Automatic Differentiation for Adjoint Code Generation

Christèle Faure

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Automatic Differentiation for adjoint code generation

Christèle Faure, Ed.

N° 3555

Novembre 1998

THÈME 2

ISSN 0249-6399

Automatic Differentiation for adjoint code generation

Christèle Faure*, Ed.

Thème 2 — Génie logiciel
et calcul symbolique
Projet SAFIR

Rapport de recherche n° 3555 — Novembre 1998 — 56 pages

Abstract: This document is a compilation of extended abstracts presented at the Automatic Differentiation session entitled "**Automatic Differentiation for adjoint code generation**"† which has been held within the **IMACS Conference on Applications of Computer Algebra** held in Prague (Czech Republic) in August 1998. This is the second Automatic Differentiation session organized within IMACS'ACA conferences.

List of contributors :

Key-words: Adjifor, Odyssée, Padre 2, Tamc, reverse mode, adjoint code, automatic differentiation, computational differentiation, fortran, program transformation.

* Email : Christele.Faure@sophia.inria.fr, URL : http://www.inria.fr/safir/WHOSWHO/Christele.Faure

† This session was partially supported by INRIA through the INRIA cooperative research action called MIO. We want to thank also the IMACS ACA'98 organizers for asking us to organize such a session.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Génération de code adjoint par Différentiation Automatique

Résumé : Ce document est le recueil de résumés étendus présentés à la session de Differentiation Automatique intitulée `Automatic Differentiation for adjoint code generation`[‡] qui faisait partie de `IMACS Conference on Applications of Computer Algebra (IMACS ACA'98)` qui a eu lieu à Prague (République Tchèque) en Août 1998. C'est la deuxième session sur le thème de la Différentiation Automatique organisée dans le cadre des conférences IMACS'ACA.

Liste des contributeurs :

Mots-clés : Adjifor, Odyssée, Padre 2, Tamc, mode inverse, code adjoint, différentation automatique, fortran, transformation de programme.

[‡] Cette session a été partiellement financée par l'INRIA dans le cadre de l'action de recherche cooperative MIO. Nous voulons aussi remercier les organisateurs de IMACS ACA'98 pour nous avoir donné la possibilité d'organiser cette session.

List of contributions

An introduction to Automatic Adjoint code generation

Automatic differentiation is a set of techniques aimed at differentiating functions based on a program that computes its values at arbitrary points. For an introduction to automatic differentiation theory and techniques refer to the proceedings of the main conferences on this subject : "Automatic Differentiation of Algorithms: Theory, Implementation, and Applications" (see [27]) and "Computational Differentiation: Applications, Techniques, and Tools" (see [3]). The method differs from finite differences in that the value of derivatives are computed exactly (up to rounding errors) and generally in a more efficient way. This is particularly true when gradients (sets of partial derivatives) are to be computed. In this case, the *reverse* automatic differentiation can be viewed as a method for generating discrete adjoint codes automatically.

Modeling and forecasting of complex physical phenomena require the computation of derivatives. In optimal design or data assimilation for example these derivatives are primarily gradients. For that purpose discrete adjoint codes are widely used in the different communities.

The reverse mode of automatic differentiation is functionally equivalent to hand written discrete adjoint codes. Several automatic differentiation tools enable the user to avoid the time-consuming and error prone task of developing adjoint code by hand. For a survey of the existing tools and their functionalities, visit the web page of the Computational Differentiation Project at Argonne National Laboratory at URL [4]. The three main source to source tools in which the reverse mode has been implemented are: Odyssée (see [18]), Padre2 (see [33]), TAMC (see [20]). A new system to be called called Adjifor is under development (see [9]).

Whereas automatically generated reverse codes work well on small and medium sized problems they typically need to be tuned and optimized on really large applications. Of particular concern is the management of the potentially very large memory requirement.

The knowledge of the physical problem being modeled, which is exploited in hand writing discrete adjoint codes, cannot usually be extracted from the code by automatic tools. It also should be noted that programmers in communities where adjoints are widely used adhere to certain coding conventions, which are not yet exploited by automatic differentiation tools.

On the other hand, AD developers have studied new trade-off between memory and execution time requirements which may be used to help adjoint developers.

The practical knowledge of discrete adjoint developers added to the technical knowledge of AD developers may lead to great improvements in the two approaches.

In this document, we show different research directions that aim at making the *reverse mode* of automatic differentiation applicable to really large codes. This document is a compilation of extended abstracts presented at the automatic differentiation session entitled `Automatic Differentiation for adjoint code generation`[1], which was held within the `IMACS Conference on Applications of Computer Algebra` held in Prague (Czech Republic) in August 1998.

Each paper emphasizes a different aspect of current research activities. The three first contributions are purely dedicated to the reverse mode and show some research directions in this field:

checkpointing This method leads to an optimal trade-off between storage and recomputation of the variables modified within a loop. The related extended abstract (see contribution 1 page 9) shows some results obtained with the discrete adjoint of a meteorological code called Meso-NH automatically generated.

program analysis Some program analysis methods can be applied within automatic differentiation translators to optimize the generated code. For example the array region analysis (see contribution 2 page 15) can be applied to diminish the amount of storage necessary in reverse mode.

parallelization Since real applications of adjoint models require a lot of computational power and memory, in the past few years many models have been developed for distributed memory parallel computers. An example of automatic generation of adjoint code from an operational parallel model is shown in contribution 3 page 23.

The three other contributions show how the direct and reverse modes of differentiation can be mixed to get more efficient codes, or can be combined to get second order derivatives:

cross-country elimination This approach is intended to generalize the chain rule beyond the pure forward or reverse modes of automatic differentiation. The related contribution (see contribution 4 page 47) shows the main theoretical aspects of such an application of the chain rule.

hierarchical approach This methodology is based on the idea that the associativity of the chain rule allows derivative propagation to be performed at arbitrary levels of abstractions. The corresponding contribution (see contribution 5 page 39) presents one implementation of this new technology based on the AIF internal representation.

[1]This session was partially supported by INRIA through the INRIA cooperative research action called MIO. We want to thank also the IMACS ACA'98 organizers for asking us to organize such a session.

second order derivatives The first goal of generating automatically adjoint codes having been achieved, the goal of getting second order derivatives can be addressed. One way of getting second order derivatives of a scalar function is to apply forward differentiation to an adjoint code (see contribution 6 page 31).

Generation of the Adjoint Code of Meso-NH

Isabelle Charpentier[2]

Isabelle.Charpentier@imag.fr

Projet IDOPT,
51 rue des mathématiques, BP 53,
F-38041 Grenoble Cedex 9.

The mesoscale meteorological model Meso-NH is able to simulate atmospheric events ranging from mesoscale down to micro-scale. The MesODiF package was generated to complete Meso-NH with both a tangent linear code and an adjoint code. The differentiation work, realized with respect to the state variables, is done on the adiabatic part of the code, physical parameterizations are not yet taken into account. The linearized codes are efficient in term of time and memory consumptions.

2.1 Introduction

Meso-scale models are designed to capture mesoscale phenomena with spatial scales of a few kilometers and time scales of a few minutes. For example these models facilitate the study of hurricanes, thunderstorms and tornadoes, and urban air pollution events when coupled with atmospheric chemistry models. Studies such as sensitivity analysis, calibration of a model, or variational data assimilation may require some gradients computations, the reader is referred to papers [36] and [35]. Nowadays these applications are attainable with the Meso-NH model ([34], [37]) through the use of the MesODiF library ([10]).

The MesODiF package contains the Meso-NH model (version 2.4), and both the tangent linear code and the adjoint code of the adiabatic part of the model. Physical parameteri-

[2]This work was supported by the INRIA action for Operative Inverse Mode, the CEMRACS'97 and the IDRIS computational center.

zations, that are discontinuous, are not yet included. The automatic differentiator Odyssée [18] is employed to perform the linearization work in order to avoid the tedious and error prone task of differentiation. Post-processing steps described in [12], [11] are then used to improve the performances of the adjoint code.

The layout of this paper is the following. A simplified geophysical equations system and its adjoint system are described in Section 2. Section 3 briefly describes the differentiation work whereas Section 4 presents the most interesting improvements realized for the adjoint code. Numerical results are displayed in Section 5 and Section 6 concludes on the usability of the MesODiF package.

2.2 Gradients computations

Gradients computations become an important tool in scientific computing, this is especially true for geophysical applications where sensitivity analyses or variational data assimilation techniques allow for model validations, identifications of parameters, forecasting experiments,...

2.2.1 Optimal control for geophysical equations

When studying the atmospheric circulation, the main goal to achieve has always been the weather forecasting: one aims to predict the state of the atmosphere after time T. In order to be able to forecast, one has to work with a good numerical meteorological model and to know a "good" approximation of the state of the atmosphere at time T. The data assimilation method proposed by F.-X. Le Dimet and O. Talagrand [35] solves the forecasting problem as follows.
Let Ω be an open bounded domain of the atmosphere, $[0,T]$ interval of time and X the state variable belonging to the set $\mathcal{X}$ of the admissible states of the atmosphere in $\Omega \times [0,T]$. The governing equations of the atmosphere defining the *direct model* are written below:

$$\frac{dX}{dt} = F(X) \text{ in } \Omega \times [0,T],\ X(0) = X_0 \text{ in } \Omega,\ +\text{Boundary Conditions}, \tag{2.1}$$

where F is a nonlinear differentiable operator describing the dynamics and X_0 is the initial state. System (2.1) is supposed to have a unique solution in $\mathcal{X}$. Then one introduces observed data $X_{obs} \in \mathcal{X}_{obs}$ into the numerical model by the mean of a cost function $J : \mathbb{R}^N \longrightarrow \mathbb{R}$ such that

$$J(X_0) = \frac{1}{2} \int_0^T ||CX - X_{obs}||^2 dt, \tag{2.2}$$

the observation operator C maps $\mathcal{X}$ to $\mathcal{X}_{obs}$. It is obvious that the solution X_0^* of the minimization problem (2.3) is the initial state such that the solution X of (2.1) fits at best the observed data in [0,T].

$$\text{Find } X_0 \text{ such that} \quad \nabla J(X_0) = 0 \tag{2.3}$$

There exists two methods for the computation of the gradient of J appearing in (2.3), we restrict our purpose to the adjoint method.

2.2.2 Reverse mode of differentiation

Let the adjoint variable $\widehat{X}$ be the solution of the *adjoint system*

$$\begin{cases} \dfrac{d\widehat{X}}{dt} + \left[\dfrac{\partial F}{\partial X}(X)\right]^* \widehat{X} = C^*(CX - X_{obs}) \quad \text{in } \Omega \times [0,T], \\ \widehat{X}(T) = 0 \quad \text{in } \Omega, \ + \text{ Boundary Conditions}, \end{cases} \tag{2.4}$$

then one proves [35] that the gradient of J is given by

$$\nabla J = -\widehat{X}(0). \tag{2.5}$$

From a computational point of view, the adjoint code is integrated backward in time along a trajectory which is formed of all the values of the variables of the direct code. Hence a first run of the direct code that enables to supply a trajectory may be done before for the adjoint integration.

2.3 Differentiation of Meso-NH

Meso-NH is differentiated with respect to the state variables including 3 wind velocity components, dry potential temperature, turbulent kinetic energy, mixing ratios of water, N passive source terms, and the pressure. The model is evaluated using explicit finite differences schemes in both time and space.

We choose to differentiate the code with the automatic tool Odyssée [18] because it enables us to generate the adjoint source code of a fortran source code, the differentiation work is described in [12]. However the size of the executable file of the adjoint code generated by Odyssée is sometimes so large that it cannot be run. This problem is essentially due to the management of the trajectory which is locally computed and saved in intermediate variables. In order to improve the codes generated by Odyssée one proceeds to several changes:
- detection of the linear parts of the direct code, this reduces the amount of storage required for the trajectory;
- storage of trajectories on files. This allows to avoid the redundant computations generated by the automatic differentiator.

In order to easily maintain the MesODiF package, we decide to manage a direct/tangent code according to three arguments:

- only state variables are stored every time-steps in the main routine;

- local variables are recomputed at each time-step.

2.4 Special issues of the differentiation of MESO-NH

2.4.1 Checkpoints schemes for the storage of trajectories

When the trajectory occupies a too large amount of memory, a solution lies on the implementation of algorithms such as the Griewank's checkpoints method [25]. In that case the calculation of the adjoint code is split and done part by part from restart points called *checkpoints*. For a general purpose the user's arbitration is essential to choose between time and memory consumptions since the choice of a checkpoint scheme depends on the computer model, the computer and the aim of the simulation. This becomes a crucial problem for operational (real time) weather forecasts. To overcome the problem we propose the TwiCe algorithm [11].

2.4.2 Differentiation of a Leap-Frog scheme

See [11].

2.4.3 Linear solver

The Fast Fourier Transform routine (in Meso-NH) is a linear self-adjoint operator that does not require differentiated routines. Arguments to avoid the differentiation of the solver are given in [23] and [17].

2.5 Numerical results

2.5.1 Performances of the differentiated codes

The "mountain wave" two dimensional simulation takes place on a computational domain (180 km×15 km) containing a bell shaped mountain with a height of 500 m and a half width of 10 km (Figure 3). The domain is discretized by 91 points in the horizontal plane that are duplicated 60 times in the vertical plane. The duration of the simulation is of 2000 s with a time-step of 20 s.

	direct code	tangent code	adjoint code
ratio $(./C_D^T)$	1	1.99	2.02

Table 2.1: Time ratios for the differentiated codes of Meso-NH

The ratios presented in Table 3 are lower than the theoretical bounds [31]. However most of the trial meteorological simulations described in [34] do not run without a checkpoints scheme. Fortunately suitable checkpoints algorithms exist, for example the ratio between the CPU time of the calculation of one gradient with the adjoint code of Meso-NH and the CPU time of the evaluation of the direct model is equal to:

- 3 when the amount of memory is sufficient to store all the trajectory,
- 4 when *TwiCe* is used,
- 5.6 for the 3D simulation described in [34] when using the Griewank's *Treeverse* subroutine.

2.6 Conclusions

Meso-NH is now designed for gradients computations that enable sensitivity analyses, forecast experiments and parameters identifications. The linearized codes are efficient in term of time and memory consumptions. For example a run of the adjoint code only requires 3 times the time required for a run of the Meso-NH model as soon as the trajectory is entirely stored in memory. Otherwise, check-points algorithms are used and simulations with a physical meaning can be tackled. In particular, an adjoint integration computed with the TwiCe scheme is only 4/3 more expensive than an adjoint integration realized in a straightforward manner.

Acknowledgments: This work was supported by the INRIA action for Operative Inverse Mode, the CEMRACS'97 and the IDRIS computational center.

Reduction of Storage of Variables in Adjoint Code

Mohamed Tadjouddine

Mohamed.Tadjouddine@sophia.inria.fr

Projet Safir, INRIA
2004, Route des Lucioles BP 93
06902 Sophia Antipolis France

The computation of derivatives by automatic differentiation in reverse mode requires storage or recomputation to transpose the computational graph. One of the methods consists in storing all the modified variables. However, this method is costly in terms of memory requirement. An extra knowledge on the code, for instance a given loop is parallel, may lead to avoid the storage of some modified variables. The program analysis appears like a way for automatically generating more efficient code.

To avoid useless storages, we propose to use array region analysis. This analyze relies on the fact that it might useful to store a variable that is Written after Read (WaR) according to the control-flow of the program. In order to refine this criterion, we analyze individual elements of array variables. Our method consists in computing and handling `Read` and `Write` regions of arrays along the hierarchical control flow graph of the program. Obviously, this array region analysis is conservative. It enables us to extract, from the input program, information on access array regions or flow between statements or procedures. But, this analyze is not sufficient. An additional information must be associated to each potential region of array to be stored. This information determines if a WaR variable is really used in the derived code.

3.1 Introduction

Adjoint models are used in many fields of science such as meteorology or oceanography. Optimizing a model in order to fit consistent model prediction and Sensitivity analysis are two examples of applications of adjoint models. As discussed in [22], if we consider a physical model, its adjoint model can be constructed from:

- The analytical equations by using analytical adjoint operators,
- The discrete model equations by using Lagrange Function and Euler-Lagrange operators,
- The numerical program of the model developed in a programming language by using Automatic Differentiation (AD).

In short, AD is a solution to get fast and accurate derivatives of a function represented as a program. The AD technique is based on two fundamental principles : any program can be seen as a composition of elementary functions and can then be differentiated using the chain rule. There are 2 modes of AD, both having predictable complexity:

1. The forward mode in which the intermediate derivatives are computed in the same order as the program computes the composition.
2. The reverse mode in which the intermediate derivatives are computed in the reverse order. The reverse mode is optimal for computing gradients because its complexity is independent from the number of input variables [27].

Suppose a physical phenomenon is modeled by a numerical code in a high level programming language such as Fortran or C. The adjoint model is obtained by differentiating the original program by the reverse mode of AD. Actually, the reverse mode of AD uses the adjoint operator (transposition) to inverse the computational graph of the program. Therefore, the `Write` variables become `Read` variables and vice versa. This method consists of two phases: The first one where the trajectory is computed is straightforward. The second one consists in computing the derivative quantities in the reverse order as the original program does.

The construction of adjoint code is done by 2 ways: Operator overloading or Source-to-source transformation code. Here, we use the second technique over Fortran 77 programs. A way of computing the trajectory is to save all the modified (written) variables. However, this method is costly for actual applications in terms of memory requirement. Our aim is to reduce the storage requirement by avoiding useless storages. For that account, we use an array region analysis, which permits us to safely determine the set of indices of an array program that may be stored.

3.2 A motivating example

Suppose we want to compute the derivative of the following piece of code by the reverse mode of Automatic Differentiation:

```
s = 0.
do i=1, 10
  s = s+x(i)*y(i)
end do
```

If we store all the modified variables, we will generate for example the following derivative code:

```
C Trajectory
      s1 = s
      s = 0.
      do i=1, 10
        s2(i) = s
        s = s+x(i)*y(i)
      end do
C Transposed linear forms
      do i=10, 1, -1
        s = s2(i)
        xcl(i) = xcl(i)+scl*y(i)
        ycl(i) = ycl(i)+scl*x(i)
      end do
      s = s1
      scl = 0.
```

We observe that we did not need to save the values of the variable s because they are not used during the transposition of the computational graph. To determine the `Write` variables that may be stored, we use the criterion below:

Criterion 3.2.1 *A variable may be stored if it is Written after Read (WaR). In other words, there is a flow dependence from the* `Read` *to the* `Write` *of the variable.*

The following example (a straight line code) shows that we have only to *save* the `Write` variable x just before the second statement:

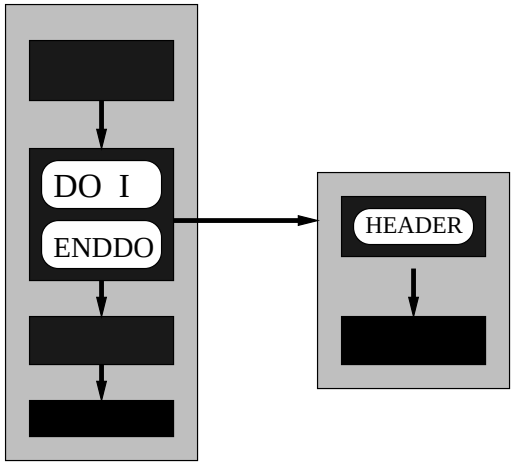
```
y = x**2                    C Trajectory
x = 2.+z                       y = x**2
y = z*2.          ⟶           s1 = x
x = 2.-z                       x = 2.+z
                               y = z*2.
                               x = 2.-z
```

In the preceding example, the variables are scalar. However, what may be happen if the variables of the program are arrays?

3.3 Array region analysis

The array region analysis is motivated at least by two reasons: First, Fortran 77 works only on static allocation. Second, many numerical codes operate on sparse matrices. Then, it is useful to analyze access regions of arrays to avoid useless storages. To analyze access regions, we use the Hierarchical Control Flow Graph (HCFG) [14]. Figure 1 shows an example of HCFG:

Figure 3.1: Example of HCFG



This graph looks like the abstract syntax tree of the program but contains more informations. A node represents a basic block, which is a sequence of atomic instructions. An edge represents a control dependence between two basic blocks. With this graph, we can get informations about enclosing loops of a statement or still bounds of a given loop.

3.3.1 Approximations

Because of high complexity or undecidability, we cannot always compute exact regions. We use conservative approximations. In other words, we compute exact regions and we switch to approximate regions when necessary. To give the semantic of array regions, let us consider the following notations:

Σ	is the set of memory stores
$\wp(\mathbb{Z}^d)$	is the set of parts of $\mathbb{Z}^d$
$\widetilde{\wp}(\mathbb{Z}^d)$	is a set such that $\wp(\mathbb{Z}^d) \subset \widetilde{\wp}(\mathbb{Z}^d)$ and $\varnothing, \mathbb{Z}^d \in \widetilde{\wp}(\mathbb{Z}^d)$

For a d-dimensional array accessed by a statement S, the semantic function of its region is defined as follows (see [13]):

$$\mathcal{R} : S \to (\Sigma \to \wp(\mathbb{Z}^d))$$

For our application, we need over-approximate region whose the semantic is the following:

$$\overline{\mathcal{R}} : S \to (\Sigma \to \widetilde{\wp}(\mathbb{Z}^d))$$

3.3.2 Simple regions

We consider a class of codes consists of DO-loops with affine loop bounds, whose bodies consist of accesses to scalars and arrays with affine subscripts. We define simple region as a way to represent the set of indices of arrays accessed by the program.

Definition 3.3.1 *A simple region is a Cartesian product of regular intervals. A simple region S is denoted by $\times_{j=1}^{n}[l_j : u_j : p_j]$ where $[l_j : u_j : p_j]$ is the integer set going from l_j to u_j by increments of p_j.*

All other forms of subscripts of arrays are over-approximated by the size authorized by the declaration. With the help of simple regions, we describe an access region R of an array as follows (see [44]):

$$R \ ::= \ \text{Simple region} \ \mid R \cap R \ \mid R \cup R$$

Of course, these expressions are evaluated for integer bounds and simplified for symbolic bounds.

3.3.3 Operators

In order to summarize the effects of the program on the arrays, we use the operators Union and Intersection. These operators are used by the array region analysis itself.

Union: Because the set of simple regions is not closed under Union ($\cup$), we need the over-approximate Union ($\overline{\cup}$). This over-approximate operator relies on the following operations:

- $[l : u] \,\overline{\cup}\, [l' : u'] = [\min(l, l') : \max(u, u')]$
- $a + b\mathbb{Z} \,\overline{\cup}\, c + d\mathbb{Z} = \min(a, c) + \gcd(b, d, a - c)\mathbb{Z}$

The operator Union allows us to summarize access regions along the control flow of the program.

Intersection: Intersection is an internal operator that is exact in the set of simple regions. We compute the intersection of 2 simple regions by using the following operations:

- $[l : u] \ \cap \ [l' : u'] = [\max(l, l') : \min(u, u')]$
- $a + b\mathbb{Z} \cap c + d\mathbb{Z} = p + \mathrm{lcm}(b, d)\mathbb{Z}$ where $p = a + bx_0 = c + dy_0$ with (x_0, y_0), a solution of the diophantine linear equation $a + bx = c + dy$.

Intersection enables us to test the dependence between two regions of the same array.

3.4 Applications

In order to point out the WaR criterion, we use a classical abstraction encountered in dependence analysis [49]. This is the dependence distance. The dependence distance between iterations I and I' is defined as the difference $d(I, I') = I' - I$. With the help of this abstraction, we may safely determine if an occurrence of an array is executed before an other one. The WaR criterion between a `Write` $A(I)$ and a `Read` $A(J)$ is satisfied if $A(I)$ is executed before $A(J)$ and the access regions of $A(I)$ and $A(J)$ have common elements.

To measure the performance of the WaR criterion, we have compared it with a standard strategy, which consists in saving all the modified variables by treating arrays as atoms, over some examples. Suppose A, B, and C are declared as $nmax \times nmax$-dimensional arrays, v a $nmax$ vector, and the programs P_1, P_2, P_3, and P_4 are built by the following formulas in which $n < nmax$:

$$\begin{aligned}
P_1: \quad p &= \prod_{i=1}^{n} v(i) \\
P_2: \quad c_{ij} &= \alpha a_{ij} + \beta b_{ij}\,; \quad 1 \le i, j \le n \\
P_3: \quad c_{ij} &= \sum_{k=1}^{n} a_{ik} * b_{kj}\,; \quad 1 \le i, j \le n \\
P_4: \quad a_{ij} &= \sum_{k=1}^{n} a_{ik} * b_{kj}\,; \quad 1 \le i, j \le n
\end{aligned}$$

Actually, n can be less than the half of $nmax$ in real examples. Table 1 shows us that, for certain codes, the WaR criterion can reduce dramatically the memory requirement. It avoids some useless storages and then reduces the sizes of declarations during the derivative code generation by AD.

Table 3.2: Comparison WaR/Standard

`Programs`	`Size of Declaration`		`Number of Storages`	
	Standard	WaR	Standard	WaR
P_1	$nmax$	n	n	n
P_2	$nmax^2$	$\emptyset$	n^2	$\emptyset$
P_3	$nmax^3$	$\emptyset$	n^3	$\emptyset$
P_4	$nmax^3$	n^3	n^3	n^3

3.5 Conclusion

This array region analysis relying on conservative approximations allows us to reduce the cost of adjoint code in terms of memory requirement. However, the WaR criterion used here, sometimes causes to save useless storage. Extra informations can be used such as the detection of non linear forms of expressions in the program.

Writing the adjoint of a parallel model using Odyssée

Yannick Tremolet

Yannick.Tremolet@noaa.gov

Environmental Modeling Center
National Centers for Environmental Prediction
W/NP2, WWB, Room 207, NOAA
5200 Auth Rd
Camp Springs, MD 20746, USA

Adjoint models are the basis for variational data assimilation both in meteorology and oceanography. Since these applications require a lot of computational power and memory, in the past few years many models have been developed for distributed memory parallel computers. We will present the methodology used at NCEP to derive the adjoint of the parallel spectral model which was written using a message passing library.

We will describe the impact of the parallelism in the code on the derivation of the adjoint and show how to obtain the adjoint of a code containing explicit message passing instructions, including point to point communications and group or global communications. We will then show how the automatic differentiation tool Odyssée can be used for that purpose. Finally, we will give some validation results of the derived code.

4.1 Introduction

Adjoint models are the basis for variational data assimilation both in meteorology and oceanography. But, until the recent years, variational data assimilation algorithms have not been used operationally because of its computational cost. In the past few years many models have been developed for distributed memory parallel computers [16, 38]. Our aim in this paper is to show how to obtain the adjoint of a code containing explicit message passing

instructions, including point to point communications and group or global communications. The results presented here can be applied to any code written using a message passing library such as MPI or a machine dependent library such as SHMEM.

Variational data assimilation consists of minimizing the discrepancy between observations of the atmosphere or of the ocean and the solution of the model for a period of time where observations are available. In order to perform the minimization, we need the gradient of the cost function which measures this discrepancy, the adjoint model is used for that purpose [35].

A description of the NCEP global spectral model which was used in this experiment can be found in [42]. The parallel version of the model described in [46] is written in Fortran 77, the communication subroutines have been encapsulated for portability reasons and are available in two versions using MPI or SHMEM. The code uses BLAS and FFT libraries and approximately 30000 lines of code have to be differentiated. The current operational resolution is T126 with 28 levels. Table 4.3 gives typical values of the number of degrees of freedom (active variables) memory and computational requirements for that resolution and other resolutions of interest. This table shows the size of the problem to be solved and the necessity of using parallel supercomputers.

Spectral truncation	T62	T170	T254	T340
Vertical resolution	28	32	48	64
Grid size	192x94	512x256	768x384	1024x512
Active Variables	455 616	3 794 148	12 599 040	29 971 854
Processors	28	128	480	512
Memory/Proc. (Mb)	6	18	26	59
Performances (Gflops)	1.1	7.7	29	42

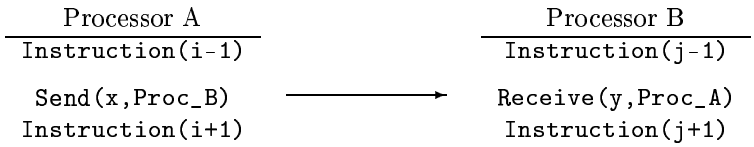
Table 4.3: Requirements and performances of the NCEP global spectral model on Cray T3E-600.

4.2 Adjoint of a parallel code

The adjoint of the computational parts of a parallel code are obtained using the same techniques as in a sequential code. In this section we will show how to transpose the parallel instructions a code may contain.

4.2.1 Point to point communication

The simplest function a parallel code may use is sending a data from one processor to another. Consider a code in which processor A sends the value of the variable x to processor B which receives it in the variable y, this may be written as:



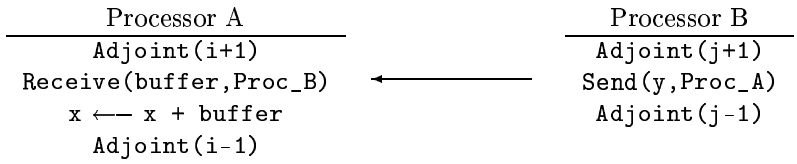
In a more concise form, we can also write:

```
y[B] = x[A]
```

where the bracketed letter represents the processor which owns the variable. The adjoint of this operation would be:

```
x[A] = x[A] + y[B]
```

This means the value of `y[B]` has to be sent from processor B to processor A and added to `x[A]`. This gives the code:



The adjoint of sending the value of a variable from one processor to another is to send the value of the adjoint of this variable in the other direction. More precisely, the adjoint of a `receive` instruction is a `send` of the same variable (if we apply the usual rule of keeping the name of a variable for its adjoint and giving another name to the initial variable) to the same processor. The adjoint of a `send` instruction is a `receive` instruction from the same processor and the addition of the received value to the variable which was sent.

4.2.2 Group communications

We can also determine the adjoint operations corresponding to the usual group communications that may be used in a parallel code: broadcast, reduction, scatter and gather.

Consider a code which contains a reduction operation, i.e. each processor owns a variable `x`, all those values have to be combined, the result being stored on one processor P_0. In such a code, each processor P_i would compute its own `x`, then all processors call the reduce function and finally the processor P_0 uses the result of this reduction. For example, the reduce function can be a sum:

$$s = \sum_{i=1}^{n} x_i.$$

We know that the adjoint of this operation is:

```
DO i=1,n
  x(i) = x(i) + s
ENDDO
```

Thus in the adjoint code P_0 would compute `s` (adjoint of P_0 uses `s`), then each processor would add this value of `s` to its own `x` and finally each processor would use its `x` (adjoint of computing `x`). The adjoint of

```
Reduce_sum(Proc_0, x, s)        is        Broadcast(Proc_0, s)
                                          x = x + s
```

Another way of obtaining the same result would be to consider that each processor sends its value of `x` to P_0 which performs the sum and uses it. We already know how to transpose a single data transmission and the result is easy to obtain. But this approach supposes a particular implementation of the reduce operation while others are possible (using a binary tree for example). Following the same model, we can deduce the adjoint of the other usual group operations, table 4.4 gives the results.

Operation	Direct Code	Adjoint Code
Send	`Send(x,Proc)`	`Receive(buffer,Proc)` `x ⟵ x + buffer`
Receive	`Receive(x,Proc)`	`Send(x,Proc)`
Synchronization	`Synchro(List_Proc)`	`Synchro(List_Proc)`

Broadcast	`Broadcast(x,List_Proc)`	`For Proc ∈ List_Proc`
		`  Receive(buffer,Proc)`
		`  x ⟵ x + buffer`

Reduction	`For Proc ∈ List_Proc`
	`  Receive(buffer,Proc)`
	`  x ⟵ x + buffer`
		`Broadcast(x,List_Proc)`

Scatter	`Scatter(X,P_0,List_Proc)`	`Gather(buffer,P_0,List_Proc)` `X ← X + buffer`

Table 4.4: Adjoint of message passing operations.

One interesting result shown in this table is that the quantity of data exchanged by two given processors is the same in the direct and adjoint codes, only the direction changes.

4.3 Application using Odyssée

4.3.1 Writing the code

The manual development of the adjoint of the previous version of the model took approximately two years. To avoid this long and error prone task we decided to use the automatic

differentiation tool Odyssée [18], even though it cannot yet produce adjoints of parallel codes. Since all the communications in the model are encapsulated in a relatively small number of subroutines, it is possible to create manually entries for these subroutines in the Odyssée data base. This information allows the system to perform a correct dependency analysis. The same methodology is used to provide information about the library calls for which we don't have a source code. The adjoints of the communication subroutines and library calls are then added manually in the code produced by Odyssée.

The results presented in this paper were obtained by differentiating a simplified version of the model with no physics. The model solves the dynamics equations using a semi-implicit time integration scheme. The number of degrees of freedom is the same as in the diabatic model and allows to perform significant experiments.

Resolution	T62-28	T170-32	T254-48	T340-64
Non linear code	6	18	26	59
Linear (Odyssée)	9	27	39	83
Adjoint (Odyssée)	1100	6200	5700	24700
Adjoint (Optimized)	9	27	39	83

Table 4.5: Memory requirements in Mb per processor for the codes generated by Odyssée and then optimized manually.

Odyssée produces a Fortran 77 source code for the adjoint but because of memory requirements, it was, at first, impossible to run that code. Table 4.5 shows that, even for a low resolution, it would require too much memory. The examination of one subroutine, the subroutine computing the non linear terms in the dynamics equations, shows that the adjoint code generated by Odyssée contains almost 5 times more lines than the non linear subroutine (see table 4.6). A more detailed examination of the code shows that it contains unnecessary computations and saves many variables which are not re-used later in the program. It is necessary to remove these lines manually, this almost reduces the number of lines by a factor of two and reduces the amount of required memory to less than twice the memory used in the non linear code.

4.3.2 Validation

After removing these lines and inserting the adjoint of the communication and library subroutines, the code can be validated. The tangent linear model is tested by comparison with a finite difference computation using the non linear model. Table 4.7 shows that the tangent linear model is correct given the precision of the machine. The adjoint is validated using the following property:

$$\langle x, L^* L x \rangle = \langle Lx, Lx \rangle$$

Subroutine	Number of lines
Non linear	137
Linear (Odyssée)	289
Adjoint (Odyssée)	622
Adjoint (Optimized)	348

Table 4.6: Code complexity using Odyssée version 1.6 for one subroutine.

where L represents the linear model and L^* the adjoint model. This test was correct up to the 14th digit at the resolution of T62 with 28 levels integrated over a 6 hours period which is in agreement with the precision of the computer.

α	$\frac{\|F(X+\alpha.\delta X)\|-\|F(X)\|}{\alpha\|dF(X,\delta X)\|}$
1.E+01	2.0085639584235575
1.E-01	1.0081838481942076
1.E-03	1.0000816532699321
1.E-05	1.0000008164164176
1.E-07	1.0000000017818156
1.E-09	0.9999994058365377

Table 4.7: Test of the linear model (T62-28) with 28 processors.

4.4 Conclusion

Using Odyssée to write the adjoint of a numerical weather forecast model is not yet fully automatic. Some steps have to be carried out manually such as the description of the communication subroutine for which the system cannot perform the dependency analysis and the removal of unnecessary computations and saving of variables. For the computational part of the code, Odyssée can perform the dependency analysis and can reliably reverse the order of the lines and transpose each line, which are very long and error prone tasks. It is interesting to note that Odyssée generates correct calls to the adjoint of the communication and library subroutines, only the adjoint subroutines have to be provided.

The adjoint obtained with Odyssée is currently being used to test some minimization algorithms in order to implement a four dimensional variational data assimilation algorithm. The adjoint of the physics packages of the model has to be included in this code, since these subroutines are long and evolve often, Odyssée should bring even more to the process of developing and testing their adjoint. In the same time, strategies for storing or recomputing the trajectory and the performance of the generated code on a parallel computer have to be evaluated.

Some other tools may be useful for the development of the adjoint of large codes such as the possibility to include different strategies for saving or recomputing the trajectory, recognizing linear computations in the generation of the adjoint and the removal of unnecessary code. But this experiment shows that the adjoint of a parallel code can be developed relatively easily using the latest automatic differentiation tools. This allows scientists to experiment with 4DVAR algorithms using the most powerful computers available and to focus on the important aspects of the problem rather than writing the adjoint line by line.

Comparison of automatically generated code for evaluation of first and second order derivatives to hand written code from the Minpack-2 collection

Ralf Giering and Thomas Kaminski

ralf@trough.mit.edu

Center for Global Change Science
Department of Earth, Atmospheric, and Planetary Sciences
Massachusetts Institute of Technology
Cambridge, MA 02139, USA.

kaminski@dkrz.de

Max-Planck-Institut für Meteorologie
Bundesstr. 55, 20146 Hamburg, Germany

Adjoint models are increasingly being used in computational fluid dynamics (CFD), in particular in meteorology, oceanography, and climate research. Typical applications are data assimilation, model tuning, and sensitivity analysis. Both data assimilation and model tuning derive a set of control variables that achieves an optimal degree of consistency between simulated and observed quantities. Thereby the degree of consistency is quantified by a scalar valued misfit or cost function, which is defined trough the (usually large and

complex) numerical model of the system under consideration. The cost function can be minimized most efficiently by use of powerful iterative gradient algorithms [24], if first order derivatives can be provided. Applying the reverse mode of automatic differentiation (AD) adjoint code evaluates this first order derivative or gradient (see introductory section). To analyze the uncertainties in the inferred optimal values of the control variables, second order derivates of the scalar valued cost function are of interest. Since, usually, the number of control variables is large, evaluation of the full second order derivative, i.e. the Hessian matrix, is prohibitively expensive. However, Hessian vector products are relatively cheap and provide a module to evaluate certain properties of the Hessian matrix. For example the best constrained directions are the leading eigenvectors of the Hessian matrix and can be determined iteratively by Lanczos type algorithms.

In practise, these adjoint applications are based on models that have been previously developed and applied for simulation of the system under consideration, i.e. the designers of these models did not necessarily have adjoint applications in mind. Typically these models are written in Fortran, more precisely some Fortran dialect in between Fortran 77 and Fortran 90, with a recent tendency towards Fortran 90. These models typically run on super computers close to the limit of resources in terms of both memory and CPU time. Since the abovementioned applications (except for sensitivity analysis) require multiple runs of the adjoint models, it is obvious that efficient use of computer resources by the adjoint code is a necessary condition for executing the generated adjoint models.

During the eighties and early nineties adjoints of CFD models have been hand coded. This task, however, is extremely error prone and time consuming. Furthermore the strategies that have been used made the adjoint code inflexible to changes in the model code. As a consequence, development of adjoint models was rare and usually limited to simplified models [47, 41]. The adjoint of the atmospheric model applied for (4d-var) data assimilation at the ECMWF constitutes an exception: it has been constructed and is maintained by hand. However construction of the adjoint seems to have taken almost a decade and has started before AD tools were well enough developed to tackle this challenge. Code for evaluation of second order derivatives, as a consequence of its even larger degree of complexity, has not been hand written for large scale applications [6].

Recently a number of AD tools are being developed that are capable of generating adjoint code (Odyssée [40], GRESS [29], TAMC [20], see also other contributions to this document). Other tools operating in reverse mode are employing operator overloading capabilities of C^{++} or Fortran-90 (ADOL-C, AD01, ADOL-F, IMAS, OPTIMA90) [4].

TAMC (Tangent linear and Adjoint Model Compiler, [20]) is a source-to-source translator for Fortran programs to generate derivative computing code operating in forward or reverse mode. The internal algorithms are based on a few principles suggested e.g. by Talagrand [45]. These principles can be derived from the chain rule of differentiation [21]. TAMC applies a number of analyses and code normalizations similar to those applied by optimizing compilers (constant propagation, index variable substitution, data dependence analysis). In addition, given the top-level routine to be differentiated and the independent and dependent variables, by applying a forward/reverse data flow analysis TAMC detects all variables that depend

on the independent variables and influence the dependent variables (active variables). This is in contrast to operator overloading based tools, where the user has to determine active variables and to declare them to be of a specific data type. TAMC can handle all but very few relevant Fortran 77 statements and an increasing number of Fortran 90 extensions, check the latest manual version on the TAMC home page [19] for the current state of development.

Recently, TAMC has been successfully applied to generate the adjoint codes of an increasing number of large and complex CFD codes [32, 43, 48, 15]. A mayor challenge of adjoint code is providing intermediate results required, e.g. to evaluate derivatives of non linear operations. Efficient adjoint code uses a combination of recalculating and restoring from a tape written previously; both strategies can be applied by TAMC. For generation of recalculations a reverse data flow analysis is applied, and, as far as possible, only statements being absolutely necessary are inserted into the adjoint code. Concerning this key issue for generation of efficient derivative code, TAMC is unique among the AD tools. For the abovementioned applications checkpointing schemes have been implemented semi automatically by TAMC. The checkpointing technique allows to use the available resources for storing intermediate results more efficiently at the cost of an additional model run and is indispensable for these large applications [25]. For some applications even a multi level checkpointing is necessary. Depending on the level of checkpointing, the run time of the adjoint code is in between a factor of 3-6 of that of the model. Thereby the pure derivative code (without the additional model evaluations) is in between a factor of 1-3 of that of the model. See the TAMC home page [19] for more details on the adjoints of these models.

TAMC generates code to compute second order derivates operating in the so-called forward over reverse mode (FOR), i.e. the first order derivative is computed in reverse mode and the second order derivative in forward mode. The constructed code computes Hessian times vector products or the full Hessian. Alternative approaches use the forward over forward mode (FOF) or Taylor series expansion (TSE) [1]. For scalar valued functions FOR is much faster, and the relative run time is independent of the number of control variables, while the cost of FOR and TSE increases with this number. In theory, a relative run time below 10 should be attainable [26].

Although for the abovementioned applications, in theory, adjoint models also could have been hand coded, probably, in practise, without AD none of these applications would have been possible. This means in particular that there exist no hand coded counterparts to compare the automatically generated adjoint code to in terms of efficiency. Hence, for this purpose we employed the Minpack-2 test problem collection [2]. For each problem the collection contains hand written code to compute a scalar valued function, its gradient, and the product of its Hessian times a vector. The number of independent variables can be chosen arbitrarily.

We selected six problems that are representative of small to medium scale optimization problems arising from applications in superconductivity, optimal design, combustion, and lubrication. Table 5.8 gives the list of problems and their number of Fortran code lines.

The code for function evaluation has been differentiated by TAMC to generate code for evaluation of the gradient (adjoint code). The comparison has been carried out on two

name	lines	short description
ept	51	elastic-plastic torsion
ssc	54	steady state combustion
pjb	61	pressure distribution in a journal bearing
gl1	70	Ginzburg-Landau (1-dimensional) superconductivity
msa	90	minimal surface area
gl2	111	Ginzburg-Landau (2-dimensional) superconductivity

Table 5.8: Names of Minpack-2 problems and their number of code lines

machines, a Sun Ultra-1 and a Cray C90. To allow a fair comparison on the Cray C90, the performance of the hand written code has been improved by inserting vectorization directives and moving conditional statements out of the inner most loop. The codes have been compiled by the vendors Fortran compiler with the precision and compiler options given in Table 5.9.

platform	precision	Fortran command line
Sun Ultra-1	double precision	f90 -O2
Cray C90	double precision	f90 -O inline3,scalar3,vector3,task0

Table 5.9: Precision and compiler options used on platforms.

The results for evaluation of the gradient codes are depicted in Fig. 5.2 for Sun Ultra-1 and in Fig. 5.3 for Cray C90. For every test problem the relative run time, i.e. the run time of the gradient code compared to the run time of the function code, has been calculated for different numbers of independent variables. On Sun Ultra-1 the hand written code is in four cases slower than the TAMC generated code (GL2,SSC,GL1,EPT). However, a remarkable difference can only be seen for the GL2 problem, in all other cases differences are small. A nested loop in the function computing code of GL2 is split into three loops in the hand written gradient code: one for interior points of the domain and two for boundary points. This has been common practice in hand written adjoint codes. In contrary, TAMC does not split the loop; instead interior and boundary points are handled simultaneously as is implied by strict application of the rules TAMC is based on [21]. In all cases, the changes of the relative run time with the dimension of the problems (the number of independent variables) are very small. On a Sun Ultra-1 performance is compromised by cache misses. Their number depends mainly on the memory needed for all variables in a loop compared to the cache size. For non-linear operators, this ratio is different for function and gradient code. This explains the spikes at certain problem sizes.

The differences in relative run time are also small on a Cray C90, except again for the GL2 problem. Here, in most cases, the relative run time increases slightly with the problem

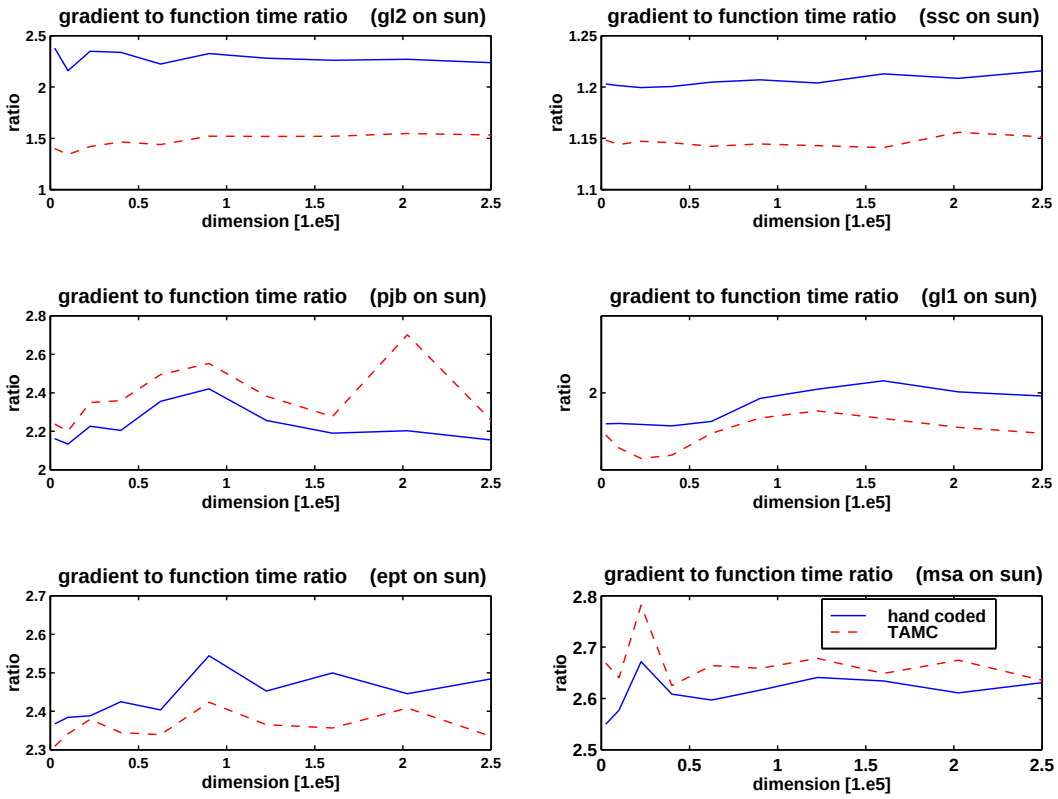


Figure 5.2: Relative run time of gradient code on Sun Ultra-1 (x-axis is the number of control variables).

size.

Some recalculations in the adjoint code are independent of the problem size. If they, for small sizes, constitute a mayor part of the whole calculations the ratio is almost one. For large sizes the run time of the adjoint code is dominated by updating adjoint variables. Thus, the ratio depends on the complexity of the non-linear operations in the corresponding function code. On vector machines like the Cray C90 run time depends mainly on the efficient use of vector pipes. For these test problems the effective vector length increases with the number of independent variables. Thus, on a Cray C90, in contrary to the Sun Ultra-1, the abovementioned transition to dominance of updating adjoint variables is at higher problem sizes.

The Hessian times vector code has only been compared on the Sun Ultra-1. The results depicted in Fig. 5.4 show the relative run time of the Hessian times vector code compared to the run time of the original function code. Only in one case (GL2) is the TAMC generated

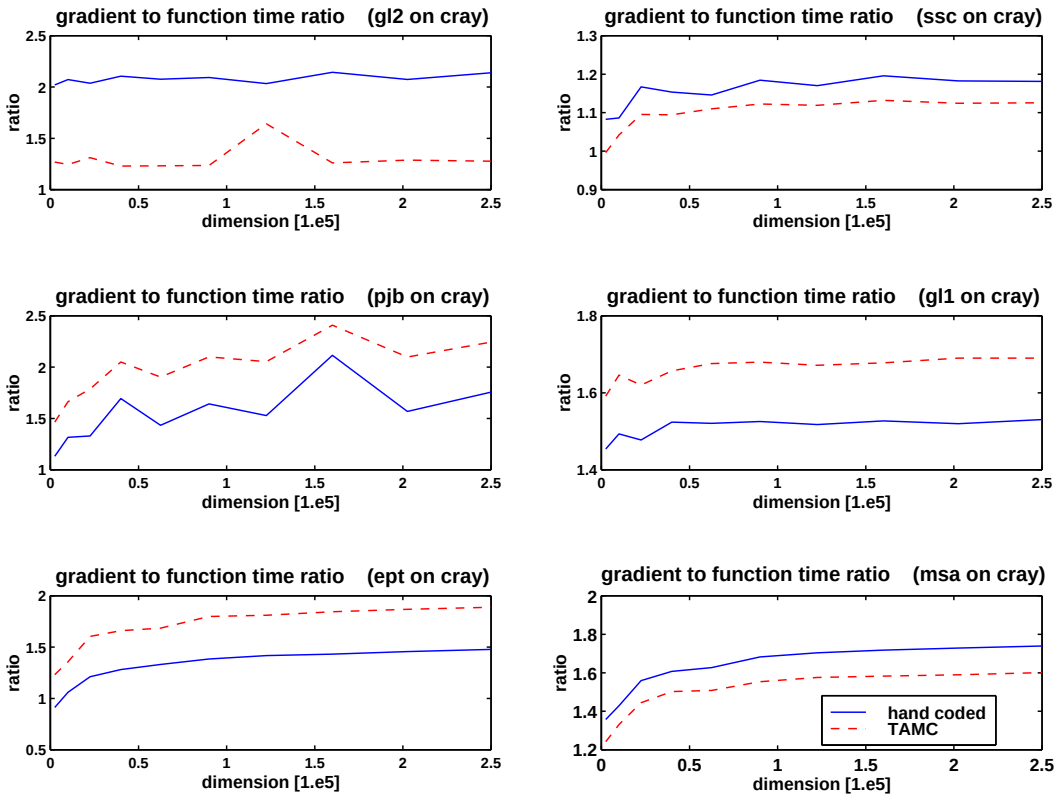


Figure 5.3: Relative run time of gradient code on Cray C90 (x-axis is the number of control variables).

code faster than the hand written code. As for the gradient code, the hand written version of the Hessian times vector code for the GL2 problem splits a nested loop into three loops. But the run time penalty for this splitting is much more pronounced: the TAMC generated code is about a factor 2 faster! For the other problems, the TAMC generated code is slower, because TAMC generates some initializations of adjoint variables to zero that could be omitted by combining them with subsequent assignments to the same variable. Although humans can easily detect these cases, automatization can become arbitrarily complex, because it might involve comparison of array subscript expressions.

In summary, the efficiency of TAMC generated adjoint code and Hessian times vector code is comparable to that of their hand written counterparts. In detail, the results depend on particular features of the computer and on the compiler that are used and also on details of the implementation of both the particular function to be differentiated and the hand

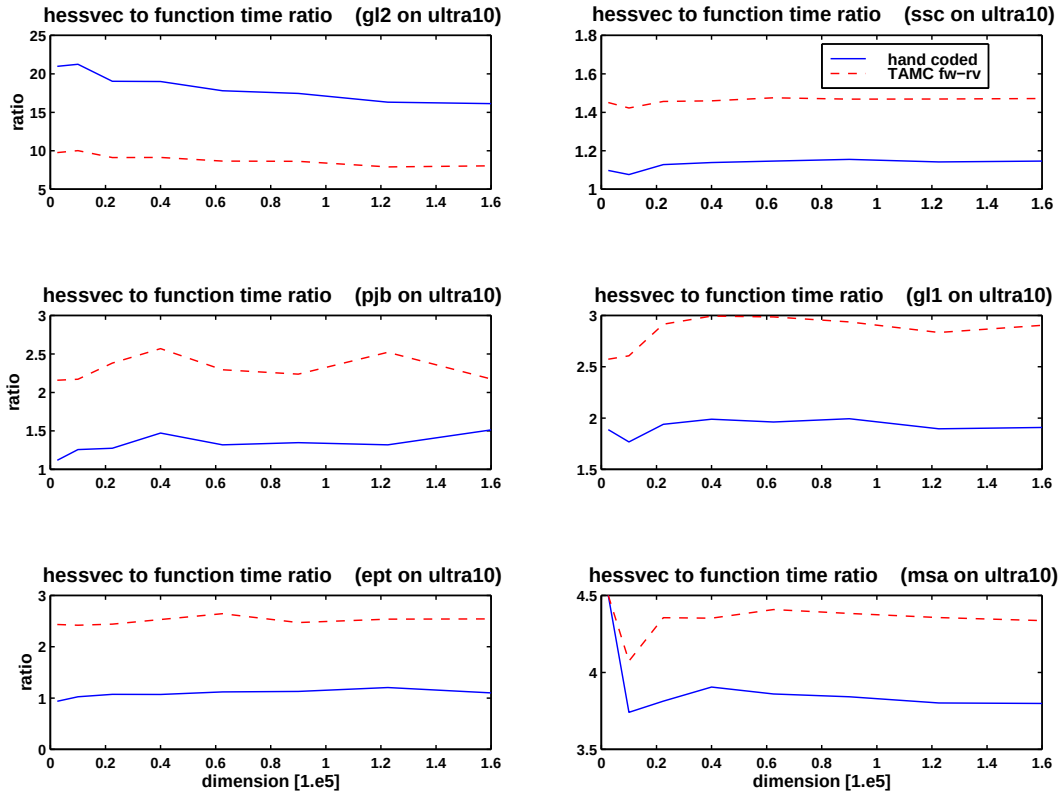


Figure 5.4: Relative run time of Hessian times vector code on Sun Ultra-1.

written derivative code. TAMC is available through its home page [19] or by electronic mail to its designer (ralf@sea.mit.edu).

A Hierarchical approach in automatic differentiation

Marco Mancini

mancini@parcolab.unical.it

Parallel Computing Laboratory
Department of Electronics, Computer Science and Systems,
University of Calabria
87036 Rende (CS) - Italy

Automatic Differentiation (AD) of computer programs is based on the systematic application of the chain rule; the associativity of the chain rule allows derivative propagation to be performed at arbitrary levels of abstractions (i.e., scopes of differentiation), such as binary operations, assignment, basic block, ... The higher is the scope of differentiation, the more efficient could be the code for computing derivatives.

Several techniques have been proposed to exploit the program structure and the associativity of the chain rule. Among them, the hierarchical approach to AD [7] seems to be a powerful methodology to design efficient AD algorithms. In this paper, we present a hierarchical approach to AD, based on the following features:

- The program graph is partitioned in extended basic blocks (EBB), where each EBB consists of *assignments* and *if statements*. For each block, information about the data flowing in and out is collected at compile time.
- Each EBB is differentiated ignoring surrounding computations and the EBB derivative sparsity patterns are exploited in order to reduce the derivative code complexity. The chain rule is used at this level of granularity to propagate the global derivatives with respect to the inputs of the program.
- A context-sensitive strategy has been used to choose between approaches working at different scopes of differentiation (assignment level, EBB level).

We have implemented the above approach by using AIF [8], a language-independent intermediate format that makes it easier to experiment with new differentiation algorithms. The computational results show performance gains of the proposed techniques compared with existing approaches.

6.1 Introduction

For a general function $f : \mathcal{R}^n \mapsto \mathcal{R}^m$, the reverse mode and the forward mode are not necessarily optimal strategies for generating its derivatives. The optimal strategy depends on the program structure and on the memory and time constraints. The hierarchical approach (HAD) described in [7] defines a methodology to exploit the program structure for generating more efficient derivative codes.

The HAD is based on partitioning the program graph at several levels of abstraction; for each "differentiation partition", the derivatives of this program fragments are computed ignoring surrounding computations and applying the chain rule at this level of granularity.

Whenever we can identify a piece of program whose number of input arguments is smaller than the number of independent variables for differentiation (i.e., interface contraction [30]), the independent differentiation of these program pieces is likely to decrease the derivative complexity in a global forward-mode approach. Context-sensitive strategies play a crucial role in order to detect an interface contraction.

Several information must be provided in order to apply an HAD strategy; the following information could be obtained by using data flow and dependence analysis (or by user directives):

- The number of variables that pass in and out of a program fragment; we denote by I the set of the input variables and by O the set of the output variables of a program segment.
- The number of floating-point operations, the degree of derivative sparsity of a program segment. This information is useful to estimate the derivative costs and to propagate derivatives.

For each code fragment, the HAD propagates derivatives following these two steps:

Step 1: Preaccumulation of local derivatives : We compute the derivatives $\frac{\partial v_o}{\partial v_i}, \forall v_o \in O, \forall v_i \in I$, considering the variables belonging to I to be independent.

Step 2: Accumulation of global derivatives : The global gradients of each v_o are accumulated. When using the forward mode for global propagation of derivatives, this is done as follows:

$$\nabla v_o = \sum_{v_i \in I} \frac{\partial v_o}{\partial v_i} \nabla v_i, \forall v_o \in O$$

```
Stmt1    v1 = a;
Stmt2    if ( flag ) v1 = b*2;
Stmt3    v2 = c*c;
Stmt4    w = v1*v2 + c;
Stmt5    z = v2*a;
```

Figure 6.5: Example code.

In order to establish whether a preaccumulation strategy is convenient, we can use an adaptive strategy that, based on a computational model, estimates the derivative cost, which depends on the particular structure of the program segment and on the number of global derivatives to be computed.

6.2 A HAD Algorithm

As an explanatory example of our HAD augmentation strategy, we consider the extended basic block in Figure 6.5, which we assume to be one partition of a more general program.

The sets representing the data flow information related to this program segment are the following:

- {a,b,c}: set of input variables.
- {w,z}: set of output variables.
- {v1,v2}: set of temporary variables whose value is not used thereafter.

Our strategy is based on a global forward mode approach. The local derivatives are computed by differentiating each assignment statement being in the extended basic block (e.g., they can be propagated by using the forward-reverse mode, such as in ADIC [8]) and then the global derivatives are propagated by using the chain rule in the forward mode.

During the preaccumulation of the local derivatives, we exploit the derivative sparsity patterns. Likely, most of the temporary variables and output variables of the code segment depends only on a subset of the input variables.

Taking into account the flow data dependences among the variables (the data dependence graph of the EBB in Figure 6.5 is reported in Figure 6.7), we can just propagate the derivatives that are not "zero". When *if-statements* are in the EBB, conservative assumptions are made during the data dependence analysis. Referring to the example code in Figure 6.5, the variable v1 can depend on b, according to the value of flag. If the value of flag is known only at run-time, we assume that v1 depends on b; thus, we propagate $\frac{\partial \mathtt{v1}}{\partial \mathtt{b}}$, even if it can be zero at run-time.

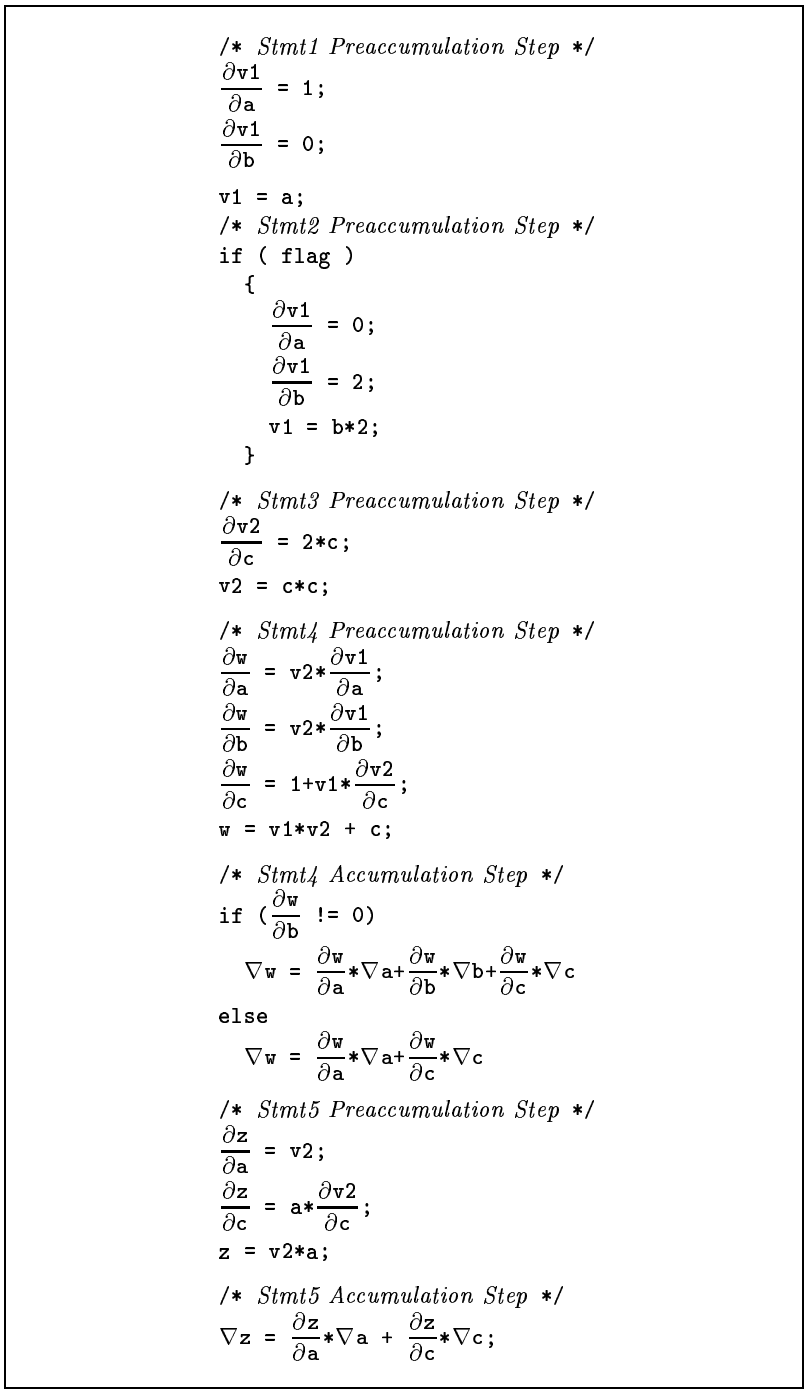


Figure 6.6: Derivative code obtained by augmenting the example code in Figure 6.5 via a hierarchical approach.

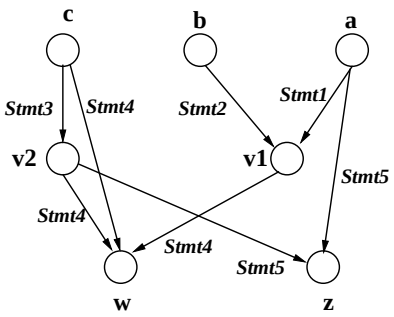


Figure 6.7: Data dependence graph of the variables for the example code in Figure 6.5. Following the paths from the input variables to each of the other variables we can figure out the derivative sparsity patterns.

The derivative code reported in Figure 6.6 is obtained by augmenting the code fragment in Figure 6.5 by using the approach outlined above.

The following two considerations have motivated the use of the *if-statement* related to the *Stmt4 Accumulation Step* (see Figure 6.6):

- if `flag` is true then the variable `v1` depends on `b`. This means that ∇`w` must be updated taking into account the contribution of ∇`b`.

- if `flag` is false then the variable `v1` does not depend on `b`. In this case, for updating ∇`w` it is necessary to not consider the contribution of ∇`b`.

The above considerations are necessary for the correctness of the derivative code. Let us assume that `b` is defined before the EBB only if `flag` is true. In this case, also ∇`b` is defined only if `flag` is true. Thus, we must update ∇`w` without taking into account ∇`b` in order to generate a correct derivative code when `flag` is false.

In order to distinguish between the two cases at run-time, we check whether $\frac{\partial \mathtt{w}}{\partial \mathtt{b}}$ is zero. If it is not zero, we are sure that `w` depends on the variable `b` and in this case `flag` is true. Instead, if it is zero, then either `w` does not depend on `b` (`flag` is false) or there has been numerical cancellations during the preaccumulation step; in any case, without taking into account the contribution of ∇`b`, we obtain a correct derivative code. Moreover, we can note that since `z` does not depend on the variable `b`, the value of $\frac{\partial \mathtt{z}}{\partial \mathtt{b}}$ is not checked since it is always zero. The last optimization has been possible due to the exploitation of the derivative sparsity patterns during the preaccumulation step.

6.3 Implementation Issues

In order to implement the strategy described in the preceding section, we have adopted a source-to-source transformation approach.

The main advantage of this approach is to provide great flexibility in implementing sophisticated algorithms, since the entire program context is available at compile time; thus, it is possible to exploit the program structure and the associativity of the chain rule.

The main drawback is that the development of robust source transformation tools requires a substantial effort. In order to make it easier to experiment with AD algorithmic techniques, the AIF [8], Automatic Differentiation Intermediate Form, was developed. AIF acts as a "glue layer" between a language-specific front-end and a language-independent transformation module that implements AD transformations at high level of abstractions. The AIF-based module, implementing the hierarchical approach, has been developed following schemes similar to ones reported in [1] and has been interfaced with the ADIC front-end.

6.4 Numerical Test

In our computational experiments, we have considered the *Flow in a Driven Cavity* function belonging to the MINPACK-2 test problem collection [2]. Since the code implementing the function is written in Fortran, it has been converted in C before augmenting it.

We have compared three methods for the computation of the Jacobian of the test function:

- the forward-reverse mode of ADIC (ADIC);
- the hierarchical approach without exploiting sparsity (HAD-FM);
- the hierarchical approach by exploiting sparsity (HAD-SFM).

The related results are reported in Figure 6.8.

The computational results show that methods based on a hierarchical approach are more effective than monolithic approaches, when they lead to an interface contraction. The HAD-FM method is less efficient than the ADIC approach when the number p of global derivatives is less than 10. Indeed, for $p < 10$, HAD-FM leads to an interface expansion.

On the other hand, the HAD-SFM method outperforms the ADIC approach even for $p = 1$, showing the benefits that could be achieved by using context-sensitive differentiation strategies.

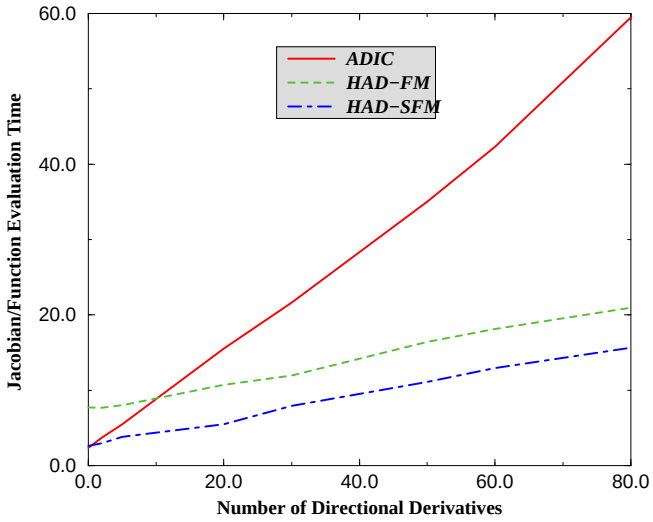


Figure 6.8: Results of application of hierarchical approaches to flow in a driven cavity problem.

The cross-country elimination problem in computational graphs

Uwe Naumann

naumann@math.tu-dresden.de

Institute of Scientific Computing
Technical University Dresden.

The chain rule can be applied to computational graphs representing a vector function $F : \mathbb{R}^n \supseteq D \to \mathbb{R}^m$ in any arbitrary order resulting in different operations counts for the calculation of the Jacobian matrix J. The minimization of the number of arithmetic operations, which are required for the computation of J, leads to a computationally hard combinatorial optimization problem. The Jacobian matrix can be expressed as a chained matrix product with factors representing the local extended Jacobians associated with each single intermediate variable. A dynamic programming algorithm can be used for the minimization of the cost of computing this product.

7.1 Objective

The calculation of derivatives of a vector function $F(x)$ by automatic differentiation is based on the application of the chain rule, which can be employed in various ways. Two special interpretations of this rule will lead to the well-known forward and reverse modes of automatic differentiation. In general, it serves as the basis for the so-called *cross-country elimination* approach, which motivates the research leading to the results desribed in this paper.

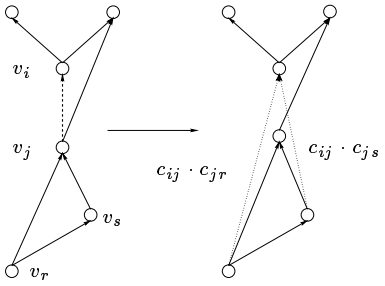


Figure 7.9: Edge (F)

Based on the graph representation of $F(x)$ we require a method of transforming the computational graph $CG(F)$ in a way that we get an equivalent representation of $J(x)$. In fact, this can be done by successively eliminating all vertices representing intermediate variables in the underlying evaluation program for $F(x)$, which is equivalent to eliminating all edges having either an intermediate vertex as source or having such a vertex as target or both, i.e. all intermediate edges. Thus, we get to a stage where the computational graph corresponding to this evaluation program represents a subgraph of the complete bipartite graph $K_{n,m}$ and the labels

$$c_{ji} \equiv \frac{\partial}{\partial x_i} y_j \qquad \text{for } i = 0, ..., n \quad \text{and } j = 0, ..., m$$

on the edges connecting the minimal vertices (representing the independent variables) with the maximal ones (dependent variables) are exactly the non-zero entries in $J(x)$.

7.2 Tools

Let the global extended Jacobian for a computational graph $CG(F)$ of a vector function F evaluated at the argument $\underline{x}_0 \in I\!R^n$ be defined as

$$I\!R^{(n+p+m)\times(n+p+m)} \ni \quad J_e(F) = (c)_{ij} \qquad \text{with} \quad c_{ij} = \begin{cases} \frac{\partial v_i}{\partial v_j}(\underline{x}_0) & j \in P_i \\ 0 & \text{otherwise} \end{cases}$$

where c_{ij} is the local partial derivative labeling the edge which connects vertices v_j with v_i in the computational graph. Here, $CG(F)$ contains p intermediate vertices for a given $F : I\!R^n \supseteq D \to I\!R^m$. P_i [S_i] denotes the set of indeces of vertices preceding [succeeding] v_i in $CG(F)$. Obviously, J_e is a square upper triangular matrix provided the numbering of the vertices in $CG(F)$ induces a topological ordering with respect to dependency.

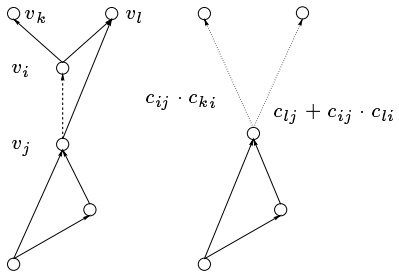


Figure 7.10: Edge (R)

The process of calculating the complete Jacobian may be regarded as a combined intermediate vertex-edge elimination procedure on $CG(F)$ with an analogous transformation process of $J_e(x)$. The elimination of an intermediate vertex v_i from the computational graph is performed by connecting each of its predecessors with each of its successors (provided they have not been connected before as we do not allow multiple edges) followed by updating the existing or generating the new elementary partial derivatives labelling the edges and, finally, the deletion of v_i. An edge (i, j) which connects two intermediate vertices v_i and v_j is forward [backward] eliminated by connecting all predecessors [successors] of v_i [v_j] with v_j [v_i]. Again, we update the local sensitivities

correspondingly and remove (i,j) from the graph. This may lead to the vanishing of v_i (v_j) if (i,j) was its only outedge [inedge].

As mentioned above, there is an equivalent for each of the three actions in the computational graph in terms of a transformation on $J_e(x)$.

- Forward elimination of an edge (i,j) (Figure 7.9)

$$\begin{aligned} J_e :&= J_e + e_j e_j^T J_e e_i e_i^T (J_e - I) \\ &= J_e + e_j e_j^T \bar{C}_i C_i - e_j e_j^T J_e e_i e_i^T \end{aligned}$$

- Reverse elimination of an edge (i,j) (Figure 7.10)

$$\begin{aligned} J_e :&= J_e + (J_e - I) e_j e_j^T J_e e_i e_i^T \\ &= J_e + \bar{C}_j C_j e_i e_i^T - e_j e_j^T J_e e_i e_i^T \end{aligned}$$

- Elimination of a vertex v_i (Figure 7.11)

$$\begin{aligned} J_e :&= (I - e_i e_i^T)(J_e + \bar{C}_i C_i)(I - e_i e_i^T) \\ &= (I - e_i e_i^T) J_e (I + e_i e_i^T J_e)(I - e_i e_i^T) \\ &= (I - e_i e_i^T)(I + J_e e_i e_i^T) J_e (I - e_i e_i^T) \end{aligned}$$

The extended local [adjoint] Jacobian C_i $[\bar{C}_i]$ contains the sensitivities of v_i with respect to each of its predecessors [the successors of v_i with respect to v_i itself]. I is the $(n+p+m)\times(n+p+m)$-identity matrix and e_i denotes the corresponding i-th cartesian basis vector.

7.3 Problem

So far, we have introduced our goal in the form of an equivalent for the Jacobian matrix and we have described three basic actions for getting to this stage. However, which problem arises from the above setup?

Let us denote the number of edges leading into [emanating from] a vertex v_j by $|P_j|$ $[|S_j|]$. Then the elimination of an intermediate vertex v_j involves exactly $|P_j|\cdot|S_j|$ multiplications each one possibly followed by an addition. Thus, the above product, which we will refer to as the **Markovitz degree**, is a characteristical value for every vertex describing the cost of eliminating it from the computational graph. Analogous, one observes that the forward [backward] elimination of an edge (i,j) takes $|P_i|$ $[|S_j|]$ multiplications, again, each

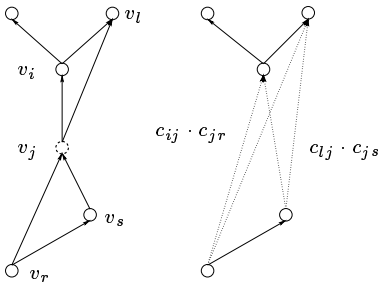


Figure 7.11: Vertex

of them possibly followed by an addition. This leads to the definition of the **forward [backward] Markovitz degree** of an edge. Notice, that the Markovitz degree is not a static value but changes with the ongoing elimination of vertices and edges.

Building on work of Rose and Tarjan [39] Herley showed in an unpublished paper that the computation of a vertex elimination sequence minimizing the number of newly generated edges (fill-in) in the computational graph of a vector function is an NP-hard combinatorial optimization problem. Although, there is no proof for it so far we expect the same property to hold for the closely related problem of minimizing the number of multiplications required to accumulate the complete Jacobian matrix of a general vector function $F(x)$.

The successive elimination of edges from the computational graph defines the so-called **metagraph** $M = M(CG) = (V_M, E_M)$ the vertices of which represent all different computational graphs that can be obtained by applying the general edge elimination strategy. If there are $p \gg \max\{m, n\}$ intermediate vertices in $CG(F)$ then an upper bound for the number of vertices in M (although not a very good one for practical cases) is given by $2^{\binom{p}{2}}$. Denoting the number of egdes in the transitive closure of CG by E^* we get $2^{|E^*|}$ as a tighter upper bound for the number of vertices in the metagraph M. The task is to solve a shortest path problem in the metagraph, which can be done using for example the algorithm by Bellman and Ford in a time that is proportional to $O(|V_M|, |E_M|)$. The difficulties arise from the fact that both the number of vertices and the number of edges in the metagraph depend on the number of intermediate vertices in the original computational graph exponentially. Thus, both an exhaustive search and any algorithm for computing a shortest path in the complete metagraph are not practicable. In order to decrease the complexity of the **general edge elimination problem** to solve one has to put certain restrictions on the metagraph, thus reducing both the number of its nodes and the number of different paths to check. This will lead to subgraphs which are then subject to an analogous shortest path problem.

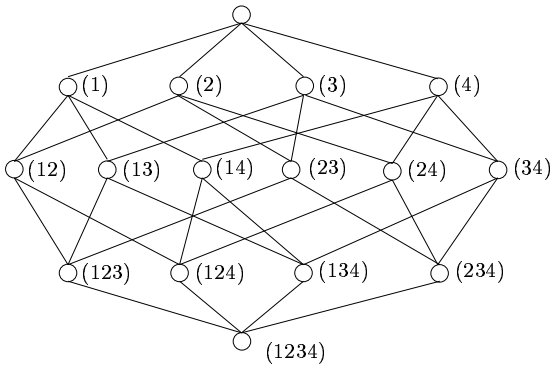


Figure 7.12: Vertex Elimination

An example is the restriction to the process of successively eliminating vertices from $CG(F)$. This approach has been exploited in several papers in order to minimize the overall operations count for computing Jacobians ([28] and [5]). Obviously, for a problem in p intermediate variables there are exactly $p!$ different elimination orderings. The vertex elimination strategy leads to a subgraph of the metagraph shown in Figure 7.12. The number of vertices in the vertex metagraph (which are exactly those vertices $w_i \in M$ that are reachable by a pure vertex elimination strategy) is equal to 2^p where p is the number of intermediate vertices in the original computational graph. The cost of computing the Jacobian using a certain vertex elimination ordering is equal to the sum over all Markovitz degrees of the

intermediate vertices at the stages $w_{i(j)} \in M$ of their elimination:

$$Cost\{J(x)\} = \sum_{1<j\leq p} mark_{i(j)}(v_j) \qquad \textbf{(overall Markovitz degree)} \tag{7.6}$$

The problem of minimizing this cost by determining a corresponding optimal vertex elimination ordering is called a **vertex elimination problem**. It is equivalent to the solution of a shortest path problem on the subgraph of the metagraph induced by the restriction to the elimination of vertices.

However, concentrating on the vertex elimination problem will not solve our general problem which was to determine a method for accumulating the Jacobian using a minimal number of multiplications. One can show that there are situations where the optimal vertex elimination sequence does in fact not minimize our objective function $Cost\{J(x)\}$. In order to be able to find this optimum one has to solve the general edge elimination problem including both the possibilities to eliminate edges forward and backward.

We conjecture that the minimal number of multiplications (from the best edge elimination ordering) differs from the number required by the optimal vertex elimination sequence by a small constant factor. Therefore, it certainly does make sense to consider the "simpler" problem of eliminating whole vertices.

7.4 Solution

Remember, that labelling the edges in the metagraph M with the cost (the number of multiplications) of getting from the graph represented by their source to the one associated with their target we end up with a shortest path problem on a graph in which the number of vertices depends on the number of intermediate vertices in the original computational graph exponentially. Thus, we have to think about certain restrictions, reducing the number of different paths to check. This will lead to subgraphs, which are then subject to an analogous shortest path problem while having a smaller size than the original metagraph.

Apart from the restriction to vertex elimination, which still results in an NP-hard problem there are numerous other ways to decrease the size of the metagraph. For example, with the terminology introduced in Section 7.2 we have a representation of the complete Jacobian J as a chained matrix product with factors representing the local extended Jacobians associated with each single intermediate variable given by

$$J = Q_m(C_{p+m} + I - e_{p+m}e_{p+m}^T) \cdots (C_1 + I - e_1 e_1^T)P_n^T$$

where P_n and Q_m are the matrices that project a given vector of length $n+p+m$ to its first n and its last m components, respectively. Using a dynamic programming algorithm one can determine a parenthesization which minimizes the number of multiplications required for the calculation of the above chained matrix product. Still, this does not deliver a solution to the general edge elimination problem. Due to the hardness of the optimization problem heuristics for finding the elimination ordering will play an important role in the search for a solution.

Bibliography

[1] J. Abate, C. Bischof, A. Carle, and L. Roh. Algorithms and design for a second-order automatic differentiation module. In *Int. Symposium on Symbolic and Algebraic Computing (ISSAC)*, pages 149–155. Association of Computing Machinery, New York, 1997.

[2] B. M. Averick, R. G. Carter, J. J. More, and G.-L. Xue. The Minpack-2 Test Problem Collection. Preprint MCS-P153-0692, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1992.

[3] M. Berz, C.H. Bischof, G.F. Corliss, and A. Griewank, editors. *Computational Differentiation: Applications, Techniques, and Tools.* SIAM, Philadelphia, 1996.

[4] C. Bischof. A collection of automatic differentiation tools. URL=`http://www.mcs.anl.gov/Projects/autodiff/AD_Tools/index.html`.

[5] C. H. Bischof. Hierarchical approaches to automatic differentiation. In [3], pages 83–94.

[6] C. H. Bischof, G. F. Corliss, L. Green, A. Griewank, K. Haigler, and P. Newman. Automatic differentiation of advanced CFD codes for multidisciplinary design. *Journal on Computing Systems in Engineering*, 3:625–638, 1992.

[7] C. H. Bischof and M. R. Haghighat. Hierarchical approaches to automatic differentiation. In [3] pages 83–94.

[8] C. H. Bischof, L. Roh, and A. Mauer. ADIC: An Extensible Automatic Differentiation Tool for ANSI-C. Preprint MCS-P626-1196, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1996.

[9] A. Carle, M. Fagan, and L. Green. Preliminary Results from the Application of Automatic Adjoint Code Generator to CFL3D. In *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, 1998.

[10] I. Charpentier. The mesodif package: Algorithmic documentation. http://www-lmc.imag.fr/~charpent/meteo.html.

[11] I. Charpentier. Génération de codes adjoints : Traitement de la trajectoire du modèle direct. Rapport de recherche 3405, INRIA, April 1998.

[12] I. Charpentier and M. Ghemires. Génération automatique de codes adjoints : Stratégies d'utilisation pour le logiciel Odyssée. Application au code météorologique Meso-NH. Rapport de recherche 3251, INRIA, September 1997.

[13] B. Creusillet. *Analyses de régions de tableaux et applications.* PhD thesis, École des Mines de Paris, 1996.

[14] B. Creusillet and F. Irigoin. Interprocedural Array Region Analysis. Rapport CRI A-282, École des Mines de Paris, January 1996.

[15] C. Eckert. *On Predictability Limits of ENSO - A Study Performed with a Simplified Model of the Tropical Pacific Ocean-Atmosphere System.* PhD thesis, Max-Planck-Institut für Meteorologie, Hamburg, Germany, 1998.

[16] ECMWF. *Use of parallel processors in meteorology: Making its mark*, 1996.

[17] F. Eyssette, J.-C. Gilbert, C. Faure, and N. Rostaing-Schmidt. Applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques dans un tube chauffant. Rapport de recherche 2795, INRIA, February 1996.

[18] C. Faure and Y. Papegay. Odyssée Version 1.6. The language reference manual. Rapport technique 0211, INRIA, November 1997.

[19] R. Giering. Tangent linear and Adjoint Model Compiler home page. URL=`http://puddle.mit.edu/~ralf/tamc`.

[20] R. Giering. *Tangent linear and Adjoint Model Compiler , Users manual*, 1997. Unpublished, available from `http://puddle.mit.edu/~ralf/tamc`.

[21] R. Giering and T. Kaminski. Recipes for Adjoint Code Construction. *ACM Trans. Math. Software*, (212), 1996. in press ACM Trans. On Math. Software.

[22] R. Giering and T. Kaminski. Recipes for Adjoint Code Construction. Technical Report, ACM TOM 212, Max-Planck-Institut für Meteorologie, 1998. in press.

[23] J.-C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21, 1992.

[24] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Pres, New York, 1981.

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

[26] A. Griewank. Some bounds on the complexity of gradients, Jacobians, and Hessians. In Panos M. Pardalos, editor, *Complexity in Nonlinear Optimization*, pages 128–161. World Scientific Publishers, 1993.

[27] A. Griewank and G.F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications*. SIAM, Philadelphia, 1991.

[28] A Griewank and S Reese. On the calculation of jacobian matrices by the markowitz rule. In [27], pages 126–135.

[29] J. E. Horwedel. GRESS: A preprocessor for sensitivity studies on Fortran programs. In [27], pages 243–250.

[30] P. Hovland, C. H. Bischof, D. Spiegelman, and M. Casella. Efficient derivative codes through automatic differentiation and interface contraction: an application in biostatistics. Preprint MCS-P491-0195, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1995.

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

[32] T. Kaminski, R. Giering, and M. Heimann. Sensitivity of the seasonal cycle of CO_2 at remote monitoring stations with respect to seasonal surface exchange fluxes determined with the adjoint of an atmospheric transport model. *Physics and Chemistry of the Earth*, 21(5–6):457–462, 1996.

[33] K. Kubota. PADRE2 - FORTRAN precompiler for automatic differentiation and estimates of rounding errors. In [3], pages 463–471.

[34] J. P. Lafore, J. Stein, N. Ascencio, P. Bougeault, V. Ducrocq, J. Duron, C. Fischer, P. Héreil, P. Mascart, V. Masson, J.-P. Pinty, J.-L. Redelsperger, E. Richard, and J. Vilà-Guerau de Arellano. The Meso-NH atmospheric simulation system. part I: adiabatic formulation and control system. *Ann. Geophysicae*, 16:90–109, 1998.

[35] F.-X. Le Dimet and O. Talagrand. Variational algorithms for analysis and assimilation of meteorological observations: Theoretical aspects. *Tellus*, 38A:97–110, 1986.

[36] J. Lewis and J. Derber. The use of the adjoint equations to solve a variational adjustment problem with advective constraints. *Tellus*, 37:309–327, 1985.

[37] Météo-France and CNRS. *The Meso-NH Atmospheric Simulation System: Scientific Documentation*, September 1995.

[38] M. O'Keefe and C. Kerr, editors. *Second international workshop on software engineering and code design in parallel meteorological and oceanographical applications*, 1998.

[39] D. J. Rose and R. E. Tarjan. Algorithmic Aspects of Vertex Elimination on Directed Graphs. *SIAM Journal of Applied Mathematics*, 34(1):176–197, January 1978.

[40] N. Rostaing, S. Dalmas, and A. Galligo. Automatic differentiation in Odyssée. *Tellus*, 45A(5):558–568, 1993.

[41] Jens Schröter. Driving of non-linear time dependent ocean models by observations of transient tracer - a problem of constrained optimization. In D.L.T. Anderson and J. Willebrand, editors, *Ocean Circulation Models: Combining Data and Dynamics*, pages 257–285. Kluwer Academic Publishers, 1989.

[42] J. G. Sela. Spectral modeling at the national meteorological center. *Monthly Weather Review*, 108(9):1279–1292, September 1980.

[43] D. Stammer, C. Wunsch, R. Giering, Q. Zhang, J. Marotzke, J. Marshall, and C. Hill. The Global Ocean Circulation estimated from TOPEX/POSEIDON Altimetry and a General Circulation Model. Technical Report 49, Center for Global Change Science, Massachusetts Institute of Technology, 1997.

[44] M. Tadjouddine, C. Faure, and F. Eyssette. Sparse jacobian computation in automatic differentiation by static program analysis. In G. Levi, editor, *Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 311–326. Springer-Verlag, September 1998.

[45] O. Talagrand. The use of adjoint equations in numerical modelling of the atmospheric circulation. In [27], pages 169–180.

[46] Y. Trémolet and J. Sela. Parallelization of the NCEP global spectral model. *Submitted to Journal of Parallel and Distributed Computing*, 1998.

[47] E. Tziperman and W. C. Thacker. An optimal control/adjoint equation approach to studying the ocean general circulation. *Journal of Physical Oceanography*, 19:1471–1485, 1989.

[48] G. J. van Oldenborgh, G. Burgers, C. Venzke, C. Eckert, and R. Giering. Tracking down the delayed ENSO oscillator with an adjoint OGCM. Technical Report 97-23, Royal Netherlands Meteorological Institute, P.O. Box 201, 3730 AE De Bilt, The Netherlands, 1997. Monthly Weather Review, in press.

[49] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley Publishing Company, 1991.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399