientific program UNS2D to illustrate the checkpointing utilization.
  Program UNS2D call-graph can be observed in Figure A.1 (next page).

  From this call-graph we carry out experiments on the sub call-graph under subroutine `INBIGFUNC`. Therefore, the selected segment of code is composed of the subroutine `INBIGFUNC`, which calls to subroutines: `DIFFAR, FLW2D, SYMMT, CALGRA`. The header of the subroutine `INBIGFUNC` includes the following input/output variables:

```
SUBROUTINE INBIGFUNC(kmult,kvisc,coefrk,dtl,
     &       th1,th2,th3,th4,
     &       nubo,vnocl,pres,dm,coor,f1,f2,f3,f4,g1,g2,g3,g4,
```

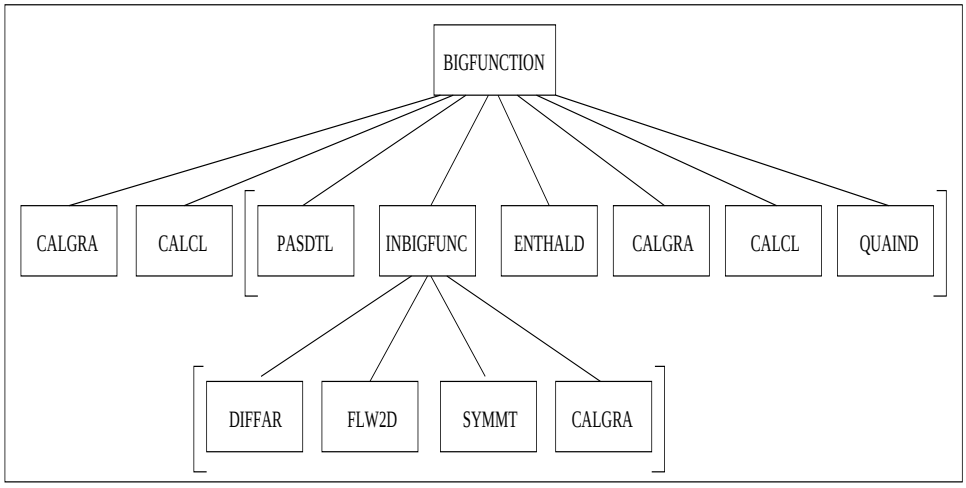


Figure A.1: UNS2D Call-graph.

```
     &       t1,t2,t3,t4,rh1,rh2,rh3,rh4,rhd1,rhd2,rhd3,rhd4,
     &       vn1,vn2,vn3,noe1,noe2,noe3,icola,iarg,icora,
     &       cson,temp,amu,amut)
```

- The body of the subroutine `INBIGFUNC` is as follows:

```
      do 200 iter=1,kmult
c
c-------------------------------------------------------------
c     Physical and Artificial Viscosity
c     ---------------------------------
      IF(ITER.LE.KVISC) THEN
c     Artificial viscosity
c     --------------------
      call diffar (nubo,vnocl,pres,dm,coor,f1,f2,g1,g2,g3,g4,
     &       t1,t2,t3,t4,rh1,rh2,rh3,rh4,rhd1,rhd2,rhd3,rhd4,
     &       vn1,vn2,vn3,noe1,noe2,noe3,icola,
     &       f3,f4,cson)
      ENDIF
c-------------------------------------------------------------
c     Inviscid Fluxes
c     ---------------
      call flw2d(nubo,vnocl,icola,t1,t2,t3,t4,rh1,rh2,rh3,rh4,
     f       g1,g2,g3,g4,pres,noe1,noe2,noe3,vn1,vn2,vn3,
     .       cson,iarg,icora,coor)
c-------------------------------------------------------------
```

```
c     Inviscid fluxes + viscous (artificial) fluxes
c     ---------------------------------------------------
      do 202 is=1,ns
      rh1(is)=rh1(is)+rhd1(is)
      rh2(is)=rh2(is)+rhd2(is)
      rh3(is)=rh3(is)+rhd3(is)
      rh4(is)=rh4(is)+rhd4(is)
  202 continue
c-----------------------------------------------------------
c     Sub-solution
c     ------------
      do 203 is=1,ns
      dtme=coefrk(iter)*dtl(is)*dm(is)
      t1(is)=th1(is)+dtme*rh1(is)
      t2(is)=th2(is)+dtme*rh2(is)
      t3(is)=th3(is)+dtme*rh3(is)
      t4(is)=th4(is)+dtme*rh4(is)
  203 continue
c-----------------------------------------------------------
c     Symetric plane if it exists
c     ---------------------------
      call symmt(noe3,vn3,t1,t2,t3,t4,iarg,icora,th1,th2,th3,th4)
c-----------------------------------------------------------
      IF(ITER.NE.KMULT) THEN
c     Computation of the primitive variables
c     --------------------------------------
      call calgra(t1,t2,t3,t4,g1,g2,g3,g4,pres,cson,temp,amu,
     .            amut)
      ENDIF
c-----------------------------------------------------------
c
  200 continue
```

- We execute TAPENADE with the usual parameters used to differentiate this code.

```
> tapenade -b -head inbigfunc -vars "dtl f1 f2 rh1 rh2 rh3
rh4 coor vnocl g1 g2 g3 t1 g4 t2 t3 t4 th1 th2 th3 th4 vn1
vn2 rhd1 rhd2 rhd3 rhd4 cson cla pres" -outvars "dtl f1 f2
rh1 rh2 rh3 rh4 coor vnocl g1 g2 g3 t1 g4 t2 t3 t4 th1 th2
th3 th4 vn1 vn2 rhd1 rhd2 rhd3 rhd4 cson pres" -o allDiff *.f
```

The differentiated subroutines are generated and printed in the file *allDiff_b.f*.

- A snippet of the reverse differentiated code is as follows:

```
      DO iter=1,kmult
C
C-------------------------------------------------------------
C     Physical and Artificial Viscosity
C     ---------------------------------
        IF (iter .LE. kvisc) THEN
          CALL PUSHINTEGER4ARRAY(icola, 21)
          CALL PUSHINTEGER4ARRAY(noe3, 1000)
          CALL PUSHINTEGER4ARRAY(noe2, 1000)
          CALL PUSHINTEGER4ARRAY(noe1, 1000)
          CALL PUSHREAL4ARRAY(vn3, 2*1000)
          CALL PUSHREAL4ARRAY(vn2, 2*1000)
          CALL PUSHREAL4ARRAY(vn1, 2*1000)
          CALL PUSHREAL4ARRAY(rh4, 2000)
          CALL PUSHREAL4ARRAY(rh3, 2000)
          CALL PUSHREAL4ARRAY(rh2, 2000)
          CALL PUSHREAL4ARRAY(rh1, 2000)
          CALL PUSHREAL4ARRAY(t4, 2000)
          CALL PUSHREAL4ARRAY(f2, 2000)
          CALL PUSHREAL4ARRAY(f1, 2000)
          CALL PUSHREAL4ARRAY(dm, 2000)
          CALL PUSHREAL4ARRAY(pres, 2000)
          CALL PUSHREAL4ARRAY(vnocl, 2*6000)
          CALL PUSHINTEGER4ARRAY(nubo, 2*6000)
C     Artificial viscosity
C     --------------------
          CALL DIFFAR(nubo, vnocl, pres, dm, coor, f1, f2,
     +       g1, g2, g3, g4, t1, t2, t3, t4, rh1, rh2, rh3,
     +       rh4, rhd1, rhd2, rhd3, rhd4, vn1, vn2, vn3,
     +       noe1, noe2, noe3, icola, f3, f4, cson)
          CALL PUSHINTEGER4(1)
        ELSE
          CALL PUSHINTEGER4(0)
        END IF
```

Here, in the differentiated code, the rest of the forward sweep and the beginning of the backward sweep is placed, until the restoration of the snapshot for DIFFAR_B.

```
        CALL POPINTEGER4(branch)
```

```
      IF (.NOT.branch .LT. 1) THEN
        CALL POPINTEGER4ARRAY(nubo, 2*6000)
        CALL POPREAL4ARRAY(vnocl, 2*6000)
        CALL POPREAL4ARRAY(pres, 2000)
        CALL POPREAL4ARRAY(dm, 2000)
        CALL POPREAL4ARRAY(f1, 2000)
        CALL POPREAL4ARRAY(f2, 2000)
        CALL POPREAL4ARRAY(t4, 2000)
        CALL POPREAL4ARRAY(rh1, 2000)
        CALL POPREAL4ARRAY(rh2, 2000)
        CALL POPREAL4ARRAY(rh3, 2000)
        CALL POPREAL4ARRAY(rh4, 2000)
        CALL POPREAL4ARRAY(vn1, 2*1000)
        CALL POPREAL4ARRAY(vn2, 2*1000)
        CALL POPREAL4ARRAY(vn3, 2*1000)
        CALL POPINTEGER4ARRAY(noe1, 1000)
        CALL POPINTEGER4ARRAY(noe2, 1000)
        CALL POPINTEGER4ARRAY(noe3, 1000)
        CALL POPINTEGER4ARRAY(icola, 21)
        CALL DIFFAR_B(nubo, vnocl, vnoclb, pres, presb, dm,
     +       , coorb, f1, f1b, f2, f2b, g1, g2, g3, g4, t1,
     +       t1b, t2, t2b, t3, t3b, t4, t4b, rh1, rh1b, rh2,
     +       rh2b, rh3, rh3b, rh4, rh4b, rhd1, rhd1b, rhd2,
     +       rhd2b, rhd3, rhd3b, rhd4, rhd4b, vn1, vn1b, vn2
     +       , vn2b, vn3, vn3b, noe1, noe2, noe3, icola, f3,
     +       f4, cson, dmb, coor)
      END IF
    ENDDO
```

The snippet represents a part of the forward/backward sweep of the subroutine `INBIGFUNC`. We can observe the snapshot before subroutine `DIFFAR`, and we can also obverse how the snapshot is restored before the reverse differentiated version of the subroutine `DIFFAR`. The snapshots are stored/restored using the PUSH/POP subroutines. So far, every subroutine is checkpointed, which is by default the strategy in TAPENADE.

- In order to change of checkpointing strategy we will deactivate the checkpointing mechanism for an arbitrary subroutine. In this case we chose to do this with subroutine `DIFFAR`. Therefore, we insert in subroutine `INBIGFUNC` the directive that deactivate the checkpointing mechanism for the subroutine `DIFFAR`. As we can observe in the following snippet of the original subroutine `INBIGFUNC`. The directive syntax is "`C$AD NOCHECKPOINT`".

```
    do 200 iter=1,kmult
c
c-------------------------------------------------------------
c     Physical and Artificial Viscosity
c     ---------------------------------
      IF(ITER.LE.KVISC) THEN
c     Artificial viscosity
c     --------------------
C$AD NOCHECKPOINT
      call diffar (nubo,vnocl,pres,dm,coor,f1,f2,g1,g2,g3,g4,
     &       t1,t2,t3,t4,rh1,rh2,rh3,rh4,rhd1,rhd2,rhd3,rhd4,
     &       vn1,vn2,vn3,noe1,noe2,noe3,icola,
     &       f3,f4,cson)
      ENDIF
```

We execute TAPENADE with the same parameter as the first execution. As a result, we obtain the following reverse differentiated code.

```
    do 200 iter=1,kmult
C
C-------------------------------------------------------------
C     Physical and Artificial Viscosity
C     ---------------------------------
        IF (iter .LE. kvisc) THEN
C     Artificial viscosity
C     --------------------
          CALL DIFFAR_FWD(nubo, vnocl, pres, dm, coor, f1, f2,
     +       g3, g4, t1, t2, t3, t4, rh1, rh2, rh3, rh4, g1, g2,
     +       rhd1, rhd2, rhd3, rhd4, vn1, vn2, vn3, noe1,
     +       noe2, noe3, icola, f3, f4, cson)
          CALL PUSHINTEGER4(1)
        ELSE
          CALL PUSHINTEGER4(0)
        END IF
```

Here, in the differentiated code, the rest of the forward sweep and the beginning of the backward sweep is placed. This code is normal reverse differentiated code, thus every subroutine is checkpointed.

```
        CALL POPINTEGER4(branch)
        IF (.NOT.branch .LT. 1) THEN
          CALL DIFFAR_BWD(nubo, vnocl, vnoclb, pres, presb,
```

```
+       coor, coorb, f1, f1b, f2, dm, dmb,
+       , f2b, g1, g2, g3, g4,
+       t1, t1b, t2, t2b, t3,
+       t3b, t4, t4b, rh1, rh1b
+       , rh2, rh2b, rh3, rh3b,
+       rh4, rh4b, rhd1, rhd1b,
+       rhd2, rhd2b, rhd3, rhd3b
+       , rhd4, rhd4b, vn1, vn1b
+       , vn2, vn2b, vn3, vn3b,
+       noe1, noe2, noe3, icola
+       , f3, f4, cson)
   ENDIF
 ENDDO
```

As we can observe deactivate the checkpoint mechanism for subroutine `DIFFAR` spare the storage of a large snapshot, therefore reducing the size of the tape for subroutine `INBIGFUNC`. However, this choice has a hidden cost in memory space. This is the local variables values that must to be stored before the end of subroutine `DIFFAR_FWD`.

We can use a TAPENADE switch to achieve the same result. This switch is "-split [name of subroutine]". In order to obtain the same above reverse differentiated code, we have to executed TAPENADE with the following parameters:

```
tapenade -b -head inbigfunc -vars "dtl f1 f2 rh1 rh2
rh3 rh4 coor vnocl g1 g2 g3 t1 g4 t2 t3 t4 th1 th2 th3
th4 vn1 vn2 rhd1 rhd2 rhd3 rhd4 cson cla pres"
-outvars "dtl f1 f2 rh1 rh2 rh3 rh4 coor vnocl g1 g2
g3 t1 g4 t2 t3 t4 th1 th2 th3 th4 vn1 vn2 rhd1 rhd2
rhd3 rhd4 cson pres" -split diffar -o allDiffnocheck *.f
```

- All possible combinations of checkpointing mechanism deactivation can be applied to the subroutines of the subroutine `INBIGFUNC`. We leave the users to try those combinations.

Appendix B

B.1 Program Examples

B.1.1 Extended Results for Validity Information with Newton Method

Implementation of the objective function $f()$.

```
REAL FUNCTION F(x)
REAL x
   f = x**3 + x**2 - 3*x - 3
RETURN
END
```

Obtaining the first order derivative with TAPENADE.

```
> tapenade -d -vars "x" -outvars "f" -o f f.f
```

Obtaining the second order derivative with TAPENADE.

```
> tapenade -d -vars "x" -outvars "f_d" -o f_d f_d.f
```

Differentiated code for objective function $f()$, including the original function, the first and second derivative.

```
C        Generated by TAPENADE     (INRIA, Tropics team)
C  Tapenade - Version 2.2 (r1229) - Tue Jun 27 15:24:32 2006
C
C  Differentiation of f_d in forward (tangent) mode:
C   variations  of output variables: f_d
C   with respect to input variables: x
      REAL FUNCTION F_D_D(x, xd0, xd, f, f_d)
      IMPLICIT NONE
      REAL f, f_d, x, xd, xd0
```

```
      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
      RETURN
      END
```

Implementation of Newton Method.

```
      PROGRAM MAIN
      REAL tol, x, xd, xd0, y, yd, ydd
      INTEGER iter_counter, iter_limit

C Iteration limit
      iter_limit = 20
C Tolerance
      tol = 1.0e-3
C Iteration counter
      iter_counter = 0
C Initial guess
      x = 2.5
C Initial guess directions
      xd = 1.0
      xd0 = 1.0
C Initializations
      y = 0.0
      yd = 0.0
      ydd = 0.0
      WRITE(*, '(A6,X,A9,2X,A9,X,A9)') 'x','y',"y'","y''"
      DO
C Call to second order differentiated code of function F
        ydd = F_D_D(x, xd0, xd, y, yd)
        WRITE(*,'(F9.5,X,F9.5,X,F9.5,X,F9.5)') x,y,yd,ydd
        x = x - (yd / ydd)
C Stoping criterion I
        IF ( abs(yd) < tol ) EXIT
        iter_counter = iter_counter + 1
C Stoping criterion II
        IF ( iter_counter > iter_limit ) EXIT
      END DO
      END
```

Results obtained executing the Newton method on the presented objective function $f()$.

```
     x          y          y'        y''
  2.50000   11.37500   20.75000   17.00000
  1.27941   -3.10708    4.46951    9.67647
  0.81752   -4.23784    0.64004    6.90510
  0.72483   -4.26830    0.02577    6.34896
  0.72077   -4.26835    0.00005    6.32460
```

Implementation of piecewise objective function *f_piecewise*().

```
      REAL FUNCTION F_PIECEWISE(x)
      REAL 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
      RETURN
      END
```

Differentiated code for objective function *f_piecewise*(), including the original function, the first and second order derivative.

```
C        Generated by TAPENADE     (INRIA, Tropics team)
C  Tapenade - Version 2.2 (r1229) - Tue Jun 27 15:24:32 2006
C
C  Differentiation of f_piecewise_d in forward (tangent) mode:
C   variations  of output variables: f_piecewise_d
C   with respect to input variables: x
      REAL FUNCTION F_PIECEWISE_D_D(x, xd0, xd, f_piecewise,
     +                              f_piecewise_d)
      IMPLICIT NONE
      REAL f_piecewise, f_piecewise_d, x, xd, xd0
C
      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
```

```
      RETURN
      END
```

Results obtained from execute the Newton method on the presented objective function *f_piecewise*().

```
     x         y         y'       y''
  2.50000  11.37500  20.75000  17.00000
  1.27941  -3.10708   4.46951   9.67647
  0.81752  -2.37972  -9.72493   8.90510
  1.90958   1.88102  11.75864  13.45748
  1.03582  -3.92319   2.29038   8.21490
  0.75701  -1.77519 -10.25278   8.54205
  1.95728   2.45734  12.40740  13.74368
  1.05451  -3.87894   2.44499   8.32705
  0.76089  -1.81492 -10.21958   8.56534
  1.95402   2.41699  12.36265  13.72413
  1.05323  -3.88207   2.43430   8.31935
  0.76062  -1.81215 -10.22191   8.56371
  1.95425   2.41979  12.36577  13.72549
  1.05331  -3.88185   2.43505   8.31989
  0.76064  -1.81234 -10.22175   8.56382
  1.95423   2.41960  12.36555  13.72540
  1.05331  -3.88186   2.43500   8.31985
  0.76064  -1.81233 -10.22176   8.56382
  1.95423   2.41961  12.36556  13.72541
  1.05331  -3.88186   2.43500   8.31985
  0.76064  -1.81233 -10.22176   8.56382
```

In order to obtain the implementation of *f_piecewise*() with the validity analysis.

```
> tapenade -dV -vars "x" -outvars "f_piecewise_d"
-o f_piecewise_d f_piecewise_d.f
```

Differentiated code for objective function *f_piecewise*(), including the original function, the first and second order derivative, and the call to the subroutine which compute the validity information.

```
C        Generated by TAPENADE     (INRIA, Tropics team)
C  Tapenade - Version 2.2 (r1229) - Tue Jun 27 15:24:32 2006
C
C  Differentiation of f_piecewise_d in forward (tangent) validated mode:
C   variations  of output variables: f_piecewise_d
C   with respect to input variables: x
```

```
      REAL FUNCTION F_PIECEWISE_D_DVA(x, xd0, xd, f_piecewise,
     +                                f_piecewise_d)
      IMPLICIT NONE
      REAL f_piecewise, f_piecewise_d, x, xd, xd0
      REAL f_piecewised, xdd
      xdd = 0.0
      f_piecewise_d_dva = 0.0
      f_piecewised = 0.0
C
      CALL VALIDITY_TEST(x - 1, xd0)
      IF (x .GT. 1) THEN
        f_piecewise_d_dva = 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_dva = 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
      RETURN
      END
```

x	y	y'	y''	gmin	gmax
2.50000	11.37500	20.75000	17.00000	-1.50000	0.00000
1.27941	-3.10708	4.46951	9.67647	-0.27941	0.00000
0.81752	-2.37972	-9.72493	8.90510	0.00000	0.18248
1.90958	1.88102	11.75864	13.45748	-0.90958	0.00000
1.03582	-3.92319	2.29038	8.21490	-0.03582	0.00000
0.75701	-1.77519	-10.25278	8.54205	0.00000	0.24299
1.95728	2.45734	12.40740	13.74368	-0.95728	0.00000
1.05451	-3.87894	2.44499	8.32705	-0.05451	0.00000
0.76089	-1.81492	-10.21958	8.56534	0.00000	0.23911
1.95402	2.41699	12.36265	13.72413	-0.95402	0.00000
1.05323	-3.88207	2.43430	8.31935	-0.05323	0.00000
0.76062	-1.81215	-10.22191	8.56371	0.00000	0.23938
1.95425	2.41979	12.36577	13.72549	-0.95425	0.00000
1.05331	-3.88185	2.43505	8.31989	-0.05331	0.00000
0.76064	-1.81234	-10.22175	8.56382	0.00000	0.23936
1.95423	2.41960	12.36555	13.72540	-0.95423	0.00000
1.05331	-3.88186	2.43500	8.31985	-0.05331	0.00000
0.76064	-1.81233	-10.22176	8.56382	0.00000	0.23936
1.95423	2.41961	12.36556	13.72541	-0.95423	0.00000

```
 1.05331  -3.88186   2.43500   8.31985  -0.05331   0.00000
 0.76064  -1.81233 -10.22176   8.56382   0.00000   0.23936
```