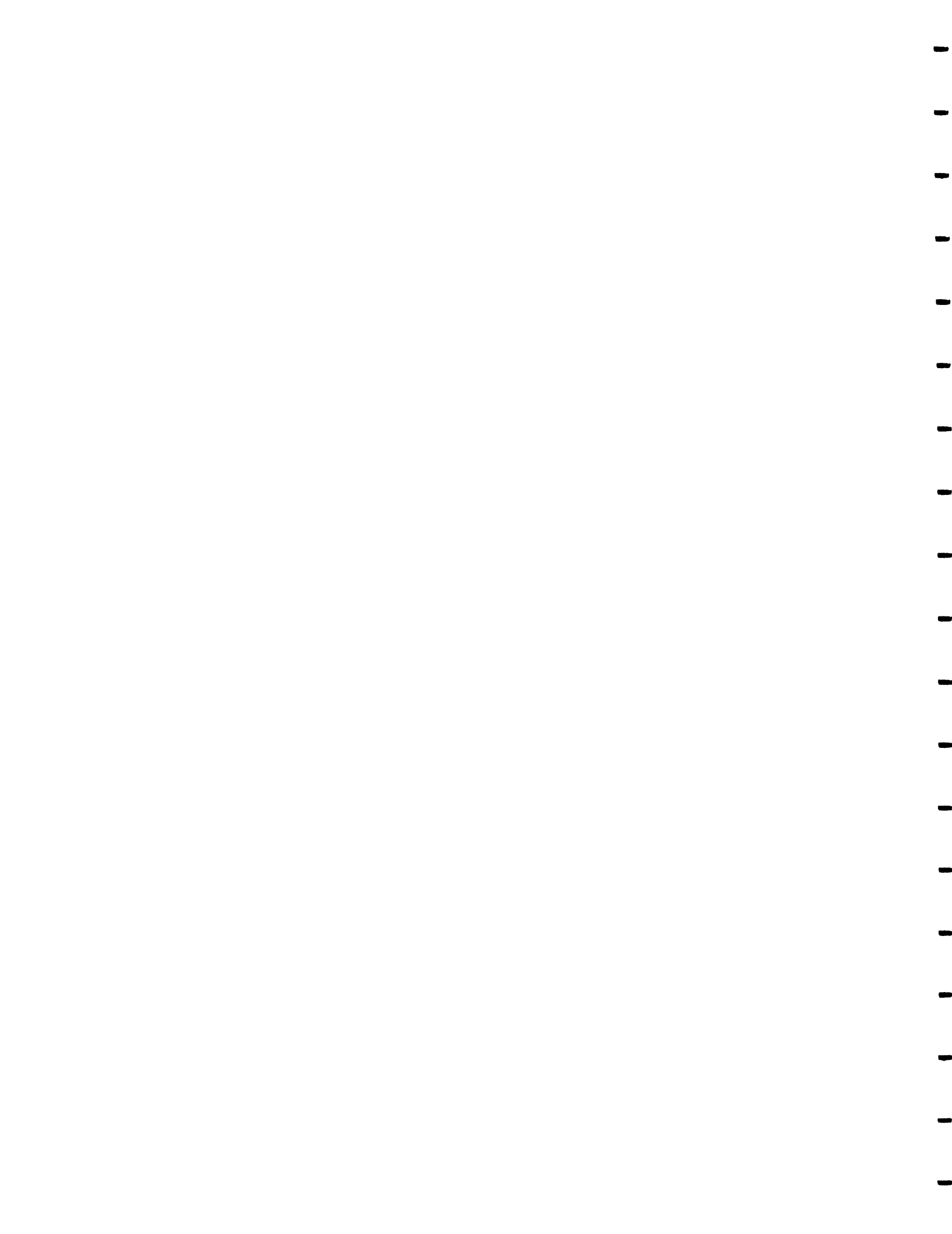


**Automatic Differentiation:
Overview and Application to Systems of
Parameterized Nonlinear Equations**

Marcela Rosemblun

**CRPC-TR92267
October 1992**

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892



RICE UNIVERSITY
**AUTOMATIC DIFFERENTIATION:
OVERVIEW AND APPLICATION TO
SYSTEMS OF PARAMETERIZED
NONLINEAR EQUATIONS**

by

Marcela Laura Rosemblun

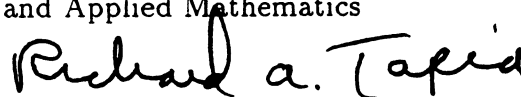
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Arts

APPROVED, THESIS COMMITTEE:



John E. Dennis, Jr., Chairman
Noah Harding Professor of Computational
and Applied Mathematics



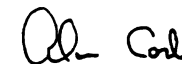
Richard A. Tapia
Noah Harding Professor of Computational
and Applied Mathematics



Ian D. Walker
Assistant Professor of Electrical and
Computer Engineering



Karen A. Williamson
Research Scientist



Alan Carle
Research Scientist

Houston, Texas
October, 1992

AUTOMATIC DIFFERENTIATION: OVERVIEW AND APPLICATION TO SYSTEMS OF PARAMETERIZED NONLINEAR EQUATIONS

Marcela Laura Rosembun

Abstract

Automatic Differentiation is a computational technique that allows the evaluation of derivatives of functions defined by computer programs. Derivatives are calculated by applying the *chain rule* of differential calculus to the sequence of elementary computations involved in the program. In this work, an overview of the theory and implementation of automatic differentiation is presented, as well as a description of the available software.

An application of automatic differentiation in the context of solving systems of parameterized nonlinear equations is discussed. In this application, the “differentiated” functions are implementations of Newton’s method and Broyden’s method. The iterates generated by the algorithms are differentiated with respect to the parameters. The results show that whenever the sequence of iterates converges to a solution of the system, the corresponding sequence of derivatives (computed by automatic differentiation) also converges to the correct value. Additionally, we show that the “differentiated” algorithms can be successfully employed in the solution of parameter identification problems via the Black-Box method.

Acknowledgments

I wish to express my deepest gratitude to professor John E. Dennis Jr. for his faith in me, and his generous support, guidance ... and patience! throughout my period of graduate study. I was indeed fortunate to have him as my advisor.

I owe a great deal of thanks to Karen A. Williamson and to Alan Carle for their guidance and valuable suggestions throughout this research , and for their careful readings of this thesis.

Very special thanks to professors Richard A. Tapia and Ian D. Walker for being part of my committee, and for taking the time to read this work.

I would also like to express my sincere gratitude to Dr. Andreas Griewank for his helpful suggestions throughout this research. Working with him at Argonne National Laboratory was a very valuable experience for me.

Thanks a lot to Mike Pearlman, Fran Mailian and Linda Neyra for being so helpful with all the graduate students of the Computational and Applied Mathematics Department.

Thanks to Siep Weiland, Wrenne Saunders, Cristina Maciel, Samir Kushalani, Piotr Krychniak, Rene Rodriguez and Klaus Holliger for all the experiences that we shared together, for their support and understanding ... and for their friendship!

Finally, my deepest appreciation goes to my parents Frida and David and to my sisters Corina and Carla, for their love. They were always with me, offering encouragement when I needed it the most.

To all my teachers.

Contents

Abstract	ii
Acknowledgments	iii
List of Illustrations	vii
List of Tables	viii
1 Introduction	1
2 Automatic Differentiation: an Overview	4
2.1 Preliminaries	4
2.2 Applying the Chain Rule to Computer Programs	9
2.3 Function Specification	11
2.4 The Forward Mode of Automatic Differentiation	15
2.5 The Reverse Mode of Automatic Differentiation	20
2.6 Graphical Interpretation	25
2.7 Computation of Jacobians	30
2.8 Implementations of Automatic Differentiation	35
2.9 Special Cases: Nondifferentiability and Branching	39
2.9.1 Nondifferentiability	39
2.9.2 Branching	40
2.10 Future Developments	41
3 Automatic Differentiation and Parameterized Fixed-Point Iterations	44

3.1	On the Solution of Systems of Nonlinear Equations	46
3.2	Differentiation of Parameterized Fixed-Point Iterations	52
4	Numerical Results	62
4.1	Differentiating LMDER and HYBRJ via ADIFOR	63
4.2	Application to the Solution of Parameter Identification Problems . . .	73
5	Concluding Remarks	86
A	Test Problems	88
	Bibliography	91

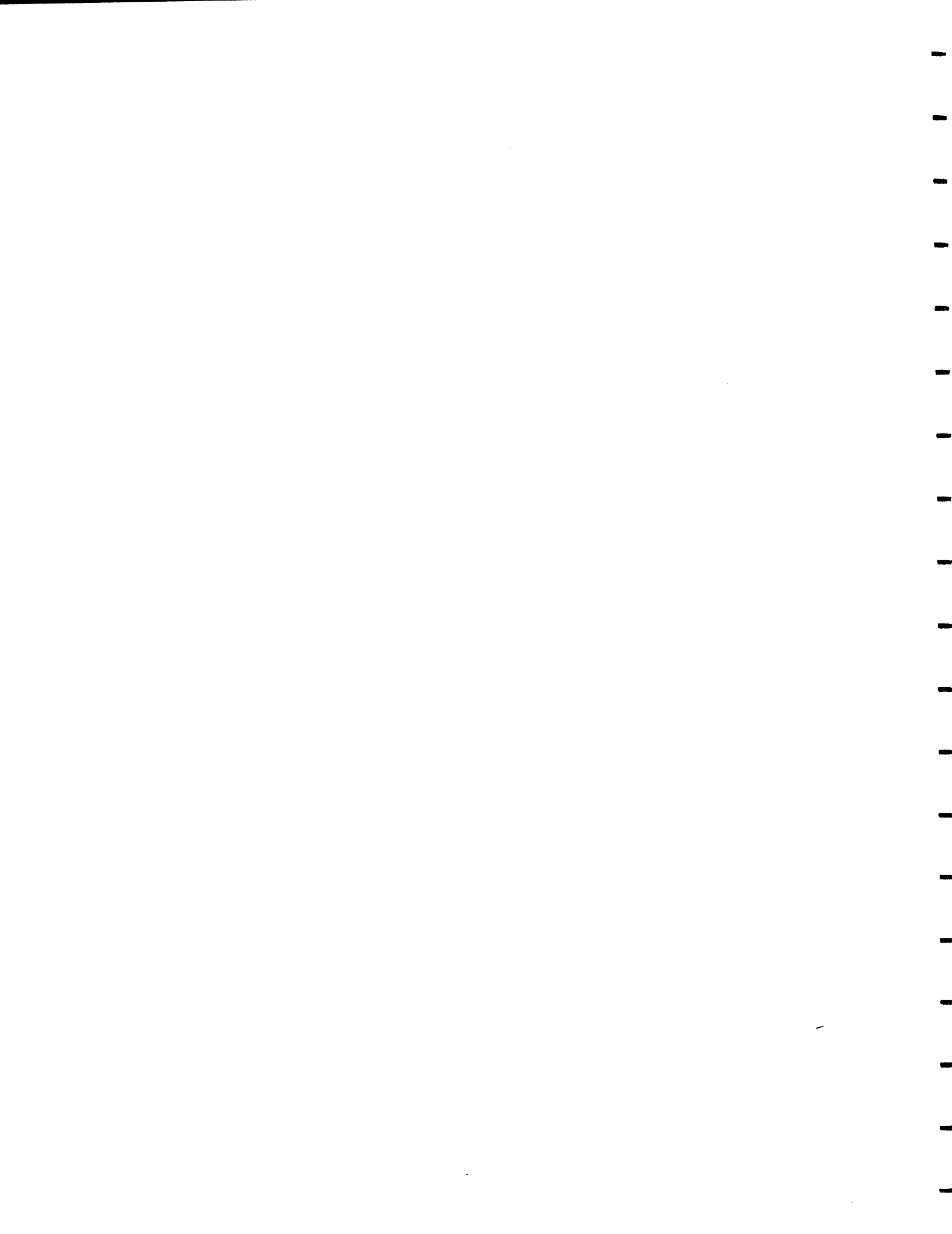
Illustrations

2.1	Elementary partial derivatives for $v_i \leftarrow \varphi_i(v_{j_1}, v_{j_2})$	27
2.2	Computational graph for $f(x) = 1.5 \cdot (x_1 \cdot x_2 + \exp(x_2)) - \cos(x_1)$	28
3.1	Execution of the algorithm \mathcal{A}	53
3.2	Execution of the “differentiated” algorithm \mathcal{A}'	54

Tables

4.1	<u>Problem 1</u>	$n_p = 2, n = 40, \bar{p} = (10^{-6}, 10^{-6})^T$	66
4.2	<u>Problem 2</u>	$n_p = 2, n = 80, \bar{p} = (3, 4)^T$	66
4.3	<u>Problem 3</u>	$n_p = 3, n = 80, \bar{p} = (2, 4, 4)^T$	67
4.4	<u>Problem 4</u>	$n_p = 3, n = 80, \bar{p} = (6, 4, 1)^T$	68
4.5	<u>Problem 5</u>	$n_p = 4, n = 80, \bar{p} = (10, 10, 30, 30)^T$	68
4.6	<u>Problem 6</u>	$n_p = 5, n = 200, \bar{p} =$ $(0.58 \times 10^{-4}, 0.26 \times 10^{-4}, 0.16 \times 10^{-4}, 0.28 \times 10^{-3}, 0.46 \times 10^{-4})^T$	69
4.7	<u>Problem 6 (ext.)</u>	$n_p = 10, n = 195, \bar{p} = (0.58 \times 10^{-4}, 0.26 \times$ $10^{-4}, 0.16 \times 10^{-4}, 0.28 \times 10^{-3}, 0.46 \times 10^{-4}, 0.10 \times 10^{-3}, 0.0, 0.0, 0.0, 0.0)^T$	69
4.8	<u>LMDER.ad - Final Results</u>		71
4.9	<u>HYBRJ.ad - Final Results</u>		71
4.10	<u>Problem 1</u>	$n_p = 2, n = 40, p_0 = (10^{-4}, 10^{-4})^T$	81
4.11	<u>Problem 1 (ext.)</u>	$n_p = 3, n = 39, p_0 = (10^{-4}, 10^{-4}, 0)^T$	81
4.12	<u>Problem 2</u>	$n_p = 2, n = 80, p_0 = (3, 4)^T$	82
4.13	<u>Problem 2 (ext.)</u>	$n_p = 4, n = 78, p_0 = (3, 4, 1, 0)^T$	82
4.14	<u>Problem 3</u>	$n_p = 3, n = 80, p_0 = (2, 4, 4)^T$	82
4.15	<u>Problem 3 (ext.)</u>	$n_p = 5, n = 78, p_0 = (2, 4, 4, 1, 0.3)^T$	82
4.16	<u>Problem 4</u>	$n_p = 3, n = 80, p_0 = (6, 4, 1)^T$	83
4.17	<u>Problem 4 (ext.)</u>	$n_p = 5, n = 78, p_0 = (6, 4, 1, 1, 0)^T$	83
4.18	<u>Problem 5</u>	$n_p = 4, n = 80, p_0 = (10, 10, 30, 30)^T$	83
4.19	<u>Problem 5 (ext.)</u>	$n_p = 6, n = 78, p_0 = (10, 10, 30, 30, 1, 0)^T$	83

4.20	<u>Problem 6</u>	$n_p = 5, n = 200, p_0 =$ $(0.58 \times 10^{-4}, 0.26 \times 10^{-4}, 0.16 \times 10^{-4}, 0.28 \times 10^{-3}, 0.46 \times 10^{-4})^T$	84
4.21	<u>Problem 6 (ext.)</u>	$n_p = 10, n = 195, p_0 = (0.58 \times 10^{-4}, 0.26 \times$ $10^{-4}, 0.16 \times 10^{-4}, 0.28 \times 10^{-3}, 0.46 \times 10^{-4}, 100, 0, 0, 0, 0)^T$	84
4.22	<u>Time ratios</u>		85
A.1	Data for Problem 1		88
A.2	Data for Problem 2		88
A.3	Data for Problem 3		89
A.4	Data for Problem 4		89
A.5	Data for Problem 5		90
A.6	Data for Problem 6		90



Chapter 1

Introduction

This work is intended to provide a comprehensive survey on the theory and implementation of the chain-rule based technique of automatic differentiation, and to present two applications: one in the context of solving systems of parameterized nonlinear equations, and the other, related to the first, in the context of solving parameter identification problems for ordinary differential equations via the so-called Black Box method.

Automatic differentiation is a computational technique that allows the calculation of derivatives of functions defined by computer programs. It can be a very convenient alternative compared to other approaches, such as *symbolic differentiation* or the approximation of derivatives by *finite differences*. It has been investigated for at least thirty years, and currently a lot of effort is being spent in the development of efficient tools. In Chapter 2, we will present the theoretical background underlying this technique, discuss different ways in which it can be implemented, and mention some of the available software.

Automatic differentiation has provided successful results in a variety of applications, including computer programs that describe very complicated models. However, one may wonder how feasible the computed derivatives are when the code that is “differentiated” involves an *iterative process*, i.e., a sequence of instructions that are executed several times, until a *stopping criterion* is satisfied.

In this work, we will analyze this situation in the context of the following application. We will consider the problem of solving the system of parameterized nonlinear

equations

$$f(x, p) = 0 \quad (1.1)$$

where $f : \mathbb{R}^{n+n_p} \rightarrow \mathbb{R}^n$ satisfies some smoothness assumptions, $p \in \mathbb{R}^{n_p}$ is a vector of parameters, and $x \in \mathbb{R}^n$ is the vector of unknowns, to be determined for a *fixed* p . Here \mathbb{R}^n denotes the n -dimensional Euclidean space.

Let us assume that for a given p , $x_\star \equiv x_\star(p)$ denotes a solution of (1.1), and that an iterative process of the form

$$x_{k+1} = \phi_k(x_k, p), \quad k = 0, 1, 2, \dots, \quad (1.2)$$

can be applied, where $\{\phi_k\}$ is a sequence of functions such that $\phi_k : \mathbb{R}^{n+n_p} \rightarrow \mathbb{R}^n$, for all k . As we will see later, the iterates generated by Newton's method and Broyden's method, for solving system (1.1), can be expressed in this form.

Assuming that the sequence of iterates $\{x_k(p)\}$ generated by (1.2) converges to x_\star , the idea is to apply automatic differentiation to a program that executes this iterative process, considering the components of the parameter vector p as the independent variables, and the components of each iterate $x_k(p)$ as the dependent variables.

Upon execution, and for a given value $p \equiv \bar{p}$, the resulting "differentiated" algorithm will generate two sequences: the sequence of iterates $\{x_k(\bar{p})\}$ and the corresponding sequence of derivatives $\{\partial x_k(\bar{p})/\partial p\}$.

The question that interests us is the following. *Does $\{\partial x_k(\bar{p})/\partial p\}$ converge to $\partial x_\star(\bar{p})/\partial p$ when $\{x_k(\bar{p})\}$ converges to $x_\star(\bar{p})$?*

Recently, J.C. Gilbert analyzed this issue in [30]. In this theoretical work, he identified a class of fixed-point iterations, that *includes Newton's method*, for which the above property holds. This class is quite restrictive, and it *does not include Broyden's method* or other secant methods. This theoretical work motivated our numerical experiments, which are discussed and presented in Chapters 3 and 4.

In Chapter 3, we introduce some background material about the iterative solution of systems of nonlinear equations, and discuss the application of automatic differentiation to algorithms that execute an iterative process of the form (1.2).

Chapter 4 is divided in two parts. In the first part, we show the results obtained by applying the automatic differentiation precompiler **ADIFOR** to two iterative solvers: **LMDER** and **HYBRJ**, from the **MINPACK-1** software library (see Morè et al. [55]). Basically, the first one is an implementation of Newton's method, and the second one is an implementation of Broyden's method. Both algorithms employ a trust-region approach as the globalization strategy (see Dennis and Schnabel [24]).

The resulting "differentiated" versions of **LMDER** and **HYBRJ** were tested on a set of 6 test problems, which are systems of parameterized nonlinear equations of the form (1.1). These systems originated in the discretization of the systems of parameterized first-order ordinary differential equations that are given in Appendix A. For all the test problems considered, the convergence property of the derivatives was satisfied. That is, whenever the sequence of iterates converged to a solution $x_*(\bar{p})$ of the system, the corresponding sequence of derivatives (computed by automatic differentiation) also converged to the correct value $\partial x_*(\bar{p})/\partial p$.

In the second part, we show that the "differentiated" versions of **LMDER** and **HYBRJ** can be successfully incorporated into a code that implements the Black-Box method for solving parameter identification problems. We tested the resulting parameter identification codes on the problems given in Appendix A, and compared their performance versus the performance of similar codes employing forward finite-difference approximations, instead of automatic differentiation. The results obtained indicate that using automatic differentiation leads to more robust but less efficient codes than using forward finite-difference approximations.

Finally, in Chapter 5 we present some concluding remarks.

Chapter 2

Automatic Differentiation: an Overview

2.1 Preliminaries

Numerical methods employed in the solution of many scientific problems often require the evaluation of the derivatives of some function

$$f(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix} \equiv \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \in \mathbb{R}^m,$$

where $f_1, \dots, f_m : \mathbb{R}^n \rightarrow \mathbb{R}$, and $x \equiv (x_1, \dots, x_n)^T \in \mathbb{R}^n$.

For differentiation purposes, the components x_1, \dots, x_n of x are usually called the *independent variables*, and the components y_1, \dots, y_m of $f(x)$ are called the *dependent variables*.

Assuming that f is well-defined and at least once-differentiable on some neighborhood of x , the derivative $f'(x)$ is given by the $m \times n$ matrix

$$J(x) = \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(x) & \cdots & \frac{\partial}{\partial x_n} f_1(x) \\ \vdots & & \vdots \\ \frac{\partial}{\partial x_1} f_m(x) & \cdots & \frac{\partial}{\partial x_n} f_m(x) \end{pmatrix},$$

called the *Jacobian*.

In the scalar case, $f(x) = y \in \mathbb{R}$, it is often necessary to compute the *gradient* vector, which is given by

$$\nabla f(x) = \begin{pmatrix} \frac{\partial}{\partial x_1} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{pmatrix} \in \mathbb{R}^n.$$

Often, the accuracy and the “computational cost” (measured in terms of speed and storage requirements) of the derivative calculation, have crucial influence in the robustness and efficiency of the numerical solution process.

In many applications, the function f to be differentiated is available either as an *algebraic expression* or in the form of a *computer program*, in some high-level programming language such as FORTRAN, Pascal, C, or another.

If f is given as an algebraic expression, then it is possible to differentiate it “*by hand*”, applying the well-known rules of differentiation. However, this is not a practical approach; it often becomes a tedious and error-prone task, even for moderate size problems.

Until recent years, automatic differentiation was not acknowledged as a “standard” computational technique for calculating derivatives, and many applied scientists considered that the practical alternatives were either *symbolic differentiation* or the approximation of derivatives by *finite differences*.

Symbolic differentiation is a capability provided by the so-called *computer algebra systems*, such as REDUCE [37], Macsyma [60], Maple [16], Mathematica [70], and others. These systems are large programming environments that solve or approximate the solution to different kinds of mathematical problems. They have become very popular tools due to their flexibility and user-friendliness.

Symbolic differentiation works as follows: it takes as input the algebraic expression (formula) representing the function, and it produces as output separate algebraic expressions corresponding to the analytic derivatives, all of them in terms of the independent variables. The “naive” symbolic differentiation algorithm is basically a straightforward application of the elementary differentiation rules on the given for-

mula, via symbolic manipulations. Mathematical expressions are represented by computational data structures such as linked lists or others (see Char [15], and Goldman et al. [31]). The generated derivative expressions can be evaluated at specific arguments, providing “exact” derivative values, up to machine precision. This means that the computed values would be exact if the computer arithmetic could be carried out with infinite precision.

The “naive” symbolic differentiation algorithm can be highly inefficient. Since the algebraic expressions for the derivatives are all expressed in terms of the independent variables, the same subexpression may occur in several places within the generated expressions. For large problems, this can result in a tremendous amount of formulae that can saturate the system. This “expression swell” problem, discussed by Griewank in [32], often limits the use of symbolic differentiation.

Most computer algebra systems offer the capability of *program generation*. In other words, they are able to translate the generated algebraic expressions for the derivatives into a computer program. But, due to the redundancies, the quality of the computational code generated from symbolic differentiation may be far from optimal. The evaluation of the partial derivatives $\partial f/\partial x_1, \dots, \partial f/\partial x_n$ can be about n times as expensive as the evaluation of f itself, or more (see Iri [43]). However, efficiency in the generated code can be considerably improved by the application of source-code optimizers which, among other tasks, try to simplify the code by removing redundant computations, as shown by Char in [15].

It is important to point out that the code generation and optimization facilities offered by symbolic differentiation may require a large amount of computing resources, even on very small problems (see Campbell [14]).

Another limitation in the application of symbolic differentiation is that it requires as input the algebraic expression for the function, which may not be available. Often,

the user can only provide the computational code that defines the function (for example, if the function is the result of a complex sequence of calculations, such as a simulation). Additionally, functions involving branches or loops cannot be handled by symbolic differentiation in a straightforward manner.

Traditionally, the alternative to computing analytic derivatives has been the use of **finite-difference approximations**.

For a scalar function $y = f(x) \in \mathbb{R}$, the partial derivative with respect to x_k , the k^{th} component of the vector $x \in \mathbb{R}^n$, can be approximated either by *one-sided differences*

$$\frac{\partial f(x)}{\partial x_k} \approx \frac{f(x \pm h \cdot e^{(k)}) - f(x)}{h}, \quad (2.1)$$

or by *central differences*

$$\frac{\partial f(x)}{\partial x_k} \approx \frac{f(x + h \cdot e^{(k)}) - f(x - h \cdot e^{(k)})}{2h}, \quad (2.2)$$

where $e^{(k)}$ denotes the k^{th} Cartesian basis vector in \mathbb{R}^n , and h is called the *finite-difference step*.

In order to compute these approximations, the user needs to provide a computer program that evaluates the function $f(x)$ at the required arguments, and possibly other parameters for the calculation of an appropriate step h .

The approximation error involved in (2.1) and (2.2) is of order h and h^2 , respectively (see Dennis and Schnabel [24], p. 77 – 80). Thus, for “sufficiently small” values of h , the central difference approximation is more accurate than the one-sided difference approximation, but it requires more function evaluations. Notice that in order to approximate the n partial derivatives $\partial f / \partial x_1, \dots, \partial f / \partial x_n$ it is necessary to perform $n + 1$ evaluations of $f(x)$ in (2.1), and $2n$ evaluations in (2.2).

For this reason, the computational cost of using finite-difference approximations can increase considerably for problems where the evaluation of the function is expensive and there is a large number of independent variables.

In the case of a vector function with n components, the one-sided difference approximation to the Jacobian also requires $n + 1$ function evaluations, and the central difference approximation requires $2n$ evaluations. However, if the Jacobian is sparse, it is possible to reduce the number of function evaluations. Curtis, Powell and Reid suggested in [20] an approach that is based on the partitioning of the columns of the Jacobian. Coleman and Morè, in [18] and [17], connected the partitioning problem to a graph coloring problem and gave some partitioning algorithms that can considerably reduce the number of required function evaluations.

An important point is the choice of the finite-difference step h . It may seem that the obvious thing to do is to choose h to be very small in order to make the approximations more accurate. However, if h becomes too small, then significant digits can be lost due to cancellation errors. If the sacrifice of significant digits occurs often enough in the program that employs the finite-difference approximations, then the final results can become meaningless. This situation creates a serious problem for the user: *to find the optimal value for h* in order to maximize the accuracy attained in the derivative approximation and to minimize the loss of significant digits. A careful analysis on how to choose the finite-difference step h can be found in the book by Dennis and Schnabel [24] (p. 96 – 99). If the steps are properly selected, then methods using finite-difference approximations to the derivatives can give similar performance to the same methods using analytic derivatives.

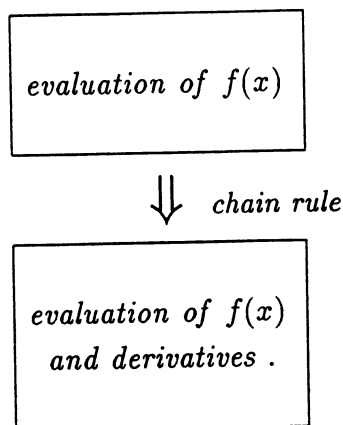
In some cases, finite-difference approximations may become too inaccurate to be useful. For the same reason, they may be unreliable for the estimation of second and higher-order derivatives.

2.2 Applying the Chain Rule to Computer Programs

Automatic differentiation often tends to be confused with symbolic differentiation. Both are computational techniques that implement the chain rule in a mechanical fashion to compute analytic, or exact, derivatives. As we mentioned earlier, this means that the computed derivative values would be exact if infinite precision arithmetic were employed. In both techniques, there is no need to select parameters that may affect the quality of the results, like in finite-difference approximations. But, while symbolic differentiation is concerned with the differentiation of functions defined by algebraic expressions, automatic differentiation considers functions defined by *computational codes*. Automatic differentiation applies the chain rule to a sequence of computational steps, rather than to formulae.

The idea of “differentiating” a computer program is attractive because most functions of practical interest can be defined, or approximated, by programs written in some high-level computer language.

Automatic differentiation can be viewed as a computational process that requires as input the sequence of computations for evaluating a function, and produces as output an extended sequence of computations for evaluating both the function and its partial derivatives. The transformation process consists of the *systematic application of the chain rule* to the sequence of elementary operations involved in the original program. Graphically, this process may be viewed as follows:



There are two basic ways in which the chain rule can be applied to the computational code that evaluates a function: the so-called *forward* and *reverse* modes of automatic differentiation.

The forward mode has been investigated for at least thirty years. The first publications in the area were the works by Beda et al. [2] and by Wengert [67].

The reverse mode has been studied at least since the early seventies, and was first published by Ostrovskii et al. [59]. This approach is closely related to the adjoint sensitivity analysis for differential equations, which has been used particularly in nuclear engineering (Cacuci [12], [13]), weather forecasting (Navon et al. [57]), and neural networks (Werbos [68]).

Numerous other references about the forward and the reverse mode can be found in the papers of Kedem [51], Rall [62], Kagiwada et al. [48], Fischer [27], and Griewank [32].

Both modes allow the computation of derivatives of functions defined by computer programs, but depending on the characteristics of the problem, one may be more efficient than the other in terms of time and storage requirements.

In the remainder of this chapter, we will present a survey of the theory and implementation of automatic differentiation. In Section 2.3 we introduce some notation

that is used to describe the evaluation of a function defined by a computer program. In Sections 2.4 and 2.5 we discuss the forward and the reverse modes of automatic differentiation. In Section 2.6 we show a different way of representing the process of evaluating a function in terms of the so-called computational graph. This representation is useful in the context of automatic differentiation. In Section 2.7 we discuss some approaches to compute Jacobian matrices by automatic differentiation. In Section 2.8 we comment about different implementations of automatic differentiation and mention some of the available software. In Section 2.9 we discuss some difficulties that may arise in “naive” application of automatic differentiation, and that may lead to incorrect results. Finally, in Section 2.10 we mention some of the current topics of research in the field.

Even though we will consider the application of automatic differentiation to compute first-order derivatives, this can be generalized to the computation of multivariate higher-order derivatives.

2.3 Function Specification

In this section, we introduce some notation that allows us to represent the evaluation process of a function defined by a computer program. We will focus on the scalar case. Vector functions will be discussed in Section 2.7.

Let us assume that a function $y = f(x) \in \mathbb{R}$ is defined by an *evaluation program* in some high-level computer language. For a given $x \equiv (v_1, \dots, v_n)^T \in \mathbb{R}^n$, the execution of this evaluation program in a computer can be viewed as the composition of a finite sequence of *unary* and *binary elementary operations*, such as $+$, $-$, \times , \div , \exp , \log , \sin , \cos , etc.

Let us denote each elementary operation executed in the evaluation of $f(x)$, by φ_i , with $i > n$, and let us assume that the result of evaluating each φ_i at the current

arguments is stored in an *intermediate variable* v_i . We assume that the arguments of each elementary operation are already computed quantities.

As an example, let us consider the function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ given by

$$y = f(x_1, x_2) = 1.5 \cdot (x_1 \cdot x_2 + \exp(x_2)) - \cos(x_1). \quad (2.3)$$

The corresponding gradient vector is

$$\nabla f(x) = \begin{pmatrix} 1.5 \cdot x_2 + \sin(x_1) \\ 1.5 \cdot (x_1 + \exp(x_2)) \end{pmatrix}.$$

For a given $x \equiv (v_1, v_2)^T \in \mathbb{R}^2$, the evaluation of $y = f(x)$ can be performed by executing the following sequence of elementary operations:

$$\begin{aligned} v_3 &\leftarrow \varphi_3(v_1, v_2) \equiv v_1 \cdot v_2 \\ v_4 &\leftarrow \varphi_4(v_2) \equiv \exp(v_2) \\ v_5 &\leftarrow \varphi_5(v_3, v_4) \equiv v_3 + v_4 \\ v_6 &\leftarrow \varphi_6(v_5) \equiv 1.5 \cdot v_5 \\ v_7 &\leftarrow \varphi_7(v_1) \equiv \cos(v_1) \\ y = v_8 &\leftarrow \varphi_8(v_6, v_7) \equiv v_6 - v_7. \end{aligned} \quad (2.4)$$

Obviously, there may be many different ways of evaluating $f(x)$ as a sequence of elementary operations. For example, $f(x)$ could also be evaluated by executing the sequence

$$\begin{aligned} v_3 &\leftarrow \varphi_3(v_1) \equiv \cos(v_1) \\ v_4 &\leftarrow \varphi_4(v_1, v_2) \equiv v_1 \cdot v_2 \\ v_5 &\leftarrow \varphi_5(v_2) \equiv \exp(v_2) \\ v_6 &\leftarrow \varphi_6(v_4, v_5) \equiv v_4 + v_5 \\ v_7 &\leftarrow \varphi_7(v_6) \equiv 1.5 \cdot v_6 \\ y = v_8 &\leftarrow \varphi_8(v_7, v_3) \equiv v_7 - v_3. \end{aligned} \quad (2.5)$$

In more general terms, for a given $x \equiv (v_1, \dots, v_n)^T$, we can denote the sequence of elementary operations executed to evaluate $f(x)$ by $\varphi_{n+1}, \dots, \varphi_r$, for some $r > n$.

As shown in the examples (2.4) and (2.5), the evaluation of each φ_i ($n+1 \leq i \leq r$) can be represented by an assignment of the form

$$v_i \leftarrow \varphi_i(v_j)_{j \in \mathcal{I}_i}, \quad (2.6)$$

where the index set \mathcal{I}_i , associated to v_i , contains the indices j corresponding to the arguments v_j on which v_i depends *directly*. These arguments are either one or two variables, depending on whether the φ_i is unary or binary. As we mentioned earlier, the arguments of φ_i must be already computed quantities, that is $\{v_j : j \in \mathcal{I}_i\} \subset \{v_1, \dots, v_{i-1}\}$.

Using an informal programming language, we can specify the evaluation process of $y = f(x)$ as follows

Function Evaluation

Given $v_1 \equiv x_1, \dots, v_n \equiv x_n$,

for $i = n + 1, \dots, r$

$v_i \leftarrow \varphi_i(v_j)_{j \in \mathcal{I}_i}$,

$y = v_r$.

(2.7)

In practice, some other parameters or constants may be provided as input in (2.7), but we do not need to take these quantities into consideration for our study.

The above notation can be applied to programs involving any number of intermediate variables. Usually, this number is much larger than the number of dependent and independent variables.

As mentioned earlier, each φ_i ($n+1 \leq i \leq r$) is unary or binary, and is typically of the form

$$\varphi_i(v_{j_1}, v_{j_2}) \quad \text{if} \quad \varphi_i \equiv +, -, \times, \div \quad (2.8)$$

or

$$\varphi_i(v_{j_1}) \quad \text{if} \quad \varphi_i \equiv \exp, \log, \sin, \cos, \text{ etc.} \quad (2.9)$$

The sequence $\varphi_{n+1}, \dots, \varphi_r$ characterizes the computation of the function. In the literature, this sequence has been assigned different names, such as: *basic representation* (Kedem [51]), *code list* (Rall [62]), *computational scheme* (Iri [42]), *computational process* (Iri et al. [44]), *characterizing sequence* (Fischer [29]), etc.

The list of elementary operations could also be extended to include any other function required by the user, provided that the corresponding partial derivatives are supplied as well.

We will assume that for each elementary operation

$$v_i \leftarrow \varphi_i(v_j)_{j \in \mathcal{I}_i}, \quad n+1 \leq i \leq r, \quad (2.10)$$

the corresponding *elementary partial derivatives*

$$\frac{\partial \varphi_i(v_j)_{j \in \mathcal{I}_i}}{\partial v_k}, \quad \text{for all } k \in \mathcal{I}_i \quad (2.11)$$

are well-defined and easily computable at all arguments of interest. This is clearly the case when φ_i is one of the operations included in the lists (2.8) or (2.9).

For example, in the third line of (2.4), from the assignment:

$$v_5 \leftarrow \varphi_5(v_3, v_4) \equiv v_3 + v_4,$$

it follows that

$$\frac{\partial \varphi_5(v_3, v_4)}{\partial v_3} = 1 \quad \text{and} \quad \frac{\partial \varphi_5(v_3, v_4)}{\partial v_4} = 1,$$

or from the fifth line:

$$v_7 \leftarrow \varphi_7(v_1) \equiv \cos(v_1),$$

we can compute

$$\frac{\partial \varphi_7(v_1)}{\partial v_1} = -\sin(v_1).$$

This is the context in which we will study the application of automatic differentiation. Basically, this technique consists in applying the *chain rule* in an appropriate way to propagate the elementary derivatives (i.e., the derivatives corresponding to the elementary operations) through the entire sequence of calculations that characterize the function. At the end of the process, the desired derivative values can be obtained. As we will see in the next sections, this can be implemented in a completely mechanical fashion.

2.4 The Forward Mode of Automatic Differentiation

The forward mode of automatic differentiation is basically a straightforward application of the chain rule to the sequence of elementary operations that characterizes the function. The idea is to *extend* in the original code (2.7), the evaluation of each intermediate variable v_i ($n+1 \leq i \leq r$), with the computation of an n -vector ∇v_i , defined by

$$\nabla v_i \equiv \begin{pmatrix} \partial v_i / \partial v_1 \\ \vdots \\ \partial v_i / \partial v_n \end{pmatrix} \in \mathbb{R}^n, \quad (2.12)$$

where v_1, \dots, v_n are the independent variables.

By applying evaluation rules together with elementary differentiation rules the pairs $(v_i, \nabla v_i)$ can be propagated throughout the whole sequence of elementary operations executed in (2.7). At the end, the last computed intermediate variable v_r will

give the value of $y = f(x)$, and its corresponding derivative vector ∇v_r will give the value of $\nabla f(x)$. In this way, the forward mode computes the derivatives of *all the intermediate quantities* involved in the calculation with respect to the *independent variables*

Each vector ∇v_i ($n+1 \leq i \leq r$) is evaluated by applying the chain rule to the corresponding elementary operation

$$v_i \leftarrow \varphi_i(v_j)_{j \in \mathcal{I}_i},$$

which gives

$$\nabla v_i = \sum_{k \in \mathcal{I}_i} \frac{\partial \varphi_i(v_j)_{j \in \mathcal{I}_i}}{\partial v_k} \cdot \nabla v_k. \quad (2.13)$$

The evaluation of ∇v_i can be easily performed because of the following facts:

- The φ_i 's are just unary or binary operations (i.e., each set \mathcal{I}_i has either one or two elements), and the partial derivatives $\partial \varphi_i / \partial v_k$, with $k \in \mathcal{I}_i$, that appear in the right-hand side of (2.13), are easy to compute.
- The vectors ∇v_k , with $k \in \mathcal{I}_i$, that also appear in the right-hand side of (2.13), can be calculated by the same formula (2.13) in a previous step of the process (for example, when the corresponding v_k 's are evaluated). Notice that $k \in \mathcal{I}_i$ implies $k < i$, since v_i depends on previously computed quantities.

Thus, by applying the chain rule as in (2.13) to each elementary assignment in (2.7), the following extended program can be generated:

Forward Mode

Given $v_1 \equiv x_1, \dots, v_n \equiv x_n$,

Initialization

for $i = 1, \dots, n$

$$\nabla v_i \leftarrow e^{(i)}.$$

Function evaluation and forward accumulation

for $i = n + 1, \dots, r$

$$v_i \leftarrow \varphi_i(v_j)_{j \in \mathcal{I}_i},$$

$$\nabla v_i \leftarrow \sum_{k \in \mathcal{I}_i} \frac{\partial \varphi_i(v_j)_{j \in \mathcal{I}_i}}{\partial v_k} \cdot \nabla v_k,$$

$$y = v_r,$$

$$\nabla y = \nabla v_r.$$

(2.14)

This extended program evaluates both $f(x)$ and $\nabla f(x)$ *simultaneously*. In (2.14), y denotes the computed value for $f(x)$ and ∇y denotes the computed value for $\nabla f(x)$.

The initialization of the gradients $\nabla v_i = e^{(i)}$ ($i = 1, \dots, n$), corresponding to the independent variables, arises from (2.12). Notice that $e^{(i)}$ represents the i^{th} Cartesian basis vector in \mathbb{R}^n .

Let us illustrate how the forward mode of automatic differentiation would proceed on the sequence of computations given by (2.4).

Initialization:

$$\nabla v_1 \leftarrow e^{(1)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\nabla v_2 \leftarrow e^{(2)} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Function evaluation and forward accumulation:

$$v_3 \leftarrow v_1 \cdot v_2$$

$$\nabla v_3 \leftarrow v_2 \cdot \nabla v_1 + v_1 \cdot \nabla v_2 = \begin{pmatrix} v_2 \\ v_1 \end{pmatrix}$$

$$v_4 \leftarrow \exp(v_2)$$

$$\nabla v_4 \leftarrow \exp(v_2) \cdot \nabla v_2 = \begin{pmatrix} 0 \\ \exp(v_2) \end{pmatrix}$$

$$v_5 \leftarrow v_3 + v_4$$

$$\nabla v_5 \leftarrow \nabla v_3 + \nabla v_4 = \begin{pmatrix} v_2 \\ v_1 + \exp(v_2) \end{pmatrix}$$

$$v_6 \leftarrow 1.5 \cdot v_5$$

$$\nabla v_6 \leftarrow 1.5 \cdot \nabla v_5 = \begin{pmatrix} 1.5 \cdot v_2 \\ 1.5 \cdot (v_1 + \exp(v_2)) \end{pmatrix}$$

$$v_7 \leftarrow \cos(v_1)$$

$$\nabla v_7 \leftarrow -\sin(v_1) \cdot \nabla v_1 = \begin{pmatrix} -\sin(v_1) \\ 0 \end{pmatrix}$$

$$y = v_8 \leftarrow v_6 - v_7$$

$$\nabla y = \nabla v_8 \leftarrow \nabla v_6 - \nabla v_7 = \begin{pmatrix} 1.5 \cdot v_2 + \sin(v_1) \\ 1.5 \cdot (v_1 + \exp(v_2)) \end{pmatrix}$$

Since the evaluation of each intermediate quantity v_i is accompanied by the evaluation of the n components $\partial v_i / \partial v_1, \dots, \partial v_i / \partial v_n$ of the corresponding gradient vector ∇v_i , the time complexity is proportional to the total number of independent

variables n . Therefore, the forward mode can be about as costly as approximating $\nabla f(x)$ by forward finite-differences.

The space complexity is also proportional to n , because of the dimension of the vectors ∇v_i . It may be possible to save space by using sparse storage techniques, since the ∇v_i 's often have many zero components. However, a sparse implementation may add some overhead.

In order to save storage, it is possible to implement the forward mode in another fashion. The idea is to consider just a *subset* of the independent variables at a time, and apply the forward mode to obtain the partial derivatives of $f(x)$ with respect to those variables.

For example, let us assume that just *one* independent variable is chosen for differentiation, say v_l ($1 \leq l \leq n$). The forward mode proceeds as in (2.14), but instead of associating with each intermediate quantity v_i an n -vector ∇v_i as before, just a scalar \hat{v}_i is computed, defined by

$$\hat{v}_i \equiv \frac{\partial v_i}{\partial v_l},$$

which is in fact the l^{th} component of the gradient ∇v_i . As before, the pairs (v_i, \hat{v}_i) can be propagated throughout the entire sequence of computations. At the end of the process, the last computed scalar \hat{v}_r will give the value of $\partial f(x)/\partial v_l$, the l^{th} component of $\nabla f(x)$. This process can be repeated n times, once for each independent variable, as described in the following algorithm.

Component-by-component forward mode

Given $v_1 \equiv x_1, \dots, v_n \equiv x_n$.

For each v_l ($1 \leq l \leq n$):

Initialization

$$\hat{v}_l \equiv \frac{\partial v_l}{\partial v_l} \leftarrow 1,$$

$$\hat{v}_i \equiv \frac{\partial v_i}{\partial v_l} \leftarrow 0, \quad i = 1, \dots, n; i \neq l,$$

(2.15)

Function evaluation and forward accumulation

for $i = n + 1, \dots, r$

$$v_i \leftarrow \varphi_i(v_j)_{j \in \mathcal{I}_i},$$

$$\hat{v}_i \leftarrow \sum_{k \in \mathcal{I}_i} \frac{\partial \varphi_i(v_j)_{j \in \mathcal{I}_i}}{\partial v_k} \cdot \frac{\partial v_k}{\partial v_l},$$

$$y = v_r,$$

$$\partial y / \partial v_l = \hat{v}_r.$$

This implementation of the forward mode is considerably less economical than (2.14) in terms of computational effort, but requires only about *twice* as much storage as the original evaluation program.

2.5 The Reverse Mode of Automatic Differentiation

The reverse mode of automatic differentiation associates with each intermediate variable v_i a *scalar* quantity \bar{v}_i , called the *adjoint*. The calculation of adjoints proceeds in *reverse order* with respect to the order in which the intermediate quantities are evaluated in the original program (2.7). This is in remarkable contrast with the for-

ward mode, where each v_i is computed *together* with the associated n -vector ∇v_i , as shown in the previous section.

For each intermediate quantity v_i ($n+1 \leq i \leq r$), the corresponding adjoint is defined by

$$\bar{v}_i \equiv \frac{\partial y}{\partial v_i} = \frac{\partial v_r}{\partial v_i}, \quad (2.16)$$

and it measures the *sensitivity* of the final result v_r with respect to the intermediate quantity v_i .

The process for calculating the adjoints requires going through the entire sequence of elementary computations in *reverse* order. Thus, it can be executed only *after* the function has been evaluated.

This can be best understood with an example. Let us consider the function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ defined by

$$y = f(x_1, x_2) = x_1 \cdot x_2 + \exp(x_2).$$

Let us assume that, for a given $x = (v_1, v_2)^T \in \mathbb{R}^2$, $f(x)$ is evaluated by the following sequence of assignments

$$\begin{aligned} v_3 &\leftarrow v_1 \cdot v_2 \\ v_4 &\leftarrow \exp(v_2) \\ y = v_5 &\leftarrow v_3 + v_4, \end{aligned}$$

which coincide with the first three elementary operations in (2.4).

Considering the final result $y = v_5$ as the dependent variable, we can compute its partial derivatives with respect to *all the quantities* involved in the calculation by applying the chain rule in a “backwards” fashion, as follows

$$\begin{aligned} \frac{\partial v_5}{\partial v_5} &= 1 \\ \frac{\partial v_5}{\partial v_4} &= \frac{\partial v_5}{\partial v_5} \cdot \frac{\partial v_5}{\partial v_4} \\ \frac{\partial v_5}{\partial v_3} &= \frac{\partial v_5}{\partial v_5} \cdot \frac{\partial v_5}{\partial v_3} \end{aligned}$$

$$\begin{aligned}\frac{\partial v_5}{\partial v_2} &= \frac{\partial v_5}{\partial v_3} \cdot \frac{\partial v_3}{\partial v_2} + \frac{\partial v_5}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_2} \\ \frac{\partial v_5}{\partial v_1} &= \frac{\partial v_5}{\partial v_3} \cdot \frac{\partial v_3}{\partial v_1} + \frac{\partial v_5}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_1} .\end{aligned}$$

Denoting

$$\bar{v}_5 \equiv \frac{\partial v_5}{\partial v_5}, \quad \bar{v}_4 \equiv \frac{\partial v_5}{\partial v_4}, \quad \bar{v}_3 \equiv \frac{\partial v_5}{\partial v_3}, \quad \bar{v}_2 \equiv \frac{\partial v_5}{\partial v_2}, \quad \bar{v}_1 \equiv \frac{\partial v_5}{\partial v_1},$$

we can rewrite the previous sequence of computations in following way

$$\begin{aligned}\bar{v}_5 &= 1 \\ \bar{v}_4 &= \bar{v}_5 \cdot \frac{\partial v_5}{\partial v_4} = 1 \cdot 1 = 1 \\ \bar{v}_3 &= \bar{v}_5 \cdot \frac{\partial v_5}{\partial v_3} = 1 \cdot 1 = 1 \\ \bar{v}_2 &= \bar{v}_3 \cdot \frac{\partial v_3}{\partial v_2} + \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_2} = 1 \cdot v_1 + 1 \cdot \exp(v_2) = v_1 + \exp(v_2) \\ \bar{v}_1 &= \bar{v}_3 \cdot \frac{\partial v_3}{\partial v_1} + \bar{v}_4 \cdot \frac{\partial v_4}{\partial v_1} = 1 \cdot v_2 + 1 \cdot 0 = v_2 .\end{aligned}$$

Since $v_1 \equiv x_1$ and $v_2 \equiv x_2$, the above sequence gives

$$\begin{aligned}\bar{v}_1 &= x_2 \\ \bar{v}_2 &= x_1 + \exp(x_2),\end{aligned}$$

which are the components of $\nabla f(x)$.

Now, we will describe the reverse mode in more general terms. After the function $f(x)$ is evaluated, the algorithm starts by initializing the adjoints as follows

$$\begin{aligned}\bar{v}_i &= 0 & i = 1, \dots, r-1, \\ \bar{v}_r &= \frac{\partial v_r}{\partial v_r} \equiv 1 .\end{aligned}$$

Then, by going in a backwards fashion through the entire sequence of computations performed in (2.7), the reverse mode considers each (already executed) elementary

operation

$$v_i \leftarrow \varphi_i(v_j)_{j \in \mathcal{I}_i}, \quad i = r, \dots, n+1,$$

and computes

$$\bar{v}_k \leftarrow \bar{v}_k + \frac{\partial \varphi_i(v_j)_{j \in \mathcal{I}_i}}{\partial v_k} \cdot \bar{v}_i \quad \text{for each } k \in \mathcal{I}_i. \quad (2.17)$$

In other words, all the adjoints \bar{v}_k such that the corresponding v_k is an argument of φ_i , are incremented by the quantity

$$\frac{\partial \varphi_i(v_j)_{j \in \mathcal{I}_i}}{\partial v_k} \cdot \bar{v}_i. \quad (2.18)$$

Notice that in order to evaluate (2.18), all the arguments v_j , with $j \in \mathcal{I}_i$, of the elementary function φ_i , must be already computed.

The intermediate variable v_i being processed must receive all the contributions to its adjoint \bar{v}_i , before it can start contributing to the adjoints of the v_k 's, with $k < i$. In this way, all the adjoints $\bar{v}_r, \dots, \bar{v}_{n+1}, \bar{v}_n, \dots, \bar{v}_1$ are computed.

At the end of the process, the adjoints $\bar{v}_1, \dots, \bar{v}_n$ corresponding to the independent variables, will give the components of the gradient vector, that is

$$\nabla f(x) \equiv \begin{pmatrix} \bar{v}_1 \\ \vdots \\ \bar{v}_n \end{pmatrix}.$$

In this way, the reverse mode computes the derivative of the *dependent variable* with respect to *all the variables* involved in the computation of the function. The derivative calculation can only be performed *after* the function is evaluated, because the values of the intermediate quantities v_j , with $j \in \mathcal{I}_i$, must be available as arguments of φ_i in (2.17), for the computation of each adjoint.

The following extended program evaluates $f(x)$ and $\nabla f(x)$ by the reverse mode of automatic differentiation.

Reverse Mode

Given $v_1 \equiv x_1, \dots, v_n \equiv x_n$,

Initialization

$$\begin{aligned}\bar{v}_i &\leftarrow 0, & i = 1, \dots, r-1, \\ \bar{v}_r &\leftarrow 1,\end{aligned}$$

Function evaluation

for $i = n+1, \dots, r$

$$v_i \leftarrow \varphi_i(v_j)_{j \in \mathcal{I}_i},$$

$$y = v_r,$$

Reverse accumulation

for $i = r, \dots, n+1$

$$\bar{v}_k \leftarrow \bar{v}_k + \frac{\partial \varphi_i(v_j)_{j \in \mathcal{I}_i}}{\partial v_k} \cdot \bar{v}_i \quad \text{for each } k \in \mathcal{I}_i,$$

$$\partial y / \partial v_i \leftarrow \bar{v}_i, \quad i = 1, \dots, n.$$

(2.19)

Basically, the propagation of adjoints reduces to a sequence of multiplications, additions and assignments. These simple rules allow the gradient computation to proceed at a cost that is a *fixed constant* times the cost required to evaluate the function itself, *no matter how many independent variables there are*. If the evaluation of the function involves R elementary operations, then the evaluation of the gradient will require approximately $c \cdot R$ operations, where $c \leq 5$, as shown by Griewank in [32].

Additionally, Iri showed in [42], [46], and [43] that as a by-product of the reverse mode, it is possible to obtain fairly sharp estimates for bounds on the rounding errors in the computed function value.

Despite the advantages of the reverse mode regarding time complexity, difficulties might arise at the implementation level, since it is necessary to access in reverse order the entire sequence of operations executed during the function evaluation. Most of the current implementations achieve this by recording the sequence of elementary operations in a file, or *trace*, while the function is being evaluated, and then accessing this information in a reverse fashion. But, since practical problems tend to involve millions of elementary operations, this file may require a very substantial or even prohibitive amount of storage.

Recently, Griewank [33] showed theoretically that more reasonable compromises between temporal and spatial complexity can be achieved by employing a technique called *recursive checkpointing*. More specifically, he showed that if one accepts an increase in the number of operations by a factor K , then the storage required by the reverse mode is limited essentially by the $\sqrt[K]{T}$, where T is the run-time for evaluating the original function (see Bischof [10]). This approach has not been implemented yet, but the claim is that it can reduce the memory requirements of the reverse mode, and facilitate its application to computations of virtually any length.

2.6 Graphical Interpretation

The process of evaluating a function and its derivatives can be “naturally” represented in terms of a *computational graph*. This is a *directed acyclic* graph, whose vertex set is formed by all the quantities involved in the computation of the function: independent, intermediate, dependent variables, and possibly constants. Every variable is represented in the graph by a vertex. An arc runs from one vertex to another

if the variable associated with the latter *depends directly* on the variable associated with the former. In other words, for each elementary operation

$$v_i \leftarrow \varphi_i(v_j)_{j \in \mathcal{I}_i}$$

in the evaluation program (2.14), the vertices corresponding to the arguments v_j , with $j \in \mathcal{I}_i$, are connected in the graph to the vertex corresponding to the result v_i . Each arc connecting two vertices has associated with it the value of an elementary partial derivative: the derivative of the variable corresponding to the vertex destination with respect to the variable corresponding to the vertex origin.

For example, for an elementary operation of the form

$$v_i \leftarrow \varphi_i(v_{j_1}, v_{j_2}),$$

the arc connecting v_{j_1} with v_i will have associated the value of $\partial\varphi_i(v_{j_1}, v_{j_2})/\partial v_{j_1}$, and the arc connecting v_{j_2} with v_i will have associated the value of $\partial\varphi_i(v_{j_1}, v_{j_2})/\partial v_{j_2}$, as shown in Figure 2.1.

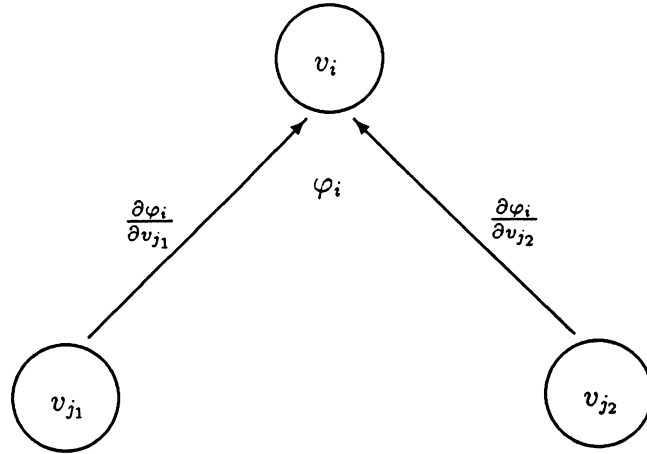


Figure 2.1: Elementary partial derivatives for $v_i \leftarrow \varphi_i(v_{j_1}, v_{j_2})$

Let us consider again the function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, defined by

$$y = f(x_1, x_2) = 1.5 \cdot (x_1 \cdot x_2 + \exp(x_2)) - \cos(x_1).$$

For a given $x = (v_1, v_2)^T$, the corresponding evaluation process can be performed by the sequence of elementary assignments (2.4), that is

$$\begin{aligned}
 v_3 &\leftarrow \varphi_3(v_1, v_2) &\equiv v_1 \cdot v_2 \\
 v_4 &\leftarrow \varphi_4(v_2) &\equiv \exp(v_2) \\
 v_5 &\leftarrow \varphi_5(v_3, v_4) &\equiv v_3 + v_4 \\
 v_6 &\leftarrow \varphi_6(v_5) &\equiv 1.5 \cdot v_5 \\
 v_7 &\leftarrow \varphi_7(v_1) &\equiv \cos(v_1) \\
 y = v_8 &\leftarrow \varphi_8(v_6, v_7) &\equiv v_6 - v_7.
 \end{aligned}$$

This evaluation process can be represented by the computational graph depicted in Figure 2.2.

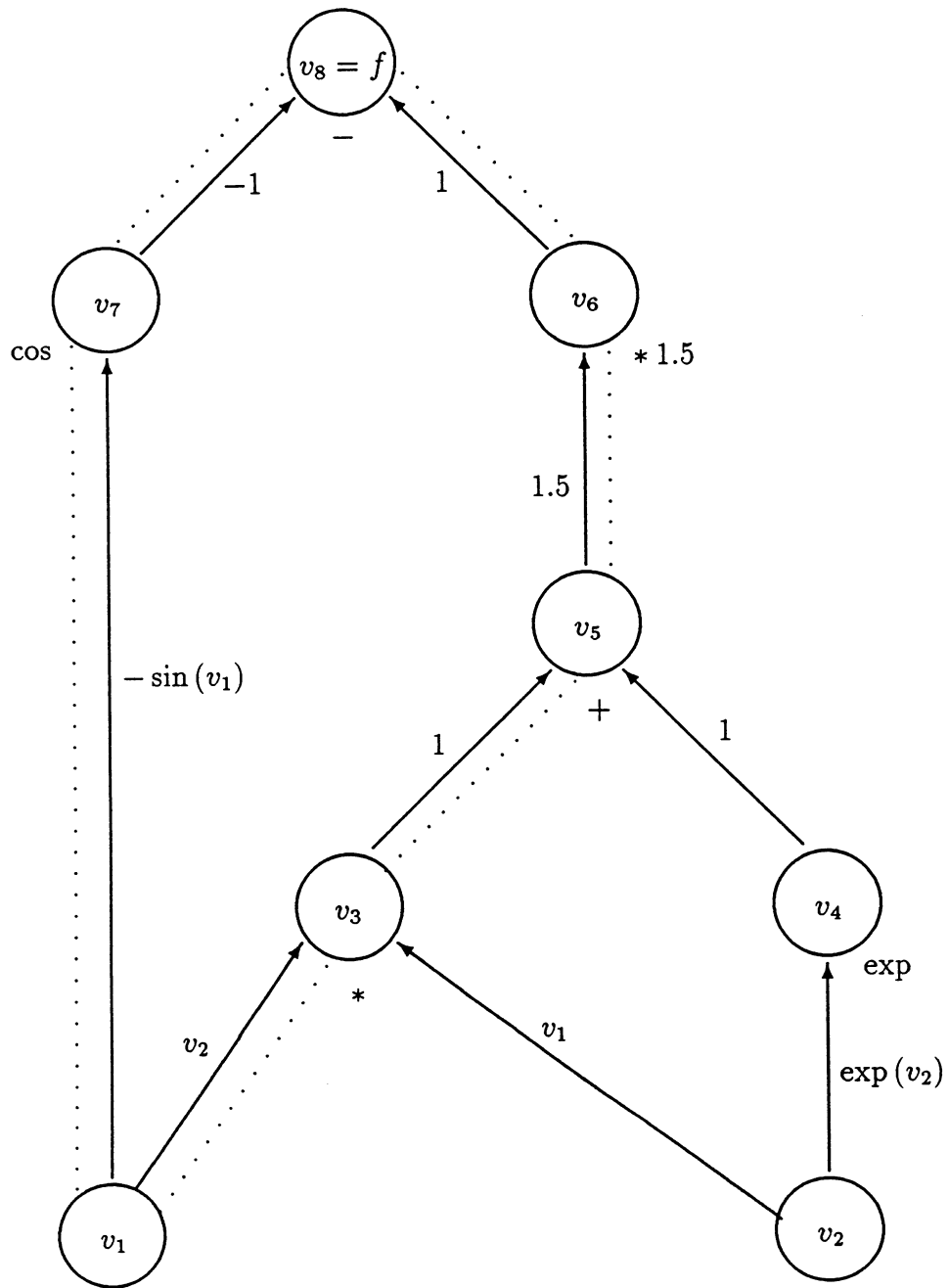


Figure 2.2: Computational graph for $f(x) = 1.5 \cdot (x_1 \cdot x_2 + \exp(x_2)) - \cos(x_1)$

From the computational graph, many valid evaluation processes can be derived. For example, just by relabeling the nodes of Figure 2.2 in an appropriate way, it could represent the evaluation process given in (2.5).

From the information stored in the graph, derivatives can be computed by accumulating the quantities associated with the arcs in an appropriate fashion. Given $x = (v_1, \dots, v_n)^T \in \mathbb{R}^n$, the partial derivative of $f(x)$ with respect to the independent variable v_l can be evaluated as follows

$$\frac{\partial f(x)}{\partial v_l} \equiv \sum_{P \in \mathcal{P}(v_l, f)} \prod_{a \in P} (\partial a)$$

where $\mathcal{P}(v_l, f)$ is the set of all directed paths P that connect v_l to the dependent variable f , and ∂a represents the value of the elementary partial derivative attached to each arc $a \in P$.

For example, in the computational graph of Figure 2.2, in order to calculate $\partial f(x)/\partial v_1$, we must consider all the directed paths that connect the vertex v_1 with the vertex f . They are indicated by the dotted lines. By multiplying the elementary partial derivatives attached to each arc on a path, and finally adding the resulting values of all the paths, we obtain

$$\begin{aligned} \frac{\partial f(x)}{\partial v_1} &= \frac{\partial f}{\partial v_6} \cdot \frac{\partial v_6}{\partial v_5} \cdot \frac{\partial v_5}{\partial v_3} \cdot \frac{\partial v_3}{\partial v_1} + \frac{\partial f}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_1} \\ &= 1.5 \cdot v_2 + \sin(v_1). \end{aligned}$$

The forward and the reverse modes of automatic differentiation correspond to two different ways of traversing the computational graph and using the information contained in it.

In the forward mode, the sequence of computations corresponds to traversing the graph *from the independent variables to the dependent variable*. For this reason, it is also called the *bottom-up* mode. Since the derivative calculation proceeds together

with the function evaluation, the forward mode can be implemented in *one pass* through the computational graph.

On the other hand, the *reverse mode* computes the adjoints in opposite order with respect to the order in which the elementary operations are executed to evaluate the function. This corresponds to traversing the graph *from dependent variable to the independent variables*. For this reason, the reverse mode is also called the *top-down* mode. As we mentioned in Section 2.5, the derivative calculation can be performed only *after* the function has been evaluated so that the values of all the intermediate quantities are available for calculating the adjoints. This mode can be implemented in *two passes* through the computational graph. In the first pass, or *forward sweep*, the function is evaluated. In the second pass, or *reverse sweep*, the adjoints are computed.

Apparently, the graphical representation of evaluation programs was first proposed by Kantorovich [50]. It has been extensively used in the context of error estimation (see Bauer [1] and Miller [53]), a task that is closely related to the reverse mode.

An important use of the computational graph is to provide dependency information that can be used for the parallel evaluation of the function and its derivatives (see Bischof et al. [8], [9]).

2.7 Computation of Jacobians

As in the scalar case, the computational process for evaluating a vector function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$$f(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{pmatrix} \equiv \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix},$$

can be regarded as a sequence of elementary unary and binary operations. For a given $x = (v_1, \dots, v_n)^T$, this evaluation process can be represented as follows

Function Evaluation

Given $v_1 \equiv x_1, \dots, v_n \equiv x_n$,

for $i = n + 1, \dots, r_1, \dots, r_m$

$v_i \leftarrow \varphi_i(v_j)_{j \in \mathcal{I}_i}$,

$y_1 \leftarrow v_{r_1}$

\vdots

$y_m \leftarrow v_{r_m}$.

(2.20)

Here, there are m dependent variables: y_1, \dots, y_m .

Automatic differentiation can be applied to evaluate the corresponding $m \times n$ Jacobian matrix

$$J(x) \equiv \begin{pmatrix} \frac{\partial}{\partial x_1} f_1(x) & \cdots & \frac{\partial}{\partial x_n} f_1(x) \\ \vdots & & \vdots \\ \frac{\partial}{\partial x_1} f_m(x) & \cdots & \frac{\partial}{\partial x_n} f_m(x) \end{pmatrix},$$

at a particular argument $x \in \mathbb{R}^n$.

There are different ways in which this matrix can be calculated using automatic differentiation. Naturally, either the forward or the reverse mode could be applied *separately* to the components $f_1(x), \dots, f_m(x)$ of the vector function $f(x)$, as explained in Sections 2.4 and 2.5. In this way, we would obtain independently of each other, the m gradient vectors $\nabla f_1(x), \dots, \nabla f_m(x)$, which are the rows of the Jacobian. However, this approach may be far from optimal if the functions f_1, \dots, f_m involve many common subexpressions.

Since the computation of the derivatives is intimately related to the evaluation of the underlying function, it is often more efficient to evaluate both function and Jacobian *together*. The following are some of the possible approaches.

- The *entire* Jacobian can be calculated by applying the *forward mode* to the evaluation program (2.20), using the same approach as in (2.14). At the beginning, each independent variable v_i is associated with a vector $\nabla v_i = e^{(i)} \in \mathbb{R}^n$. During the evaluation process, each intermediate quantity v_i ($i = n+1, \dots, r_m$) is computed together with a corresponding n -vector

$$\nabla v_i \equiv \begin{pmatrix} \partial v_i / \partial v_1 \\ \vdots \\ \partial v_i / \partial v_n \end{pmatrix}.$$

The pairs $(v_i, \nabla v_i)$ are propagated throughout the entire sequence of computations, as explained in Section 2.4. At the end, the vectors ∇v_{r_j} associated with the dependent variables y_j ($j = 1, \dots, m$), will give *all the rows* of the Jacobian.

- The *entire* Jacobian can be calculated by applying the *reverse mode* to the evaluation program (2.20), using the same approach as in (2.19), but instead of associating with each intermediate quantity v_i ($i = r_m, \dots, n+1$) a real-valued adjoint, an m -vector is computed:

$$\bar{v}_i \equiv \begin{pmatrix} \partial f_1 / \partial v_i \\ \vdots \\ \partial f_m / \partial v_i \end{pmatrix}.$$

At the beginning, the adjoints corresponding to the dependent variables y_j are initialized by $\bar{y}_j = e^{(j)} \in \mathbb{R}^m$. In the forward sweep, all the intermediate quantities v_i are computed. In the reverse sweep, the adjoint vectors \bar{v}_i are propagated, as explained in Section 2.5. At the end, the vectors \bar{v}_i associated

with the independent variables v_i ($i = 1, \dots, n$), will give *all the columns* of the Jacobian.

- The Jacobian can be calculated *column by column* by the *forward mode*, using the same approach as in (2.15). First, an independent variable is chosen, say v_l ($1 \leq l \leq n$). The associated scalar \hat{v}_l is initialized to 1, and the remaining \hat{v}_i 's to 0. During the evaluation process, each intermediate quantity v_i is computed together with the corresponding scalar $\hat{v}_i = \partial v_i / \partial v_l$. The pairs (v_i, \hat{v}_i) are propagated throughout the sequence of computations, as explained in Section 2.4. At the end of the process, the last m computed scalars \hat{v}_{r_j} , associated with the dependent variables y_j , will give the values of $\partial f_j / \partial v_l$, $j = 1, \dots, m$, which are the entries in the l^{th} column of the Jacobian. This process can be repeated n times, for $l = 1, \dots, n$, to obtain all the columns.

If the particular application does not require the evaluation of the Jacobian $J \equiv J(x)$, but instead the product Ju for some vector $u \in \mathbb{R}^n$, this can be done using the same approach, and initializing $\hat{v}_i = u_i$, for $i = 1, \dots, n$, at the beginning of the algorithm.

- The Jacobian can be calculated *row by row* by the *reverse mode*, using the same approach as in (2.19). First, a dependent variable is chosen, say f_l ($1 \leq l \leq m$). The associated adjoint \bar{v}_{r_l} is initialized to 1, and the other adjoints to 0. In the forward sweep, all the intermediate variables v_i are computed. In the reverse sweep, the scalars \bar{v}_i are propagated, as explained in Section 2.5. At the end of the process, the last n computed adjoints \bar{v}_i , associated with the independent variables v_i , will give the values of $\partial f_l / \partial x_i$, $i = 1, \dots, n$, which are the entries in the l^{th} row of the Jacobian. This process can be repeated m times, for $l = 1, \dots, m$, to obtain all the rows.

As in the previous case, for some vector $w \in \mathbb{R}^m$, the product $w^T J$ can be computed by using the same approach, and initializing the adjoints $\bar{v}_{r_j} = w_j$, for $j = 1, \dots, m$, at the beginning of the algorithm.

Griewank [34] related the task of computing the Jacobian to the problem of *successively eliminating intermediate vertices in the computational graph* that represents the function. This process involves an accumulation procedure which can be viewed mathematically as a generalization of the chain rule. He conjectured that the problem of finding the optimal accumulation procedure (with minimal number of arithmetic operations) is NP-hard.

Jacobian matrices are often sparse, and the dependency information obtained from the computational graph can be used to deduce the *sparsity structure*. Notice that if there is a path from the i^{th} independent variable v_i to the j^{th} dependent variable f_j , then $\partial f_j / \partial v_i \neq 0$ (in the absence of numerical cancellation), which means that the component in the position (i, j) of the Jacobian is nonzero. In this way, it is possible to determine all the nonzero entries. This approach is discussed by Bischof et al. in [8].

Once the sparsity structure is known, by using the *graph coloring* approach of Coleman, Garbow and Moré [17] (which originated in the context of finite-difference approximations, as mentioned in Section 2.1), it is possible to identify the component functions f_j that depend on disjoint subsets of independent variables. The gradients of component functions that belong to the same subset (i.e., were assigned the same color by the graph coloring procedure), can be evaluated in the same pass of the reverse mode. This procedure can considerably decrease the number of reverse passes required to compute all the rows of the Jacobian. Similarly, graph coloring can be applied to detect a suitable partition of the columns of the Jacobian, so that all the columns in each group can be computed in one pass of the forward mode.

As in the case of finite-difference approximations, the use of graph coloring techniques could improve the computational efficiency of automatic differentiation when the Jacobian is sparse.

2.8 Implementations of Automatic Differentiation

Basically, the principle underlying automatic differentiation implementations is the addition of extra computations to the original function evaluation program, to evaluate both the function and the required derivatives. A wide variety of implementations have been developed over the years, mostly in the form of FORTRAN *precompilers* and *overloading utilities* in Pascal-SC, Ada, C++, etc. In this section, we will summarize the features of these two approaches. A survey and classification of twenty-nine available software tools for automatic differentiation has been done by Juedes in [47].

- **FORTRAN Precompilers**

A precompiler for automatic differentiation is a special-purpose program that transforms the sequence of instructions for the evaluation of a function into a sequence of instructions (or subroutines) for the evaluation of the function and the required derivatives.

In the case of FORTRAN precompilers, the user is usually required to supply as input the source code for evaluating the function in some “dialect” of FORTRAN, and to nominate the dependent and independent variables. The source code is then fed to the precompiler, which analyzes the arithmetic assignment statements, in a way similar to a compiler. All calculations involving real variables are broken down into elementary arithmetic operations and univariate functions, as explained in Section 2.3. Control statements remain unaltered.

For each elementary operation, the precompiler already has built-in expressions for the corresponding partial derivatives. With this knowledge, it produces as

output an extended FORTRAN program that, upon execution, will evaluate both the function and the required derivatives.

In the forward mode, the amount of storage can be predicted, since it is proportional to the number of independent variables (unless a sparse storage technique is being used). In the reverse mode, the storage is proportional to the total number of elementary operations executed, which makes the storage allocation a more complex issue, as mentioned in Section 2.5.

A drawback of many existing precompilers is that they usually impose restrictions on the kind of codes that they accept as input, since they support only a subset of the language. Additionally, the code generated as output tends to be rather cryptic. On the other hand, precompilers allow scope for performing an optimization of the resulting code, to improve its efficiency.

In 1980 Speelpenning [64] wrote a precompiler called JAKE, that implements the reverse mode. JAKE was upgraded at Argonne National Laboratory to JAKEF [38], which can calculate first derivatives of vector functions. Other precompilers are GRESS [40], PADRE2 [45], and ADIFOR [6].

GRESS can compute first derivatives using either the forward or the reverse mode. It has been successfully applied on a variety of problems. Examples of the application of GRESS can be found in [41] and [39].

PADRE2 also implements both the forward and the reverse modes, and allows the computation of first and second-order derivatives, as well as rounding error estimates.

In all these implementations either the number of independent variables or the total number of arithmetic operations must be of moderate size relative to the available computing environment.

ADIFOR uses a *hybrid* approach. It is based on the forward mode, but employs the reverse mode to compute the gradients of expressions on the right-hand sides of assignment statements. This avoids some of the memory limitations of the straightforward reverse mode. The current implementation of ADIFOR allows the computation of first-order derivatives of vector functions. The user can also incorporate information available about the sparsity structure of derivative matrices in order to evaluate them more efficiently. The authors intend to provide, in an upgraded version of the software, other capabilities such as the computation of second and higher-order derivatives, the automatic detection of sparsity, and the detection and handling of exceptions (see Section 2.9).

- **Operator Overloading**

Operator overloading is a feature of modern programming languages like Ada, C++, Pascal-SC, and others. In these languages, the compiler can be assigned the task of executing certain instructions to evaluate the derivatives corresponding to each elementary operation. First, the user has to identify all the variables that correspond to “differentiable quantities”. Usually, this set consists of the independent variables and all quantities that directly or indirectly depend on them. The occurrence of such variables as arguments of an elementary operation triggers the compiler to issue additional instructions for the calculation of the corresponding derivatives. Every operation is associated to the corresponding derivative calculation.

Compared to precompilation, operator overloading may require more insight from the user, but it does not generate intermediate source code. All the derivative calculations are handled at the compiler level.

A drawback of this approach is that it is difficult to implement any optimization in the derivative calculation that transcends elementary operations, since all the computations are performed as by-products of these elementary operations. Bischof studied in detail this issue in [8]. Another limitation is that the majority of the programs from which it is necessary to compute derivatives are written in FORTRAN77, which does not support operator overloading.

The implementation of the forward mode via operator overloading is quite straightforward. A detailed description can be found in Rall [63]. For implementing the reverse mode, all computations performed during the function evaluation are recorded on a file, and then an interpreter performs a backward pass on the data.

Apparently, the first implementation of automatic differentiation by operator overloading is due to Kedem [51]. His software AUGMENT allowed the computation of gradients and truncated Taylor series in the forward mode. A few years later, Rall [63] achieved an implementation of the forward mode in Pascal-SC, an extension of PASCAL.

The forward evaluation of first and second-order derivatives in Ada is discussed by Dixon and Mohseninia in [26].

Both the forward and the reverse modes have been implemented in the C++ package ADOL-C, developed by Giewank et al. [36]. ADOL-C allows the computation of first and higher-order derivatives of vector functions. It is the only tool available that supports the computation of arbitrarily high-order derivatives.

2.9 Special Cases: Nondifferentiability and Branching

As mentioned earlier, automatic differentiation is based on the application of the chain rule. The computed results are guaranteed to be correct provided that all the elementary functions involved in the original program are sufficiently smooth in a neighborhood of the points at which they are evaluated. Computer programs may violate the smoothness requirements by *evaluating elementary functions at points of nondifferentiability*, or by *branching*.

2.9.1 Nondifferentiability

We call a function *nondifferentiable* if there are points in its domain for which its derivative does not exist. An attempt to evaluate derivatives at a point of mathematical nondifferentiability is called an *exception*.

Elementary operations such as $+$, $-$, \times , \sin , \cos , \exp , etc., are everywhere differentiable. Other elementary operations such as $/$, \tan , \ln , \log_{10} , etc., fail to be differentiable *only* at points where they fail to be defined. The operations in both sets are considered to be differentiable, because they are mathematically differentiable at each point in their domain of definition. The handling of exceptions for these operations should be in the scope of the original program (unless overflow occurs in the evaluation of the derivatives but not in the function).

On the other hand, functions such as abs , sign , max , min , etc., are nondifferentiable at certain points. These cases can be detected by the automatic differentiation software, which can generate special code to handle the points of nondifferentiability.

Bischof et al. present in [11] a complete discussion about the detection and handling of exceptions in the precompiler ADIFOR.

2.9.2 Branching

Conditional (IF) statements in a program, may define functions that are not differentiable or that are not even continuous.

When automatic differentiation is applied to the program that evaluates the function, the flow of control remains unaltered. Thus, if the original code contains conditional statements, the resulting “differentiated” code will contain corresponding conditional statements as well. When this code is evaluated at a point where a branching occurs, the computed derivative value, at this point, may be incorrect. The following examples illustrate this situation.

Suppose that the original evaluation program employs a conditional statement to compute the absolute value $abs(x)$, for some $x \in \mathbb{R}$, as follows

$$abs(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0. \end{cases}$$

Automatic differentiation, applied to the above conditional, would compute

$$abs'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0. \end{cases} \quad (2.21)$$

However, if the function $abs(x)$ is evaluated by the conditional

$$abs(x) = \begin{cases} x & \text{if } x > 0 \\ -x & \text{if } x \leq 0, \end{cases}$$

automatic differentiation would give

$$abs'(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x \leq 0. \end{cases} \quad (2.22)$$

Finally, another way of computing $abs(x)$ could be

$$abs(x) = \begin{cases} x & \text{if } x > 0 \\ -x & \text{if } x < 0 \\ 0 & \text{if } x = 0, \end{cases}$$

and in this case automatic differentiation would give

$$abs'(x) = \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0. \end{cases} \quad (2.23)$$

The three different, but equivalent, ways of computing $abs(x)$ give $abs'(0) = 1$ in (2.21), $abs'(0) = -1$ in (2.22), and $abs'(0) = 0$ in (2.23).

Another example is the following. Given $x \in \mathbb{R}$, assume that the function $f(x) = x^2$ is evaluated by the conditional statement

$$f(x) = \begin{cases} 1 & \text{if } x = 1 \\ x^2 & \text{otherwise,} \end{cases}$$

then, automatic differentiation would compute

$$f'(x) = \begin{cases} 0 & \text{if } x = 1 \\ 2x & \text{otherwise,} \end{cases}$$

which, evaluated at $x = 1$, would give the incorrect value $f'(1) = 0$.

We should point out that in these two examples, the incorrect results are not due to automatic differentiation, but to the way in which the functions are defined. The user of automatic differentiation should be aware of the possibility that conditional statements can produce incorrect derivative values, if they are evaluated at points where the branching occurs.

2.10 Future Developments

The chain-rule based technique of automatic differentiation can be a convenient software tool. Given the code for evaluating the function, first and higher-order derivatives can be calculated in a completely mechanical fashion. However, even if this technique has been used in many different fields (see the papers published in the proceedings [35]), it is not yet considered as a “standard tool” in scientific computations.

Aspects such as *computational time*, *storage requirements*, and *user-friendliness* are being improved in current implementations.

As Iri mentions in his comprehensive survey [43], it is expected that automatic differentiation software will offer capabilities such as

- *rounding error estimation*,
- *flexibility in choosing among the forward or the reverse mode*,
- *differentiation of multivariate implicit functions*,
- *detection and handling of exceptions*,
- *possibility of integration with special-purpose packages, such as optimization, differential equations, or others*,
- *efficient implementation on high-performance computers*,
- *portability of the software to different computing environments*,
- *ability to deal efficiently with large-scale applications*,
- *efficient utilization of parallelism and vectorization*.

The availability of accurate derivatives at a reasonable cost could greatly improve the design and implementation of algorithms and aid in the modeling and solution of many large-scale nonlinear problems, in which finite-difference approximations can not be trusted, and symbolic approaches are infeasible.

Cooperation with compiler developers may be indispensable, in order to enhance the efficiency in time and space of automatic differentiation. This has been confirmed by the successful results obtained with the precompiler ADIFOR (see [6]), a joint effort between the compiler group and the numerical optimization group in the

Center of Research in Parallel Computation, an NSF Science and Technology Center headquartered at Rice University.

The issue of parallelism has also crucial importance. The works of Dixon [25], Fischer [28], Bischof et al. [8], and others, deal with this aspect.

To date, automatic differentiation studies have been primarily focused in the evaluation of first and second-order derivatives. However, algorithms designed specifically for the evaluation of higher-order derivatives have been proposed by Kalaba [49], Wexler [69], and Neidinger [58], and software for this purpose has been implemented by Berz [4] and Michelotti [52].

Golman suggests in [31] that a combination between symbolic differentiation and automatic differentiation could also provide efficient derivative calculation.

It is important to point out that automatic differentiation is more than just “implementing the chain rule.” Even the efficient evaluation of first derivatives is a hard problem in the sense of computational complexity. The evaluation of second and higher-order derivatives also poses many questions, whose resolution will require further research and experimentation in mathematics and computer science.

Chapter 3

Automatic Differentiation and Parameterized Fixed-Point Iterations

In the previous chapter, we discussed the chain-rule based technique of automatic differentiation. This technique has proved to be able to compute correct derivatives for a variety of functions with different degrees of complexity (see for example the papers in [35]). However, until recently it was not clear if it could be expected to yield useful derivative values when the evaluation program involves an iterative process with a variable number of steps. Many functions of practical interest are evaluated by iterative processes, for example those that are implicitly defined as solutions of systems of algebraic or differential equations.

In this chapter, we will be concerned with an application of automatic differentiation in which the derivative to be computed is $\partial x_*(p)/\partial p \in \mathbb{R}^{n \times n_p}$, where $x \equiv x_*(p) \in \mathbb{R}^n$ is a solution of the system of parameterized nonlinear equations

$$f(x, p) = 0,$$

with $f : \mathbb{R}^{n+n_p} \rightarrow \mathbb{R}^n$, and $p \in \mathbb{R}^{n_p}$ is a vector of parameters.

We will assume that the algorithm employed to solve this system for a given value of p executes an iterative process of the form

$$x_{k+1} = \phi_k(x_k, p), \quad k = 0, 1, 2, \dots \quad (3.1)$$

We will discuss these “generalized fixed-point iterations” in more detail in Section 3.1.

We are interested in applying automatic differentiation to this iterative process, considering the components of the vector p as the independent variables, and the components of each iterate $x_k(p)$ generated by (3.1) as the dependent variables.

For a given value $p \equiv \bar{p}$, the “differentiated” code upon execution will generate two sequences: the sequence of iterates $\{x_k(\bar{p})\} \subset \mathbb{R}^n$ and the corresponding sequence of derivatives $\{\partial x_k(\bar{p})/\partial p\} \subset \mathbb{R}^{n \times n_p}$. Assuming that $\{x_k(\bar{p})\}$ converges to a solution $x_*(\bar{p})$ of the system, we are interested in whether $\{\partial x_k(\bar{p})/\partial p\}$ converges to $\partial x_*(\bar{p})/\partial p$.

J. C. Gilbert examined the application of automatic differentiation to parameterized fixed-point iterations in [30]. He identified a class of iterations that satisfy the following property: under certain smoothness assumptions, if the sequence of iterates $\{x_k(\bar{p})\}$ generated by the algorithm converges to a solution $x_*(\bar{p})$ of the system, then the sequence of derivatives $\{\partial x_k(\bar{p})/\partial p\}$ converges to $\partial x_*(\bar{p})/\partial p$. He showed that this class of iterative processes includes Newton’s method. No experimental results were provided in this work.

The hypotheses of Gilbert’s convergence theorem do not hold for Broyden’s method or for some other useful iterative methods. It is still an open question whether a more general convergence result can be proved for these algorithms. As we will see in Chapter 4, the numerical results obtained for Broyden’s method are quite promising in this respect since the sequence of derivatives converged for all the tested problems.

This chapter is organized as follows. In Section 3.1 we present background material about some iterative methods for solving systems of nonlinear equations. For a complete discussion in this area, the interested reader may want to see the book by Dennis and Schnabel [24]. We will focus in particular on Newton’s method and Broyden’s method. In Section 3.2 we discuss the application of automatic differentiation to generalized fixed-point iterations and present some convergence results.

We will denote by $\|\cdot\|$ the l_2 vector norm or any matrix norm that is consistent with the l_2 norm in the sense that $\|Ax\|_2 \leq \|A\| \|x\|_2$ for each $x \in \mathbb{R}^n$ and $A \in \mathbb{R}^{n \times n}$.

3.1 On the Solution of Systems of Nonlinear Equations

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ be a mapping with domain and range in \mathbb{R}^n , and consider the problem of finding a solution of the system of n nonlinear equations in n unknowns

$$f(x) \equiv \begin{pmatrix} f_1(x) \\ \vdots \\ f_n(x) \end{pmatrix} = 0, \quad (3.2)$$

where $f_i : \mathbb{R}^n \rightarrow \mathbb{R}$, $1 \leq i \leq n$.

The following are the so-called “standard Newton assumptions” for problem (3.2).

- (a) The mapping f is continuously differentiable in an open convex set $\mathcal{D} \subset \mathbb{R}^n$.
- (b) There is an $x_\star \in \mathcal{D}$ such that $f(x_\star) = 0$ and $f'(x_\star)$, the Jacobian of f evaluated at x_\star , is nonsingular.
- (c) f' is Lipschitz continuous at x_\star in \mathcal{D} , i.e., there exists a constant $\gamma \in \mathbb{R}$ such that

$$\|f'(x) - f'(x_\star)\| \leq \gamma \|x - x_\star\| \quad \text{for all } x \in \mathcal{D}.$$

Assumptions (a) and (b) guarantee that x_\star is a locally unique solution of system (3.2). The bound given in (c) is useful in the proofs of convergence for different methods.

The best known method for attacking problem (3.2) is *Newton's method*, which can be formulated as the iteration

$$x_{k+1} = x_k - f'(x_k)^{-1} f(x_k), \quad k = 0, 1, 2, \dots \quad (3.3)$$

Under the assumptions (a)-(c) given above, if Newton's method is started from an initial point x_0 which is "sufficiently close" to x_* , then it can be shown that the sequence of iterates $\{x_k\}$ generated by (3.3) is well-defined and converges to x_* *q-quadratically*. In practice, q-quadratic convergence implies that the number of significant digits in x_k as an approximation to x_* at least doubles at each iteration, once x_k is near x_* (see Dennis and Schnabel [24], Chapter 5).

A *secant* method is defined by an iteration of the form

$$x_{k+1} = x_k - A_k^{-1} f(x_k), \quad k = 0, 1, 2, \dots, \quad (3.4)$$

where $A_k \in \mathbb{R}^{n \times n}$ is an approximation to $f'(x_k)$. At each iteration k , the matrix A_k is updated to a matrix A_{k+1} that must satisfy the so-called *secant equation*

$$A_{k+1} s_k = y_k,$$

where

$$s_k = x_{k+1} - x_k \quad \text{and} \quad y_k = f(x_{k+1}) - f(x_k).$$

Broyden's method is defined by an iteration like (3.4), in which the matrices A_k are updated by the formula

$$A_{k+1} = A_k + \frac{(y_k - A_k s_k) s_k^T}{s_k^T s_k}, \quad k = 0, 1, 2, \dots, \quad (3.5)$$

where the initial matrix A_0 is often computed as a finite-difference approximation to $f'(x_0)$.

Under the assumptions (a)-(c) given above, if the initial iterate x_0 is "sufficiently close" to x_* , and if the initial matrix A_0 is "sufficiently close" to $f'(x_*)$, then it can be shown that the sequence of iterates $\{x_k\}$ generated by Broyden's method is well-defined and converges to x_* *q-superlinearly* (see Dennis and Schnabel [24], Chapter 8).

Broyden's method is computationally less expensive per iteration (in terms of arithmetic operations performed) than Newton's method, and it does not require the evaluation of the Jacobian, but the price paid is a reduction in the rate of convergence, from q-quadratic to q-superlinear.

In both methods, convergence is guaranteed only when the starting point x_0 is a good approximation to x_* . For this reason they are called *local methods*. To facilitate the convergence from poor starting points, these methods are augmented by the so-called *globalization strategies*. The two main approaches are the *line-search* methods and the *trust-region* methods (see Dennis and Schnabel [24], Chapter 6).

Newton's method is an example in a class of iterative processes which are known as *fixed-point iterations*. The iterative processes in this class are defined by iterations of the form

$$x_{k+1} = \phi(x_k), \quad k = 0, 1, 2, \dots \quad (3.6)$$

The mapping $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is called an *iteration function*, and it is chosen so that any solution of the equation

$$x = \phi(x),$$

called a *fixed point*, is also a solution of the original system $f(x) = 0$. For a detailed discussion about fixed-point iterations, the reader is referred to the book by Conte and de Boor [19].

If the iteration function ϕ is continuous and if the sequence $\{x_k\}$ generated by (3.6) is well-defined and converges to a point $\xi \in \mathbb{R}^n$, then ξ is a fixed point of $\phi(x)$, since

$$\xi = \lim_{k \rightarrow \infty} x_{k+1} = \lim_{k \rightarrow \infty} \phi(x_k) = \phi(\lim_{k \rightarrow \infty} x_k) = \phi(\xi).$$

Notice that Newton's method, defined by the iteration (3.3), is a special case of (3.6) in which the iteration function ϕ is given by

$$\phi(x) \equiv x - f'(x)^{-1} f(x). \quad (3.7)$$

Clearly, if x_* is a fixed point of the iteration function $\phi(x)$ in (3.7), then it is also a solution of the equation $f(x) = 0$, since $f'(x)$ is continuous at x_* and $f'(x_*)$ is nonsingular (under the "standard Newton assumptions").

On the other hand, iterative processes of the form (3.4) cannot be described by (3.6) because the iteration function ϕ_k depends on the choice of the matrix A_k , which may not be a function of the iterate x_k alone, as in the case of Broyden's method.

A more general iterative process, which encompasses iterations of the form (3.4) and (3.6), is given by

$$x_{k+1} = \phi_k(x_k), \quad k = 0, 1, 2, \dots, \quad (3.8)$$

where $\phi_k : \mathbb{R}^n \rightarrow \mathbb{R}^n$, for all k . We will call (3.8) a *generalized fixed-point iteration*. Any point $\xi \in \mathbb{R}^n$ such that

$$\xi = \phi_k(\xi) \quad \text{for all } k,$$

is called a fixed point of the sequence $\{\phi_k\}$.

Let us consider the generalized fixed-point iteration given by

$$x_{k+1} = \phi_k(x_k) \equiv x_k - A_k^{-1} f(x_k), \quad k = 0, 1, 2, \dots, \quad (3.9)$$

where the matrices $A_k \in \mathbb{R}^{n \times n}$ are nonsingular.

Notice that Newton's method can be regarded as a particular instance of (3.9) taking $A_k \equiv f'(x_k)$, for all k . Clearly, Broyden's method is also an example of this

kind of iteration, with the matrices A_k updated by formula (3.5).

Now, one may wonder what condition can be imposed on the sequence of matrices $\{A_k\}$ in (3.9) so that, if the generated sequence $\{x_k\}$ converges to a point, this point is a solution of the original system (3.2). Here, we prove a theorem that states one such condition.

Theorem 3.1

Given a sequence of nonsingular matrices $\{A_k\} \subset \mathbb{R}^{n \times n}$, assume that the sequence $\{x_k\}$ generated by (3.9) converges to a point $x_ \in \mathbb{R}^n$. Then, if the sequences $\{A_k\}$ and $\{A_k^{-1}\}$ are uniformly bounded, $f(x_*) = 0$.*

Proof

By the Bolzano–Weierstrass Theorem, since the sequence $\{A_k\}$ is uniformly bounded, it has a convergent subsequence, say $\{A_{k_j}\}$. Thus, there exists a matrix $A_* \in \mathbb{R}^{n \times n}$ such that

$$\lim_{j \rightarrow \infty} A_{k_j} = A_*.$$

Since the matrices A_{k_j} are nonsingular for all j , we can consider the corresponding subsequence of inverse matrices $\{A_{k_j}^{-1}\}$, which is uniformly bounded, because $\{A_k^{-1}\}$ is uniformly bounded.

Again, by the Bolzano–Weierstrass Theorem, $\{A_{k_j}^{-1}\}$ has a convergent subsequence, say $\{A_{k_{j_l}}^{-1}\}$. Thus, there exists a matrix $B_* \in \mathbb{R}^{n \times n}$ such that

$$\lim_{l \rightarrow \infty} A_{k_{j_l}}^{-1} = B_*.$$

Now, since

$$A_{k_{j_l}} A_{k_{j_l}}^{-1} = I, \quad \text{for all } l,$$

it follows that

$$I = \lim_{l \rightarrow \infty} (A_{k_{j_l}} A_{k_{j_l}}^{-1})$$

$$\begin{aligned}
&= (\lim_{l \rightarrow \infty} A_{k_{j_l}}) (\lim_{l \rightarrow \infty} A_{k_{j_l}}^{-1}) \\
&= A_{\star} B_{\star}.
\end{aligned}$$

Similarly, we can obtain that

$$I = B_{\star} A_{\star}.$$

Hence, A_{\star} is nonsingular and $B_{\star} = A_{\star}^{-1}$.

By hypothesis the sequence $\{x_k\}$ generated by (3.9) converges to x_{\star} . Thus, any subsequence of $\{x_k\}$ also converges to x_{\star} ; in particular $\{x_{k_{j_l}}\}$ converges to x_{\star} . The elements of this subsequence are defined by the iteration (3.9). Therefore,

$$x_{k_{j_l}+1} = x_{k_{j_l}} - A_{k_{j_l}}^{-1} f(x_{k_{j_l}}), \quad \text{for all } j.$$

Hence,

$$\begin{aligned}
x_{\star} &= \lim_{l \rightarrow \infty} x_{k_{j_l}+1} \\
&= \lim_{l \rightarrow \infty} \phi_{k_{j_l}}(x_{k_{j_l}}) \\
&= \lim_{l \rightarrow \infty} (x_{k_{j_l}} - A_{k_{j_l}}^{-1} f(x_{k_{j_l}})) \\
&= x_{\star} - A_{\star}^{-1} f(x_{\star}).
\end{aligned}$$

Since A_{\star} is nonsingular, from the above equation it follows that

$$f(x_{\star}) = 0.$$

□

In the next section, we will discuss the application of automatic differentiation to a generalized fixed-point iteration of the form (3.9) that depends on a given vector of parameters.

3.2 Differentiation of Parameterized Fixed-Point Iterations

In this section we will consider the problem of finding a solution of the system of *parameterized nonlinear equations*

$$f(x, p) \equiv \begin{pmatrix} f_1(x, p) \\ \vdots \\ f_n(x, p) \end{pmatrix} = 0, \quad (3.10)$$

where $f : \mathbb{R}^{n+n_p} \rightarrow \mathbb{R}^n$ satisfies some smoothness assumptions, $p \equiv (p_1, \dots, p_{n_p})^T \in \mathbb{R}^{n_p}$ is a given vector of parameters, and $x \equiv (x_1, \dots, x_n)^T \in \mathbb{R}^n$ is the vector of unknowns, to be determined for a fixed p .

Notice that for a particular value of p (3.10) determines a system of n nonlinear equations in n unknowns: the components of the vector x .

Let us assume that a sequence of nonsingular matrices $\{A_k\} \in \mathbb{R}^{n \times n}$ is provided, such that both $\{A_k\}$ and $\{A_k^{-1}\}$ are uniformly bounded, and let us consider the generalized fixed-point iteration (3.9) given in the previous section. For problem (3.10), this iterative process becomes

$$x_{k+1} = \phi_k(x_k, p) \equiv x_k - A_k^{-1} f(x_k, p), \quad k = 0, 1, 2, \dots \quad (3.11)$$

Let us assume that the sequence of iterates generated by (3.11) converges to a point $x_*(p) \in \mathbb{R}^n$. By Theorem 3.1, the fact that both $\{A_k\}$ and $\{A_k^{-1}\}$ are uniformly bounded guarantees that this limit point is in fact a solution of (3.10).

Notice that the iterates x_k generated by (3.11) depend on the parameter vector p , that is $x_k \equiv x_k(p)$, for all k . The initial iterate x_0 may also be considered as a function of p , even if it is just the constant function $x_0 \equiv x_0(p)$.

Let us consider an algorithm \mathcal{A} that executes such an iterative process.

Algorithm \mathcal{A}

- Given $p \in \mathbb{R}^{n_p}$, and $x_0 \equiv x_0(p) \in \mathbb{R}^n$.

- Repeat for $k = 0, 1, 2, \dots$

1. compute A_k ,

2. $x_{k+1} = \phi_k(x_k, p) \equiv x_k - A_k^{-1} f(x_k, p)$,

until “stopping criterion” is satisfied.

Iterative algorithms for solving systems of nonlinear equations usually require as input some other user-supplied quantities (for example, tolerances related to the stopping criterion), but we will not consider them in our study. We will be mainly concerned with the parameter vector p and with the sequence $\{x_k(p)\}$ generated by \mathcal{A} .

Given an input value for p , say $p \equiv \bar{p}$, let us assume that \mathcal{A} upon execution generates the iterates $x_1(\bar{p}), \dots, x_r(\bar{p})$, such that the last computed iterate $x_r(\bar{p})$ solves (3.10) according to the stopping criterion. Thus, $x_r(\bar{p}) \equiv x_*(\bar{p})$ is the *computed solution* of the system, for the given value \bar{p} . The execution of this process is described by Figure 3.1.

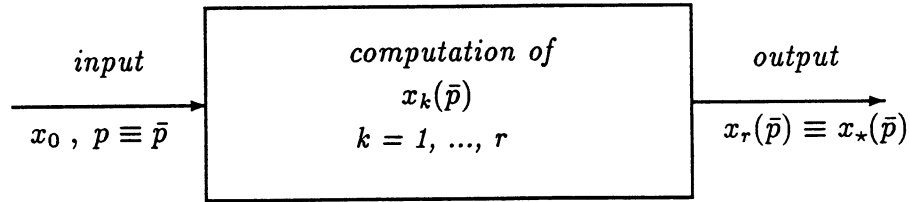


Figure 3.1: Execution of the algorithm \mathcal{A}

Let us assume that automatic differentiation is applied to \mathcal{A} regarding $p \in \mathbb{R}^{n_p}$ as the vector of independent variables and each iterate $x_k(p) \in \mathbb{R}^n$ as the vector of dependent variables. The result will be a “differentiated” algorithm, say \mathcal{A}' , that for

the same input value $p \equiv \bar{p}$ will generate upon execution the sequence $\{x_k(\bar{p})\} \subset \mathbb{R}^n$ and the corresponding sequence of derivatives $\{\partial x_k(\bar{p})/\partial p\} \subset \mathbb{R}^{n \times n_p}$.

Recall from Chapter 2 that automatic differentiation does not alter the original sequence of computations in an algorithm, but rather it *extends* it by adding the required derivative computations. In our case, since \mathcal{A} generates the sequence $x_1(\bar{p}), \dots, x_r(\bar{p})$, then \mathcal{A}' will generate the same sequence, and additionally the sequence of derivatives $\partial x_1(\bar{p})/\partial p, \dots, \partial x_r(\bar{p})/\partial p$. Of course, we are assuming that \mathcal{A}' is started from the same initial point $x_0 \equiv x_0(\bar{p})$ used to start \mathcal{A} . Figure 3.2 describes the execution of this process.

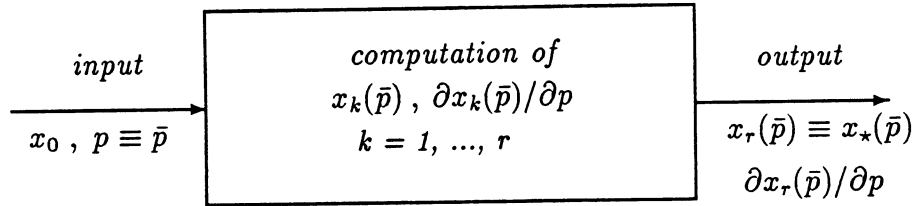


Figure 3.2: Execution of the “differentiated” algorithm \mathcal{A}'

We are interested in studying how the convergence of $\{x_k(\bar{p})\}$ to $x_*(\bar{p})$ in \mathcal{A}' affects the behavior of the sequence of derivatives $\{\partial x_k(\bar{p})/\partial p\}$. Does $\{\partial x_k(\bar{p})/\partial p\}$ converge to $\partial x_*(\bar{p})/\partial p$ when $\{x_k(\bar{p})\}$ converges to $x_*(\bar{p})$?

Notice that the derivative $\partial x_*(\bar{p})/\partial p$ can be computed by solving the so-called *implicit gradient equation*. This equation is obtained by applying the chain rule on the original system (3.10). For $p \equiv \bar{p}$ this gives

$$f_x(x, \bar{p}) \cdot \frac{\partial x(\bar{p})}{\partial p} + f_p(x, \bar{p}) = 0, \quad (3.12)$$

where $f_x(x, \bar{p}) \in \mathbb{R}^{n \times n}$, $\partial x(\bar{p})/\partial p \in \mathbb{R}^{n \times n_p}$, and $f_p(x, \bar{p}) \in \mathbb{R}^{n \times n_p}$. The derivatives f_x and f_p in (3.12) could be computed by automatic differentiation, as discussed in Chapter 2.

Assuming that f_x and f_p are available and that the solution $x_\star(\bar{p})$ of the system has already been computed, then the evaluation of (3.12) at $x_\star(\bar{p})$ gives

$$f_x(x_\star(\bar{p}), \bar{p}) \cdot \frac{\partial x_\star(\bar{p})}{\partial p} = -f_p(x_\star(\bar{p}), \bar{p}). \quad (3.13)$$

This system of linear equations may be solved for $\partial x_\star(\bar{p})/\partial p$ by Gaussian elimination, or by some other method. We will denote the solution of (3.13) by

$$\frac{\partial x_\star(\bar{p})}{\partial p} \equiv \frac{\partial x_{\star IG}}{\partial p},$$

where the subscript “IG” stands for “Implicit Gradient”.

It is important to distinguish between this solution, and the last derivative value computed by automatic differentiation, that is $\partial x_\tau(\bar{p})/\partial p$. We will denote this derivative by

$$\frac{\partial x_\tau(\bar{p})}{\partial p} \equiv \frac{\partial x_{\star AD}}{\partial p},$$

where the subscript “AD” stands for “Automatic Differentiation”. This matrix may not necessarily be equal, or even close, to $\partial x_{\star IG}/\partial p$.

One important consideration here is that the application of automatic differentiation to \mathcal{A} involves the propagation of the derivatives throughout the entire algorithm, which may include a matrix factorization and the solution of a linear system of equations (as in the case of Newton’s method or Broyden’s method). For this reason, the resulting “differentiated” algorithm \mathcal{A}' may be considerably more expensive, in terms of arithmetic operations performed, than the original algorithm \mathcal{A} .

Another important point is that if \mathcal{A} involves the computation of the Jacobian matrix $f_x(x, p)$ (as in the case of Newton’s method), then the application of automatic

differentiation to \mathcal{A} will involve the computation of second-order derivatives of f . These derivatives will show up implicitly in \mathcal{A}' . However, the use of second derivatives should not be necessary. It is clear from equation (3.13) that the solution $\partial x_{*IG}/\partial p$ is defined in terms of first derivatives of f alone, and does not depend on second derivatives.

Therefore, the reader may wonder which of the two approaches mentioned above may be more convenient, namely :

1. computing $\partial x_{*AD}/\partial p$ by applying automatic differentiation on the original iterative algorithm, or
2. computing $\partial x_{*IG}/\partial p$ by solving the system (3.13).

The first approach will be tested in Chapter 4. We will show that if the iterative algorithm employed to solve (3.10) is either Newton's method or Broyden's method, then the resulting derivative computed by automatic differentiation, i.e. $\partial x_{*AD}/\partial p$, is very close to the solution $\partial x_{*IG}/\partial p$ of (3.13) in some appropriate norm.

The second approach seems to be more straightforward than the first and does not involve second-order information about the function. However, equation (3.13) requires the evaluation of the derivatives f_x and f_p , which may not be practical. For example, in certain applications there is no explicit formula or computational code for evaluating f because this function is implicitly embedded by the iterative algorithm that solves (3.10). In such a case, it is not possible to compute explicitly the derivatives f_x and f_p . Notice also that in order to solve (3.13) it is necessary to compute beforehand the solution $x_*(\bar{p})$ of (3.10), with $p \equiv \bar{p}$, by executing the original algorithm.

Each of the two approaches mentioned above has advantages and disadvantages. It will depend on the given application problem which approach may be more convenient

in terms of computational efficiency.

J. C. Gilbert studied in [30] the application of automatic differentiation to fixed-point iterations of the form

$$x_{k+1} = \phi(x_k, p), \quad k = 0, 1, 2, \dots, \quad (3.14)$$

with $\phi : \mathbb{R}^{n+n_p} \rightarrow \mathbb{R}^n$. Under some smoothness assumptions on ϕ , he proved that for a given value $p = \bar{p}$ the convergence of the sequence of iterates $\{x_k(\bar{p})\}$ to $x_*(\bar{p})$ implies the convergence of the corresponding sequence of derivatives $\{\partial x_k(\bar{p})/\partial p\}$ to $\partial x_{*IG}/\partial p$. He showed that this property holds for the iterates generated by Newton's method. Theorem 3.2 was and Corollary 3.1 summarize these results. The proof of Theorem 3.2 can be found in [30]. Here we provide a proof for Corollary 3.1.

Theorem 3.2 (Gilbert [30])

Let $S_x \subset \mathbb{R}^n$ and $S_p \subset \mathbb{R}^{n_p}$ be open sets. Assume that

$\phi : S_x \times S_p \subset \mathbb{R}^{n+n_p} \rightarrow \mathbb{R}^n$ is continuously differentiable, and that ϕ' is Lipschitz continuous at the point $(x_, \bar{p}) \in S_x \times S_p$.*

If the following conditions hold:

- (i) the initial iterate x_0 in (3.14) is a differentiable function of p on S_p ;*
- (ii) for a fixed $\bar{p} \in S_p$, the sequence $\{x_k(\bar{p})\}$ defined by (3.14) converges to $x_* \in S_x$;*
- (iii) the spectral radius ρ of $\phi_x(x_*, \bar{p})$ (the partial derivative of ϕ with respect to x), satisfies*

$$\rho(\phi_x(x_*, \bar{p})) < \tau < 1, \quad \text{for some } \tau \in \mathbb{R};$$

then

- $\{x_k(\bar{p})\}$ converges to x_* q -linearly, i.e., there exists an index k_0 such that

$$\|x_{k+1}(\bar{p}) - x_*\| \leq \tau \|x_k(\bar{p}) - x_*\|, \quad \text{for all } k \geq k_0;$$

- $\{\partial x_k(\bar{p})/\partial p\}$ converges to $\partial x_{*IG}/\partial p$.

Corollary 3.1

If the mapping f given in (3.10) is twice continuously differentiable in $S_x \times S_p \subset \mathbb{R}^{n+n_p}$, and if there exists $x_* \in S_x$ and $\bar{p} \in S_p$ such that $f(x_*, \bar{p}) = 0$ and $f_x(x_*, \bar{p})^{-1}$ is nonsingular, then the iteration function defined by Newton's method for problem (3.10) satisfies the hypotheses of Theorem 3.2 in a neighborhood $\mathcal{N}_* \subset S_x$ of x_* .

Proof

The iteration function defined by Newton's method for problem (3.10) is given by

$$\phi(x, p) = x - f_x(x, p)^{-1} f(x, p), \quad (3.15)$$

Since f is twice continuously differentiable in $S_x \times S_p$, the function ϕ defined by (3.15) is continuously differentiable in $S_x \times S_p$.

Now, we will prove that ϕ' is Lipschitz continuous at x_* in a neighborhood $\mathcal{N}_* = \{x \in S_x : \|x - x_*\| < r\}$, for some $r \in \mathbb{R}$.

Differentiating ϕ with respect to x in (3.15), we obtain

$$\phi_x(x, p) = f_x(x, p)^{-1} f_{xx}(x, p) f_x(x, p)^{-1} f(x, p). \quad (3.16)$$

Hence, for all $x \in \mathcal{N}_*$ and $p \equiv \bar{p}$, we obtain

$$\|\phi_x(x, \bar{p}) - \phi_x(x_*, \bar{p})\| \leq \|f_x(x, \bar{p})^{-1} f_{xx}(x, \bar{p}) f_x(x, \bar{p})^{-1}\| \|f(x, \bar{p})\|, \quad (3.17)$$

since $\phi_x(x_*, \bar{p}) = 0$ because $f(x_*, \bar{p}) = 0$.

Since the product of derivatives in (3.17) is a continuous function and $\tilde{\mathcal{N}}_*$ (the closure of \mathcal{N}_*) is compact, then there exists a constant $M \in \mathbb{R}$ such that

$$\|f_x(x, \bar{p})^{-1} f_{xx}(x, \bar{p}) f_x(x, \bar{p})^{-1}\| \leq M, \quad \text{for all } x \in \mathcal{N}_*.$$

By the continuity of f' , there exists a constant $\gamma \in \mathbb{R}$, such that

$$\|f(x, \bar{p})\| = \|f(x, \bar{p}) - f(x_*, \bar{p})\| \leq \gamma \|x - x_*\|, \quad \text{for all } x \in \mathcal{N}_*.$$

Combining these two bounds in (3.17) we obtain

$$\begin{aligned} \|\phi_x(x, \bar{p}) - \phi_x(x_*, \bar{p})\| &\leq M \gamma \|x - x_*\| \\ &= \gamma_1 \|x - x_*\|, \end{aligned} \tag{3.18}$$

for all $x \in \mathcal{N}_*$, where $\gamma_1 \equiv M \gamma$. Therefore, $\phi_x(x, \bar{p})$ is Lipschitz continuous at x_* in \mathcal{N}_* .

On the other hand, differentiating ϕ with respect to p in (3.15), we obtain

$$\phi_p(x, p) = f_x(x, p)^{-1} f_{xp}(x, p) f_x(x, p)^{-1} f(x, p) - f_p(x, p). \tag{3.19}$$

Using the fact that $f(x_*, \bar{p}) = 0$ and applying the triangular inequality, we obtain that

$$\begin{aligned} \|\phi_x(x, \bar{p}) - \phi_x(x_*, \bar{p})\| &\leq \|f_x(x, \bar{p})^{-1} f_{xp}(x, \bar{p}) f_x(x, \bar{p})^{-1}\| \|f(x, \bar{p})\| + \\ &\quad \|f_p(x, \bar{p}) - f_p(x_*, \bar{p})\|, \end{aligned} \tag{3.20}$$

for all $x \in \mathcal{N}_*$.

Again, by compactness of $\tilde{\mathcal{N}}_*$, there exists a constant $M' \in \mathbb{R}$ such that

$$\|f_x(x, \bar{p})^{-1} f_{xp}(x, \bar{p}) f_x(x, \bar{p})^{-1}\| \leq M', \quad \text{for all } x \in \mathcal{N}_*.$$

By continuity of f' , we have that

$$\|f(x, \bar{p})\| = \|f(x, \bar{p}) - f(x_*, \bar{p})\| \leq \gamma \|x - x_*\|, \quad \text{for all } x \in \mathcal{N}_*.$$

Finally, since f is twice continuously differentiable, then f_p is Lipschitz continuous in \mathcal{N}_* , which means that there exists a constant γ' such that

$$\|f_p(x, \bar{p}) - f_p(x_*, \bar{p})\| \leq \gamma' \|x - x_*\|, \quad \text{for all } x \in \mathcal{N}_*.$$

Combining these three bounds in (3.20), we obtain

$$\begin{aligned} \|\phi_x(x, \bar{p}) - \phi_x(x_*, \bar{p})\| &\leq M' \gamma \|x - x_*\| + \gamma' \|x - x_*\| \\ &= \gamma_2 \|x - x_*\|, \quad \text{for all } x \in \mathcal{N}_*, \end{aligned} \quad (3.21)$$

where $\gamma_2 \equiv M' \gamma + \gamma'$. Therefore, $\phi_p(x, \bar{p})$ is Lipschitz continuous at x_* in \mathcal{N}_* .

From (3.18) and (3.21), it follows that ϕ' is Lipschitz continuous at x_* in \mathcal{N}_* .

Now, it is not difficult to see that Newton's method satisfies the hypotheses (i)-(iii) of Theorem 3.2.

First, notice that the initial iterate x_0 in Newton's method can be chosen as a differentiable function of p , for example the constant function $x_0 \equiv x_0(p)$.

Second, if x_0 "sufficiently close" to x_* , then we know that the sequence $\{x_k(\bar{p})\}$ generated by Newton's method is well-defined and converges to x_* q-quadratically.

Finally, evaluating $\phi_x(x, p)$ at $x \equiv x_*$ and $p \equiv \bar{p}$ in (3.16), we obtain

$$\phi_x(x_*, \bar{p}) = 0,$$

which implies

$$\rho(\phi_x(x_*, \bar{p})) = 0.$$

□

Thus, if f is twice continuously differentiable, the hypotheses of Theorem 3.2 are verified for Newton's method, and the sequence of derivatives $\{\partial x_k(\bar{p})/\partial p\}$ will converge to $\partial x_{\star IG}/\partial p$.

On the other hand, in Broyden's method the iterates are defined by the iteration

$$x_{k+1} = \phi_k(x_k, p) = x_k - A_k^{-1} f(x_k, p), \quad k = 0, 1, 2, \dots,$$

where A_k is updated by formula (3.5). The update of each A_k is a function of all the previous points x_k and of the initial choice A_0 , and this functional dependence is not necessarily smooth. It is well-known that even if the sequence $\{x_k(\bar{p})\}$ converges to x_\star , the updates A_k are not guaranteed to converge to $f_x(x_\star, \bar{p})$ or to any other limit (see Dennis and Schnabel [24]). Therefore, the iteration function given by Broyden's method does not satisfy the required smoothness hypothesis in Theorem 3.2, even if the function f is sufficiently smooth.

Even if Broyden's method does not fit in the class of iterative processes defined by Theorem 3.2, the results that will be presented in Chapter 4 show that the convergence property for the derivatives is verified in practice for this method, when the derivatives are computed by automatic differentiation.

Chapter 4

Numerical Results

The purpose of this chapter is to show the numerical results obtained by applying the automatic differentiation tool **ADIFOR** on two iterative algorithms that execute a generalized fixed-point iteration of the form

$$x_{k+1} = \phi_k(x_k, p) \equiv x_k - A_k^{-1} f(x_k, p), \quad k = 0, 1, 2, \dots$$

to calculate a solution $x \equiv x_*(p) \in \mathbb{R}^n$ of the system

$$f(x, p) = 0, \tag{4.1}$$

where $f : \mathbb{R}^{n+n_p} \rightarrow \mathbb{R}^n$ is a continuously differentiable mapping, and $p \in \mathbb{R}^{n_p}$ is a vector of parameters with fixed value \bar{p} . Basically, one of the algorithms is an implementation of Newton's method and the other is an implementation of Broyden's method.

For the given value $p = \bar{p}$, each of the “differentiated” codes produces upon execution a sequence of derivatives $\{\partial x_k(\bar{p})/\partial p\} \subset \mathbb{R}^{n \times n_p}$.

In Section 4.1 we will see for all the test problems considered that the generated sequence of derivatives converges to the correct value $z \equiv \partial x_{*IG}/\partial p$, a solution of the implicit gradient equation

$$f_x(x_*, \bar{p}) \cdot z = -f_p(x_*, \bar{p}) \tag{4.2}$$

corresponding to problem (4.1), for the given value $p \equiv \bar{p}$. We will analyze the results obtained by employing the “differentiated” algorithms.

In Section 4.2, we will show the results obtained by employing these “differentiated” algorithms in a code for solving parameter identification problems via the so-called Black-Box method. In this context, we will compare the performance of using automatic differentiation versus using forward finite-difference approximations.

4.1 Differentiating LMDER and HYBRJ via ADIFOR

In this section we will show an application of the automatic differentiation precompiler **ADIFOR** on two iterative algorithms for solving systems of nonlinear equations: **LMDER** and **HYBRJ**, obtained from the **MINPACK-1** software library (see More et al. [55], [56]).

LMDER implements a modification of the Levenberg-Marquardt algorithm, proposed by Moré in [54]. **HYBRJ** implements a modification of Powell’s hybrid method [61], and it uses Broyden’s update. Close to the solution of the system, **LMDER** is expected to behave like Newton’s method, and **HYBRJ** like Broyden’s method. In order to facilitate convergence from poor starting points, both algorithms implement a trust-region technique. **LMDER** uses a full trust-region approach and **HYBRJ** uses a dogleg step. For background material about these methods, the reader is referred to the book by Dennis and Schnabel [24]. A detailed description of the codes **LMDER** and **HYBRJ** is given in the technical report by Moré et al. [56].

In what follows, we will adopt the following naming convention. By “**LMDER.test**” and “**HYBRJ.test**”, we will denote the iterative solvers *before* the application of automatic differentiation. The only difference between these two codes and the “original” codes, **LMDER** and **HYBRJ** as provided in **MINPACK-1**, is that some set up was necessary in order to tailor these subroutines to our application. By “**LMDER.ad**” and “**HYBRJ.ad**”, we will denote the “differentiated” codes, obtained by applying **ADIFOR** to **LMDER.test** and **HYBRJ.test**, respectively.

The reader may relate the `.test` and `.ad` codes with the iterative algorithms \mathcal{A} and \mathcal{A}' , respectively, discussed in Chapter 3.

For the numerical tests, we considered six problems of the form (4.1). They were provided to us by Dr. Karen Williamson (Center for Research in Parallel Computation and Department of Computational and Applied Mathematics, Rice University), and they originated from the discretization of the systems of parameterized first-order ordinary differential equations given in Appendix A. For a complete discussion about the discretization scheme used the interested reader may see the work of Dennis et al. [23].

Two versions of problem 6 were implemented: the original one and an “extended” version, denoted with the suffix “(ext).”, for which the dimension n_p is increased. The generation of “extended” problems from the original ones is explained in Section 4.2.

We applied the precompiler **ADIFOR** to **LMDER.test** and **HYBRJ.test**, considering p as the vector of independent variables, and each iterate $x_k(p)$ generated by the algorithm as the vector of dependent variables. Upon execution, for an input $p \equiv \bar{p}$, each of the “differentiated” codes produced two sequences: the original sequence of iterates $\{x_k(\bar{p})\}$, and the corresponding sequence of derivatives $\{\partial x_k(\bar{p})/\partial p\}$. Our results show that, *whenever the sequence $\{x_k(\bar{p})\}$ converged to a solution $x_*(\bar{p})$ of (4.1), then $\{\partial x_k(\bar{p})/\partial p\}$ converged to $\partial x_*(\bar{p})/\partial p \equiv \partial x_{*IG}/\partial p$, the corresponding solution of (4.2), for all the test problems.*

Tables 4.1 – 4.7 show the convergence of the sequences $\{x_k(\bar{p})\}$ to $x_*(\bar{p})$, and $\{\partial x_k(\bar{p})/\partial p\}$ to $\partial x_{*IG}/\partial p$, for all the problems. The tests were performed on a SPARC Station 2.

In each table, the first column gives the current iteration k of the algorithm. The second column gives a measure of the function value at the current iterate $x_k(\bar{p})$.

The third column gives the relative error (RE) in $x_k(\bar{p})$ with respect to the solution $x_*(\bar{p})$ of (4.1), that is

$$\text{RE}(x_k, x_*) = \frac{\|x_k(\bar{p}) - x_*(\bar{p})\|_2}{\|x_*(\bar{p})\|_2},$$

where $\|\cdot\|_2$ denotes the Euclidean, or l_2 norm. Finally, the fourth column gives the relative error in the derivative $\partial x_k(\bar{p})/\partial p$ computed by automatic differentiation, with respect to the derivative $\partial x_{*IG}/\partial p$ computed by solving (4.2), that is

$$\text{RE}\left(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p}\right) = \frac{\|\partial x_k(\bar{p})/\partial p - \partial x_{*IG}/\partial p\|_F}{\|\partial x_{*IG}/\partial p\|_F},$$

where $\|\cdot\|_F$ denotes the Frobenius norm (see Dennis and Schnabel [24]).

The results show that the sequence of derivatives $\{\partial x_k(\bar{p})/\partial p\}$, computed by automatic differentiation, converged to the correct value $\partial x_{*IG}/\partial p$ for all the tested problems. Notice that in all the cases the derivatives converged slower than the iterates.

Table 4.1: **Problem 1** $n_p = 2$, $n = 40$, $\bar{p} = (10^{-6}, 10^{-6})^T$

<u>LMDER.ad</u>	iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
	1	$0.26 \times 10^{+02}$	$0.12 \times 10^{+01}$	$0.10 \times 10^{+01}$
	2	$0.22 \times 10^{+01}$	0.95×10^{-01}	0.36×10^{-01}
	3	0.18×10^{-01}	0.46×10^{-03}	0.26×10^{-03}
	4	0.63×10^{-06}	0.79×10^{-08}	0.18×10^{-07}
	5	0.72×10^{-13}	0.00	0.18×10^{-14}

<u>HYBRJ.ad</u>	iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
	1	$0.26 \times 10^{+02}$	$0.12 \times 10^{+01}$	$0.10 \times 10^{+01}$
	2	$0.22 \times 10^{+01}$	0.95×10^{-01}	0.36×10^{-01}
	3	$0.21 \times 10^{+00}$	0.48×10^{-02}	0.12×10^{-02}
	4	0.82×10^{-02}	0.23×10^{-03}	0.14×10^{-03}
	5	0.57×10^{-03}	0.11×10^{-04}	0.11×10^{-04}
	6	0.65×10^{-05}	0.12×10^{-06}	0.19×10^{-06}
	7	0.18×10^{-06}	0.30×10^{-08}	0.62×10^{-08}
	8	0.39×10^{-08}	0.79×10^{-10}	0.19×10^{-09}
	9	0.10×10^{-09}	0.00	0.53×10^{-11}

Table 4.2: **Problem 2** $n_p = 2$, $n = 80$, $\bar{p} = (3, 4)^T$

<u>LMDER.ad</u>	iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
	1	$0.12 \times 10^{+01}$	$0.75 \times 10^{+00}$	$0.10 \times 10^{+01}$
	2	0.74×10^{-14}	0.35×10^{-14}	0.27×10^{-14}
	3	0.25×10^{-14}	0.00	0.12×10^{-14}

<u>HYBRJ.ad</u>	iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
	1	$0.12 \times 10^{+01}$	$0.75 \times 10^{+00}$	$0.10 \times 10^{+01}$
	2	0.54×10^{-14}	0.00	0.31×10^{-14}

Table 4.3: Problem 3 $n_p = 3$, $n = 80$, $\bar{p} = (2, 4, 4)^T$

LMDER.ad

iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
...
7	0.38×10^{-01}	0.59×10^{-01}	$0.19 \times 10^{+00}$
8	0.30×10^{-01}	0.41×10^{-02}	0.23×10^{-01}
9	0.21×10^{-03}	0.20×10^{-04}	0.16×10^{-03}
10	0.57×10^{-08}	0.52×10^{-09}	0.57×10^{-08}
11	0.47×10^{-14}	0.00	0.12×10^{-14}

HYBRJ.ad

iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
...
16	0.31×10^{-02}	0.79×10^{-03}	0.25×10^{-01}
17	0.98×10^{-03}	0.25×10^{-03}	0.77×10^{-02}
18	0.27×10^{-03}	0.49×10^{-04}	0.15×10^{-02}
19	0.62×10^{-04}	0.14×10^{-04}	0.63×10^{-03}
20	0.24×10^{-04}	0.62×10^{-05}	0.22×10^{-03}
21	0.64×10^{-05}	0.14×10^{-05}	0.55×10^{-04}
22	0.11×10^{-05}	0.18×10^{-06}	0.12×10^{-04}
23	0.31×10^{-06}	0.36×10^{-07}	0.32×10^{-05}
24	0.12×10^{-06}	0.22×10^{-07}	0.11×10^{-05}
25	0.20×10^{-07}	0.46×10^{-08}	0.40×10^{-06}
26	0.33×10^{-08}	0.29×10^{-09}	0.27×10^{-07}
27	0.61×10^{-09}	0.00	0.69×10^{-08}

Table 4.4: Problem 4 $n_p = 3$, $n = 80$, $\bar{p} = (6, 4, 1)^T$

LMDER.ad

iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
1	$0.65 \times 10^{+00}$	$0.34 \times 10^{+00}$	$0.10 \times 10^{+01}$
2	0.96×10^{-01}	0.68×10^{-01}	$0.29 \times 10^{+00}$
3	0.39×10^{-02}	0.31×10^{-02}	0.21×10^{-01}
4	0.11×10^{-04}	0.68×10^{-05}	0.83×10^{-04}
5	0.75×10^{-10}	0.26×10^{-10}	0.60×10^{-09}
6	0.17×10^{-14}	0.00	0.11×10^{-14}

HYBRJ.ad

iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
1	$0.65 \times 10^{+00}$	$0.34 \times 10^{+00}$	$0.10 \times 10^{+01}$
2	0.96×10^{-01}	0.68×10^{-01}	$0.29 \times 10^{+00}$
3	0.26×10^{-01}	0.21×10^{-01}	$0.11 \times 10^{+00}$
4	0.31×10^{-02}	0.20×10^{-02}	0.14×10^{-01}
5	0.48×10^{-03}	0.26×10^{-03}	0.20×10^{-02}
6	0.32×10^{-04}	0.15×10^{-04}	0.13×10^{-03}
7	0.41×10^{-05}	0.22×10^{-05}	0.21×10^{-04}
8	0.26×10^{-06}	0.12×10^{-06}	0.13×10^{-05}
9	0.14×10^{-07}	0.58×10^{-08}	0.73×10^{-07}
10	0.77×10^{-09}	0.00	0.34×10^{-08}

Table 4.5: Problem 5 $n_p = 4$, $n = 80$, $\bar{p} = (10, 10, 30, 30)^T$

LMDER.ad

iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
1	$0.39 \times 10^{+01}$	$0.30 \times 10^{+00}$	$0.10 \times 10^{+01}$
2	0.54×10^{-14}	0.55×10^{-15}	0.20×10^{-14}
3	0.26×10^{-14}	0.00	0.59×10^{-15}

HYBRJ.ad

iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
1	$0.39 \times 10^{+01}$	$0.30 \times 10^{+00}$	$0.10 \times 10^{+01}$
2	0.43×10^{-14}	0.00	0.23×10^{-14}

Table 4.6: Problem 6 $n_p = 5$, $n = 200$, $\bar{p} = (0.58 \times 10^{-4}, 0.26 \times 10^{-4}, 0.16 \times 10^{-4}, 0.28 \times 10^{-3}, 0.46 \times 10^{-4})^T$

	iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
<u>LMDER.ad</u>	1	$0.14 \times 10^{+02}$	0.37×10^{-01}	$0.10 \times 10^{+01}$
	2	0.40×10^{-12}	0.78×10^{-15}	0.61×10^{-14}
	3	0.29×10^{-12}	0.00	0.21×10^{-14}

	iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
<u>HYBRJ.ad</u>	1	$0.14 \times 10^{+02}$	0.37×10^{-01}	$0.10 \times 10^{+01}$
	2	0.35×10^{-12}	0.00	0.10×10^{-13}

Table 4.7: Problem 6 (ext.) $n_p = 10$, $n = 195$, $\bar{p} = (0.58 \times 10^{-4}, 0.26 \times 10^{-4}, 0.16 \times 10^{-4}, 0.28 \times 10^{-3}, 0.46 \times 10^{-4}, 0.10 \times 10^{+3}, 0.0, 0.0, 0.0, 0.0)^T$

	iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
<u>LMDER.ad</u>	1	$0.14 \times 10^{+02}$	0.38×10^{-01}	$0.10 \times 10^{+01}$
	2	0.39×10^{-12}	0.95×10^{-15}	0.55×10^{-14}
	3	0.29×10^{-12}	0.00	0.19×10^{-14}

	iter. k	$\ f(x_k, \bar{p})\ _2$	$\text{RE}(x_k, x_*)$	$\text{RE}(\frac{\partial x_k}{\partial p}, \frac{\partial x_{*IG}}{\partial p})$
<u>HYBRJ.ad</u>	1	$0.14 \times 10^{+02}$	0.38×10^{-01}	$0.10 \times 10^{+01}$
	2	0.34×10^{-12}	0.00	0.20×10^{-13}

Notice that for problems 1, 3, and 4, the final derivative values obtained with **HYBRJ.ad** are not so accurate as the ones obtained with **LMDER.ad**. This is because **LMDER.ad** was able to get closer to the solution of the system than **HYBRJ.ad**, as can be observed from the values given in the second column of Tables 4.1 – 4.7.

Tables 4.8 and 4.9 summarize the results obtained for all the test problems. The first column gives the problem number. The second column gives the dimension n_p of the vector of parameters \bar{p} . Notice that the problems are ordered by increasing values of n_p . The third column gives the dimension n of the system.

The fourth column gives the number of iterations required by the algorithm to achieve convergence to a solution $x_*(\bar{p})$ of (4.1).

The fifth column gives the norm of the residual in the implicit gradient equation (4.2) for the last derivative value computed by automatic differentiation: $\partial x_{*AD}/\partial p$, i.e.,

$$\| \text{IG res.} \|_F = \| f_x(x_*, \bar{p}) \cdot \frac{\partial x_{*AD}}{\partial p} + f_p(x_*, \bar{p}) \|_F.$$

Here, “IG res.” stands for “Implicit Gradient residual”.

The sixth column gives an estimate of the relative error of $\partial x_{*AD}/\partial p$ in (4.2). This estimate is given by

$$\text{RE}\left(\frac{\partial x_{*AD}}{\partial p}\right) \simeq \frac{\| \text{IG res.} \|_F \cdot \mathcal{K}(f_x(x_*, \bar{p}))}{\| f_x(x_*, \bar{p}) \|_F \cdot \left\| \frac{\partial x_{*IG}}{\partial p} \right\|_F},$$

where $\mathcal{K}(f_x(x_*, \bar{p}))$ denotes the condition number of the Jacobian $f_x(x_*, \bar{p})$ (see Stewart [65]).

The seventh column gives the run-time ratio between the “differentiated” (.ad) algorithm and the corresponding “undifferentiated” (.test) one. This ratio indicates the overhead with respect to the original function evaluation involved in the derivative calculation.

Finally, the eighth column gives an estimate of the run-time ratio between calculating $\partial x_{*AD}/\partial p$, and approximating $\partial x_{*IG}/\partial p$ by forward finite-differences. In order to get this estimate, we assumed that the run time for computing the approximation is about $(n_p + 1)$ times the run time for evaluating the original function, which is the .test code.

Table 4.8: LMDER.ad - Final Results

prob.	n_p	n	iter.	$\ IG \text{ res.}\ _F$	$RE(\frac{\partial x_{*AD}}{\partial p})$.ad/.test	.ad/f.d.
1	2	40	5	4.77×10^{-08}	3.99×10^{-16}	6.26	2.09
2	2	80	3	4.83×10^{-16}	1.77×10^{-16}	7.32	2.44
3	3	80	11	2.98×10^{-15}	4.60×10^{-16}	9.55	2.39
4	3	80	6	1.97×10^{-16}	1.46×10^{-16}	9.21	2.30
5	4	80	3	1.71×10^{-16}	1.35×10^{-16}	10.89	2.18
6	5	200	3	4.29×10^{-09}	2.88×10^{-16}	12.74	2.12
6 (ext.)	10	195	3	4.05×10^{-09}	2.65×10^{-16}	21.65	2.16

Table 4.9: HYBRJ.ad - Final Results

prob.	n_p	n	iter.	$\ IG \text{ res.}\ _F$	$RE(\frac{\partial x_{*AD}}{\partial p})$.ad/.test	.ad/f.d.
1	2	40	9	2.26×10^{-04}	1.14×10^{-12}	7.77	2.59
2	2	80	2	1.06×10^{-15}	2.29×10^{-16}	7.49	2.52
3	3	80	27	2.85×10^{-08}	2.40×10^{-09}	11.04	2.76
4	3	80	10	9.69×10^{-10}	3.38×10^{-10}	11.05	2.76
5	4	80	2	4.02×10^{-16}	2.88×10^{-17}	12.67	2.53
6	5	200	2	1.04×10^{-08}	4.78×10^{-16}	14.39	2.40
6 (ext.)	10	195	2	1.06×10^{-08}	6.76×10^{-16}	23.65	2.36

From the above tables we can observe that, for all the test problems, the quantities $RE(\partial x_{*AD}/\partial p)$ computed by **LMDER.ad** are of the order of the machine precision. Therefore, the obtained derivative values are very accurate in this case. The derivatives generated by **HYBRJ.ad** also achieve high accuracy, except for problems 3 and

4. As mentioned earlier, this is related to the fact that **LMDER.ad** was able to get closer to the solution of the system than **HYBRJ.ad**.

Due to the small size of the relative errors in $\partial x_{*AD}/\partial p$ we can conclude that *automatic differentiation computed the correct derivative values, for all the test problems considered.*

On the other hand, the ratios displayed in the seventh and eighth columns of Tables 4.8 and 4.9 reveal that the cost involved in computing the derivatives by automatic differentiation can be significant.

Each value in the seventh column indicates that the “differentiated” (.ad) algorithm took about that number of times longer to execute than the corresponding “undifferentiated” (.test) code. This cost in the derivative calculation compared to the function evaluation is related to the fact that automatic differentiation propagates the derivatives over the *entire sequence* of elementary operations involved in the original algorithm. In the case of **LMDER.ad** and **HYBRJ.ad**, most of the overhead arises probably from the *matrix factorization*, which is “differentiated” in both cases. Notice also that the ratios displayed in this column increase when the number of independent variables n_p increases. This is related to the fact that the precompiler **ADIFOR** mainly implements the forward mode of automatic differentiation (it uses the reverse mode only to compute derivatives of expressions in the right-hand sides of assignment statements). Recall that the time required to compute derivatives in the forward mode is proportional to the the number of independent variables, as explained in Section 2.4.

Finally, the ratios in the eighth column estimate that computing the derivatives by **ADIFOR** is about 2.5 times slower than approximating them by forward finite-differences. These ratios keep more or less constant even if the number of independent variables n_p increases. The number 2.5 probably arises from the fact that, as a

consequence of the chain rule, the application of automatic differentiation approximately doubles the number of multiplications executed in the original code, (i.e., $v_i = v_{j_1} \times v_{j_2}$ implies $v'_i = v'_{j_1} \times v_{j_2} + v_{j_1} \times v'_{j_2}$).

Notice that the run-time ratio between approximating derivatives by central differences and approximating them by forward differences is about 2 (see Section 2.1), which is comparable to the run-time ratio of 2.5 between computing derivatives by **ADIFOR** and approximating them by forward finite differences.

Even though the approximation of derivatives by forward or central differences may be faster than using **ADIFOR**, the derivative values computed in this way are less accurate.

One alternative to improve the computational time required by the **.ad** codes would be to start them from an initial point x_0 that is a better approximation to the solution x_* of the system. This approximation could be obtained, for example, by a first run of the corresponding **.test** code. The iterations performed by the **.ad** algorithm, from the new initial guess, would aim only to achieve convergence in the derivative values. In this case, an adequate stopping criterion related to the derivatives may be employed. This possibility is discussed by Griewank et al. in ([7]).

4.2 Application to the Solution of Parameter Identification Problems

In this section, we will show that the subroutines **LMDER.ad** and **HYBRJ.ad** tested in the previous section can be successfully employed in the context of solving parameter identification problems by the Black-Box method. We will compare the results obtained by using automatic differentiation versus employing forward finite-difference approximations.

The code used was developed by Dr. J. Dennis, Dr. K. Williamson and Dr. G. Li (Center for Research in Parallel Computation and Department of Computational and Applied Mathematics, Rice University) for solving parameter identification problems. We will refer to the particular code that we employed for our numerical experiments as the “**PID**” code.

The algorithm implemented in **PID** has been designed to solve parameter identification problems of the form

$$\begin{aligned} & \underset{p}{\text{minimize}} && g(p) \equiv \frac{1}{2} R(p)^T R(p) \\ & \text{subject to} && y' = F(t, y; p) \ , \quad y(t_0; p) = y_0 \ , \end{aligned} \tag{4.3}$$

where $g : \mathbb{R}^{n_p} \rightarrow \mathbb{R}$ is the objective function, $p \in \mathbb{R}^{n_p}$ is a vector of parameters to be identified, and

$$y' \equiv \frac{\partial y}{\partial t} = F(t, y; p)$$

is a system of parameterized first-order ordinary differential equations, or ODE’s. Here $t \in \mathbb{R}$ is the independent variable, usually thought of as time, and $y(t; p) \in \mathbb{R}^{n_y}$ denotes the solution of the system, which must satisfy the initial condition $y(t_0; p) = y_0$. The residual $R : \mathbb{R}^{n_p} \rightarrow \mathbb{R}^{n_r}$ gives a measure of how the solution $y(t; p)$ fits some given data points. The goal is to find a vector of parameters p_\star such that $y(t; p_\star)$ “best fits” the given data. Both F and R satisfy some smoothness assumptions. A detailed description of this problem is given in the paper by Dennis et al. [23]. Here, we present only the general ideas, since we are mainly concerned with the implementation aspects.

Notice that the minimization problem in (4.3) has a nonlinear least-squares structure. The first and second derivatives of the objective function $g(p)$ are given by

$$\nabla g(p) = J(p)^T R(p),$$

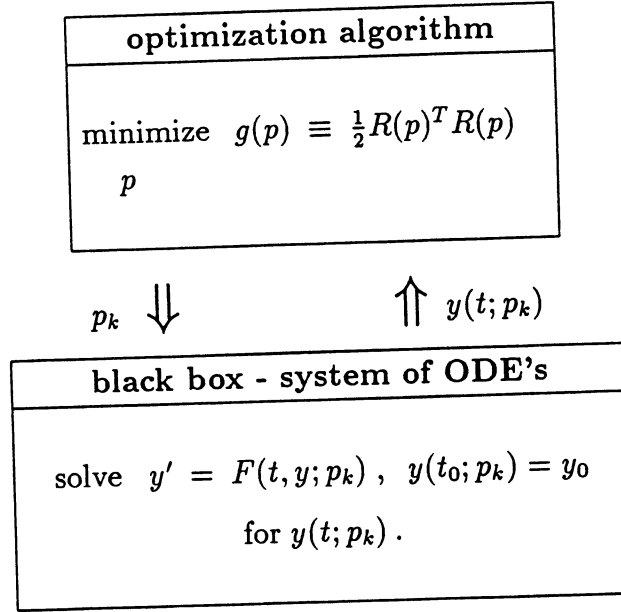
and

$$\nabla^2 g(p) = J(p)^T J(p) + \sum_{i=1}^{n_r} R_i(p) \cdot \nabla^2 R_i(p),$$

where $J(p) \in \mathbb{R}^{n_r \times n_p}$ is the Jacobian of $R(p)$. The i^{th} component of $R(p)$ is the scalar function $R_i(p)$, with $R_i : \mathbb{R}^{n_p} \rightarrow \mathbb{R}$ ($1 \leq i \leq n_r$). $\nabla^2 R_i(p)$ denotes the matrix of second derivatives of $R_i(p)$.

In order to solve problems of the form (4.3), **PID** employs a *Black-Box* approach. The name is derived from the fact that the optimization algorithm that minimizes the objective function $g(p)$ treats the numerical solution of the system of ODE's as a "black box". Basically, the procedure is the following. Starting from an initial approximation p_0 to a minimizer p_* of g , the optimization algorithm generates a sequence of iterates $\{p_k\}$ that is intended to converge to p_* . At each iteration k , the current iterate p_k is passed as input to the "black box" which returns to the optimizer the computed values for the solution $y(t; p_k)$ needed for the evaluation of the residual $R(p_k)$, and possibly its Jacobian $J(p_k)$ (both depend on $y(t; p_k)$). Then, the optimizer either computes a new iterate p_{k+1} and repeats the above steps, or exits if the stopping criterion is satisfied. The following diagram describes this interaction between the optimizer and the "black box".

Black Box approach



The code **PID** employed in our tests has this structure. The optimization part is executed by the subroutine **NL2SOL**, given by Dennis et al. in [21] and [22]. In the “black box” of **PID**, the system of ODE's is discretized using a *collocation* scheme. A detailed description of this scheme is given by Dennis et al. in [23].

For each iterate p_k given by **NL2SOL**, the system of ODE's

$$y' = F(t, y; p_k) , \quad y(t_0; p_k) = y_0 \quad (4.4)$$

is discretized by approximating the solution $y(t; p_k)$ with a piecewise polynomial function whose coefficients are unknown. Replacing $y(t; p_k)$ by this approximation transforms the system of ODE's into a system of parameterized nonlinear equations of the form

$$f(x, p_k) = 0 , \quad (4.5)$$

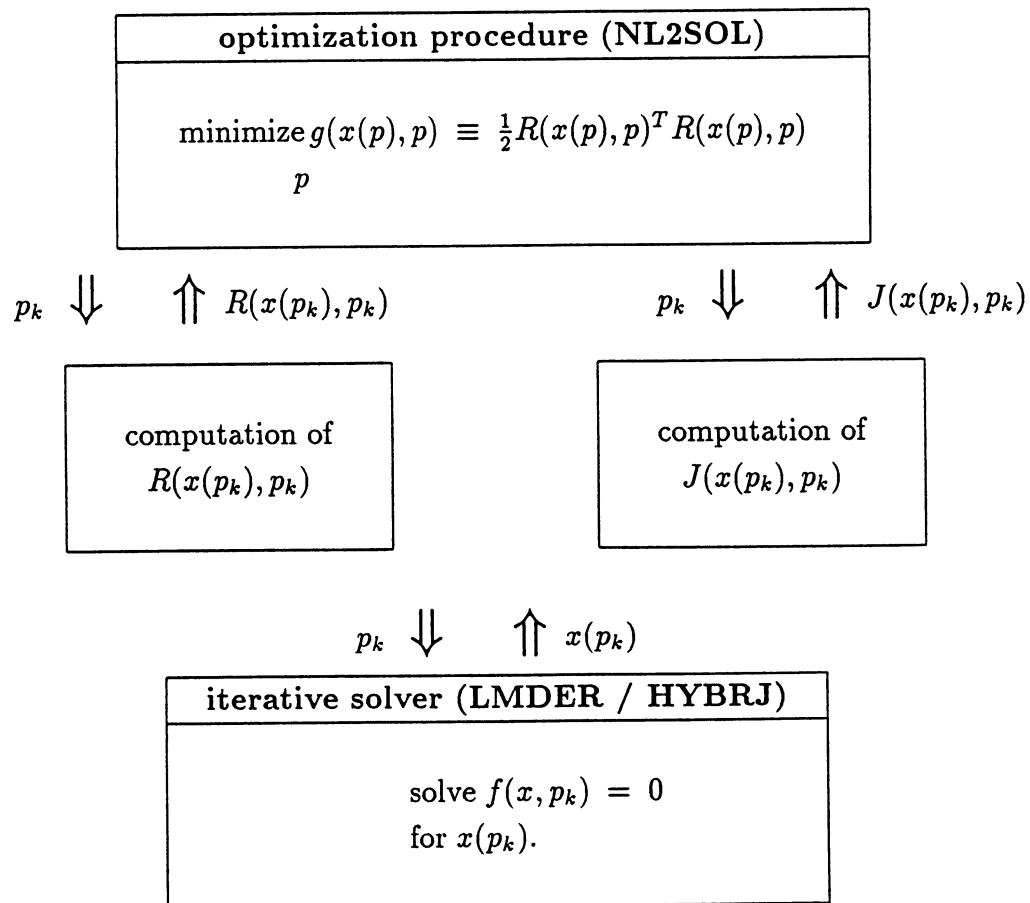
where $f : \mathbb{R}^{n+n_p} \rightarrow \mathbb{R}^n$, $p_k \in \mathbb{R}^{n_p}$ is fixed, and the unknowns $x \in \mathbb{R}^n$ are the variables that arise from the discretization of (4.4). Notice that this system is of the

form (4.1), considered in the previous section. Therefore, it can be solved using the iterative algorithms **LMDER.test** or **HYBRJ.test**, as before.

Assuming that the solution of (4.5) exists, it is given by the implicit function $x \equiv x(p_k) \in \mathbb{R}^n$, which provides the desired coefficients for the piecewise polynomial approximation to $y(t; p_k)$. This approximation replaces $y(t; p_k)$ when the residual $R(p_k)$ or its Jacobian $J(p_k)$ are evaluated in the optimization algorithm. Since both R and J depend on $y(t; p_k)$ and therefore on $x(p_k)$, we can denote $R(p_k) \equiv R(x(p_k), p_k)$ and $J(p_k) \equiv J(x(p_k), p_k)$.

The interaction between the optimizer **NL2SOL** and the iterative solver (**LMDER.test** or **HYBRJ.test**) in the code **PID** can be visualized as follows

PID code



Now we will present our numerical results. Four different versions of the parameter identification code **PID** were implemented and tested. All of them employ **NL2SOL** as the optimization algorithm. Basically, the four codes differ in the iterative solver employed to compute the solution $x(p_k)$ of (4.5) and in the way the Jacobian $J(x(p_k), p_k)$ is evaluated. We point out that **NL2SOL** does not require the evaluation of the Jacobian every time that the residual R is evaluated. The following two approaches were implemented to compute $J(x(p_k), p_k)$.

1. Forward finite differences.

Approximating $J(x(p_k), p_k)$ column by column using forward finite-differences requires the evaluation of the residual R at n_p points of the form $p_k + \Delta_j$ ($1 \leq j \leq n_p$), where Δ_j denotes the finite-difference step used for the j^{th} column of the Jacobian. Notice that each evaluation of $R(x, p_k + \Delta_j)$ involves the solution of the system $f(x, p_k + \Delta_j) = 0$, to compute $x \equiv x(p_k + \Delta_j)$.

2. Automatic differentiation.

Notice that the chain rule applied on $R(x(p), p)$ gives

$$J(x(p), p) = \frac{\partial R(x(p), p)}{\partial p} + \frac{\partial R(x(p), p)}{\partial x} \cdot \frac{\partial x(p)}{\partial p}. \quad (4.6)$$

Evaluating $J(x(p_k), p_k)$ by formula (4.6) requires the computation of $x(p_k)$ by solving system (4.5), and the computation of the derivative $\partial x(p_k)/\partial p$, which can be done by applying automatic differentiation on the iterative algorithm employed to solve (4.5) (using the same approach as in Section 4.1). We assume that the partial derivatives $\partial R/\partial p$ and $\partial R/\partial x$ are available, which is the case of all the test problems considered.

Here, will use the same notation as in the previous section. By **LMDER.test** and **HYBRJ.test** we will denote the iterative solvers *before* the application of automatic

differentiation. In the tests, these solvers were used to compute the solution $x(p_k)$ of the system (4.5). By **LMDER.ad** and **HYBRJ.ad**, we will denote the corresponding “differentiated” (by **ADIFOR**) codes, which were employed to compute the derivative $\partial x(p_k)/\partial p$.

We will refer to the four implemented versions of the code **PID** by **PID-L.fd**, **PID-L.ad**, **PID-H.fd**, and **PID-H.ad**. Here, the “L” and “H” after “PID” stand for “LMDER” and “HYBRJ”, respectively. The terminations “.fd” and “.ad” stand for finite differences and automatic differentiation, respectively. The differences between these four implementations of **PID** can be summarized as follows.

PID-L.fd: $x(p_k)$ computed by applying **LMDER.test** on (4.5), and $J(x(p_k), p_k)$ approximated by forward finite-differences.

PID-H.fd: $x(p_k)$ computed by applying **HYBRJ.test** on (4.5), and $J(x(p_k), p_k)$ approximated by forward finite-differences.

PID-L.ad: $x(p_k)$ computed by applying **LMDER.test** on (4.5), and $J(x(p_k), p_k)$ evaluated as in (4.6), with $\partial x(p_k)/\partial p$ computed by **LMDER.ad**.

PID-H.ad: $x(p_k)$ computed by applying **HYBRJ.test** on (4.5), and $J(x(p_k), p_k)$ evaluated as in (4.6), with $\partial x(p_k)/\partial p$ computed by **HYBRJ.ad**.

Tables 4.10 – 4.21 display the results obtained by employing these four codes to solve the set of parameter identification problems given in Appendix A. For each test problem, two cases were considered: (1) the vector of initial values $y_0 \in \mathbb{R}^{n_y}$ was fixed during the solution of the parameter identification problem and (2) the initial values were treated as variables and included as parameters in the problem. In this case, the dimension of the vector of parameters p was increased to $n_p + n_y$ to include the initial values (see Dennis et al. [23]). These “extended” problems are identified in the tables with the suffix “(ext.)”.

In each table, n_p denotes the dimension of the vector of parameters p , n denotes the dimension of the system (4.5), and p_0 denotes the starting point for NL2SOL.

The following information is displayed. The first column gives the name of the tested code. The second column gives the total number of iterations executed by NL2SOL. The third column gives the total execution time (in seconds). The fourth column shows the termination message returned by NL2SOL. "Y" denotes that NL2SOL converged to the solution p_* of the parameter identification problem, while "N" denotes that NL2SOL failed to converge. The fifth column gives the run-time ratio between the parameter identification code that employed automatic differentiation to compute the Jacobian and the corresponding code that employed forward finite-differences.

Finally, table 4.22 summarizes the results for all the test problems. Notice that the codes that employed automatic differentiation (PID-L.ad and PID-H.ad) took about *three* times longer to execute than the codes that used forward finite differences (PID-L.fd and PID-H.fd), upon successful termination.

For all the problems, PID-L.ad and PID-L.fd *achieved convergence*. On the other hand, PID-H.fd *failed to converge* in three cases: problems 1, 1 (ext.), and 5 (ext.), for which PID-H.ad *succeeded*.

For problems 1 and 1 (ext.), the failure of PID-H.fd occurred because at a given iteration k the finite-difference approximation to the Jacobian $J(x, p_k)$ could not be computed (HYBRJ.test was not able to calculate the solution of one of the corresponding systems of nonlinear equations). In the case of problem 5 (ext.), PID-H.fd failed because the optimization algorithm NL2SOL stopped with a termination message of "false convergence", which means that the generated sequence of iterates $\{p_k\}$ was converging to a noncritical point of the objective function g .

Therefore, *for all the problems in which PID-H.fd failed, PID-H.ad terminated successfully.* The failures in **PID-H.fd** originated from the inaccuracies involved in the finite-difference approximations to the Jacobian.

Thus, in return for the overhead in run time, **PID-H.ad** seems to be more robust than **PID-H.fd**.

Table 4.10: Problem 1 $n_p = 2$, $n = 40$, $p_0 = (10^{-4}, 10^{-4})^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	12	13.20	Y	2.74
PID-L.ad	12	36.15	Y	
PID-H.fd	2	6.04	N	—
PID-H.ad	12	17.36	Y	

Table 4.11: Problem 1 (ext.) $n_p = 3$, $n = 39$, $p_0 = (10^{-4}, 10^{-4}, 0)^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	11	51.31	Y	1.66
PID-L.ad	10	84.99	Y	
PID-H.fd	2	5.04	N	—
PID-H.ad	10	16.56	Y	

Table 4.12: Problem 2 $n_p = 2$, $n = 80$, $p_0 = (3, 4)^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	10	15.01	Y	2.39
PID-L.ad	9	35.95	Y	
PID-H.fd	9	11.34	Y	2.61
PID-H.ad	9	29.63	Y	

Table 4.13: Problem 2 (ext.) $n_p = 4$, $n = 78$, $p_0 = (3, 4, 1, 0)^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	9	20.55	Y	2.13
PID-L.ad	9	43.87	Y	
PID-H.fd	10	19.70	Y	2.21
PID-H.ad	9	43.59	Y	

Table 4.14: Problem 3 $n_p = 3$, $n = 80$, $p_0 = (2, 4, 4)^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	8	24.70	Y	5.23
PID-L.ad	8	129.29	Y	
PID-H.fd	8	16.29	Y	4.35
PID-H.ad	8	70.92	Y	

Table 4.15: Problem 3 (ext.) $n_p = 5$, $n = 78$, $p_0 = (2, 4, 4, 1, 0.3)^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	8	33.55	Y	5.58
PID-L.ad	8	187.13	Y	
PID-H.fd	8	22.32	Y	4.27
PID-H.ad	8	95.34	Y	

Table 4.16: Problem 4 $n_p = 3$, $n = 80$, $p_0 = (6, 4, 1)^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	6	15.82	Y	
PID-L.ad	6	62.47	Y	3.95
PID-H.fd	7	12.53	Y	
PID-H.ad	7	39.36	Y	3.14

Table 4.17: Problem 4 (ext.) $n_p = 5$, $n = 78$, $p_0 = (6, 4, 1, 1, 0)^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	6	20.79	Y	
PID-L.ad	6	80.08	Y	3.85
PID-H.fd	7	16.71	Y	
PID-H.ad	7	49.72	Y	2.97

Table 4.18: Problem 5 $n_p = 4$, $n = 80$, $p_0 = (10, 10, 30, 30)^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	7	18.77	Y	
PID-L.ad	7	42.93	Y	2.29
PID-H.fd	7	16.79	Y	
PID-H.ad	7	38.98	Y	2.32

Table 4.19: Problem 5 (ext.) $n_p = 6$, $n = 78$, $p_0 = (10, 10, 30, 30, 1, 0)^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	11	36.13	Y	
PID-L.ad	7	54.09	Y	1.50
PID-H.fd	9	34.78	N	
PID-H.ad	7	49.10	Y	—

Table 4.20: Problem 6 $n_p = 5, n = 200, p_0 = (0.58 \times 10^{-4}, 0.26 \times 10^{-4}, 0.16 \times 10^{-4}, 0.28 \times 10^{-3}, 0.46 \times 10^{-4})^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	6	263.22	Y	2.31
PID-L.ad	6	609.50	Y	
PID-H.fd	6	230.91	Y	2.47
PID-H.ad	6	571.22	Y	

Table 4.21: Problem 6 (ext.) $n_p = 10, n = 195, p_0 = (0.58 \times 10^{-4}, 0.26 \times 10^{-4}, 0.16 \times 10^{-4}, 0.28 \times 10^{-3}, 0.46 \times 10^{-4}, 100, 0, 0, 0, 0)^T$

code	iter.	run time	conv.	time ratio
PID-L.fd	6	449.17	Y	2.08
PID-L.ad	6	932.60	Y	
PID-H.fd	6	392.43	Y	2.25
PID-H.ad	6	883.76	Y	

Table 4.22: Time ratios

problem	np	n	PID-L.ad vs. PID-L.fd	PID-H.ad vs. PID-H.fd
1	2	40	2.74	PID-H.fd failed. PID-H.ad conv.
1 (ext.)	3	39	1.66	PID-H.fd failed. PID-H.ad conv.
2	2	80	2.39	2.61
2 (ext.)	4	78	2.13	2.21
3	3	80	5.23	4.35
3 (ext.)	5	78	5.58	4.27
4	3	80	3.95	3.14
4 (ext.)	5	78	3.85	2.97
5	4	80	2.29	2.32
5 (ext.)	6	78	1.50	PID-H.fd failed. PID-H.ad conv.
6	5	200	2.31	2.47
6 (ext.)	10	195	2.08	2.25

Chapter 5

Concluding Remarks

In this work, we presented a survey of the theory and implementation of automatic differentiation, and we showed an application of this technique in the context of solving systems of parameterized nonlinear equations, and parameter identification problems. Our application was particularly interesting because the two “differentiated” computer programs were implementations of Newton’s method (**LMDER**) and Broyden’s method (**HYBRJ**), respectively. The automatic differentiation software used in our experiments was the FORTRAN77 precompiler **ADIFOR**.

First, we tested the “differentiated” codes on a set of problems that are systems of parameterized nonlinear equations that originated in the discretization of systems of parameterized first-order ordinary differential equations. Automatic differentiation was employed to compute the derivatives of the iterates generated by the original algorithm (i.e., Newton’s method or Broyden’s method) with respect to the parameters.

We saw that whenever the sequence of iterates generated by the original algorithm converged, then the corresponding sequence of derivatives, computed by automatic differentiation, also converged, and it converged to the correct value, for all the test problems considered.

Even though the theory developed so far supports the convergence of the derivatives only for Newton’s method and not for Broyden’s method, our results indicate that the derivatives achieved convergence in both cases. Therefore, we conjecture that there is a more general class of iterative processes for which this property holds, and that this class includes Broyden’s method, and maybe other secant methods as well. The proof or refutation of this conjecture provides a field of future research.

In the context of this application, we estimated that automatic differentiation takes about 2.5 times longer to compute derivatives than approximating them by forward finite-differences, and this ratio does not depend on the number of independent variables. However, the use of finite differences would probably involve some inaccuracies in the computed derivative values.

Finally, we showed that the “differentiated” versions of Newton’s method and Broyden’s method were successfully employed in the the solution of parameter identification problems via the so-called Black-Box method.

We compared the performance of two parameter identification codes that computed derivatives by automatic differentiation versus similar codes that employed forward finite-difference approximations. The codes that employed automatic differentiation took longer to execute but were more robust than the other ones. More specifically, the parameter identification code that employed Newton’s method produced correct results using automatic differentiation and finite differences. The code that employed Broyden’s method succeeded for all the problems using automatic differentiation but failed in three of the problems using finite differences. Of course, the price paid for the robustness offered by automatic differentiation was a considerable increase in the execution time.

Appendix A

Test Problems

Problem 1: Bellman's Problem

(Bellman et al. [3])

$$y_1' = p_1(126.2 - y_1)(91.9 - y_1)^2 - p_2 y_1^2.$$

t	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0
y_1	0.0	1.4	6.3	10.4	14.2	17.6	21.4	23.0
t	10.0	12.0	15.0	20.0	25.0	30.0	40.0	
y_1	27.0	30.4	34.4	38.8	41.6	43.5	45.3	

Table A.1: Data for Problem 1

Problem 2: First-order Irreversible Chain Reaction

(Tjoa and Biegler [5])

$$y_1' = -p_1 y_1$$

$$y_2' = p_1 y_1 - p_2 y_2$$

t	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
y_1	1.0	.606	.368	.223	.135	.082	.050	.030	.018	.011	.007
y_2	0.0	.373	.564	.647	.669	.656	.624	.583	.539	.494	.451

Table A.2: Data for Problem 2

Problem 3: Barnes' Problem

(Van Doomselaar and Hemker [66])

$$y_1' = p_1 y_1 - p_2 y_1 y_2$$

$$y_2' = p_2 y_1 y_2 - p_3 y_2$$

t	0.0	.50	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0
y_1	1.0	1.1	1.3	1.1	.90	.70	.50	.60	.70	.80	1.0
y_2	.30	.35	.40	.50	.50	.40	.30	.25	.25	.30	.35

Table A.3: Data for Problem 3**Problem 4: Catalytic Cracking of Gasoil (Tjoa and Biegler [5])**

$$y_1' = -(p_1 + p_3)y_1^2$$

$$y_2' = p_1 y_1^2 - p_2 y_2$$

t	0.0	.025	.050	.075	.100	.125	.150	.175	.200	.225	.250
y_1	1.0	.741	.588	.488	.417	.364	.323	.290	.263	.241	.222
y_2	0.0	.199	.281	.307	.307	.292	.271	.247	.223	.200	.178
t	.300	.350	.400	.450	.500	.550	.650	.750	.850	.950	
y_1	.192	.169	.151	.137	.125	.115	.099	.087	.078	.070	
y_2	.140	.110	.086	.068	.054	.043	.029	.020	.014	.011	

Table A.4: Data for Problem 4

Problem 5: First-order Reversible Chain Reaction

(Tjoa and Biegler [5])

$$y_1' = -p_1 y_1 + p_2 y_2$$

$$y_2' = p_1 y_1 - (p_2 + p_3) y_2 + p_4 (y_1(t_0) - y_1 - y_2)$$

t	0.0	.05	.10	.15	.20	.25	.30	.35	.40	.45	.50
y_1	1.0	.824	.685	.575	.487	.417	.361	.316	.281	.253	.230
y_2	0.0	.094	.135	.165	.190	.209	.225	.237	.247	.255	.261

t	.55	.60	.65	.70	.75	.80	.85	.90	.95	1.0	
y_1	.213	.198	.187	.178	.171	.165	.161	.157	.154	.152	
y_2	.266	.270	.273	.276	.278	.279	.281	.282	.283	.283	

Table A.5: Data for Problem 5

Problem 6: Thermal Isomerization of α -Pinene

(Tjoa and Biegler [5])

$$y_1' = -(p_1 + p_2) y_1$$

$$y_2' = p_1 y_1$$

$$y_3' = p_2 y_1 - (p_3 + p_4) y_3 + p_5 y_5$$

$$y_4' = p_3 y_3$$

$$y_5' = p_4 y_3 - p_5 y_5$$

t	0.0	1230.	3060.	4920.	7800.	10680.	15030.	22620.	36420.
y_1	100.0	88.35	76.4	65.1	50.4	37.5	25.9	14.0	4.5
y_2	0.0	7.3	15.6	23.1	32.9	42.7	49.1	57.4	63.1
y_3	0.0	2.3	4.5	5.3	6.0	6.0	5.9	5.1	3.8
y_4	0.0	0.4	0.7	1.1	1.5	1.9	2.2	2.6	2.9
y_5	0.0	1.75	2.8	5.8	9.3	12.0	17.0	21.0	25.7

Table A.6: Data for Problem 6

Bibliography

- [1] F. L. Bauer. Computational graphs and rounding errors. *SIAM Journal on Numerical Analysis*, 11:87 – 96, 1974.
- [2] L. M. Beda, L. N. Korolev, N. V. Sukkikh, and T. S. Frolova. Programs for automatic differentiation for the machine BESM. Technical Report, Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, USSR, 1959. (In Russian).
- [3] R. Bellman, J. Jacquez, R. Kalaba, and S. Schwimmer. Quasilinearization and the estimation of chemical rate constants from raw kinetic data. *Math. Biosci.*, 1:71–76, 1967.
- [4] M. Berz. Forward algorithms for high orders and many variables with application to beam physics. In A. Griewank and . F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.
- [5] L. T. Biegler and B. Tjoa. Simultaneous solution and optimization strategies for parameter estimation of differential-algebraic equation systems. *Ind. Eng. Chem. Res.*, 30, 1991.
- [6] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1), 1972.
- [7] C. Bischof, G. Corliss, A. Griewank, and K. Williamson. Automatic differentiation of iterative nonlinear solvers. In preparation.
- [8] C. Bischof, A. Griewank, and D. Juedes. Exploiting parallelism in automatic differentiation. Preprint MCS-P204-0191, Mathematics and Computer Sciences Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4801, 1991.

- [9] C. Bischof and J. Hu. Utilities for building and optimizing a computational graph for algorithmic decomposition. Technical Memorandum ANL/MCS-TM-148, Mathematics and Computer Sciences Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4801, April 1991.
- [10] Christian Bischof. Issues in parallel automatic differentiation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.
- [11] Christian Bischof, George Corliss, and Andreas Griewank. ADIFOR exception handling. Technical Memorandum ANL/MCS-TM-159, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439-4801, 1991.
- [12] D. G. Cacuci. Sensitivity theory for nonlinear systems. I. Nonlinear functional analysis approach. *J. Math. Phys.*, 22(12):2794 – 2802, 1981.
- [13] D. G. Cacuci. Sensitivity theory for nonlinear systems. II. Extension to additional classes of responses. *J. Math. Phys.*, 22(12):2803 – 2812, 1981.
- [14] S. L. Campbell, E. Moore, R. T. Hood, and Z. Yangchun. Utilization of automatic differentiation in control algorithms.
- [15] B. W. Char. Computer algebra as a toolbox for program generation and manipulation. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.
- [16] B. W. Char, G. J. Fee, K. O. Geddes, G. H. Gonnet, and M. B. Monagan. A tutorial introduction to MAPLE. *Journal of Symbolic Computation*, 2(2):179 – 200, 1986.
- [17] T. F. Coleman, B. S. Garbow, and J. J. Moré. Software for estimating sparse Jacobian matrices. *ACM Trans. Math. Software*, 10:329 – 345, 1984.
- [18] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20:187 – 209, 1984.

- [19] S. D. Conte and C. de Boor. *Elementary Numerical Analysis*. McGraw-Hill, New York, 1980.
- [20] A. Curtis, M.J.D Powell, and J.K. Reid. On the estimation of sparse Jacobian matrices. *IMA Journal of Applied Mathematics*, 13:117 – 120, 1974.
- [21] J. E. Dennis, D. Gay, and R. E. Welsch. Algorithm 573. NL2SOL — An adaptive nonlinear least-squares algorithm. *ACM Trans. Math. Software*, 7:369 – 383, 1981.
- [22] J. E. Dennis, D. Gay, and R. E. Welsch. An adaptive nonlinear least-squares algorithm. *ACM Trans. Math. Software*, 7:348 – 368, 1981.
- [23] J. E. Dennis, G. Li, and K. A. Williamson. Optimization algorithms for parameter identification. In preparation, Rice University, Dept. of Computational and Applied Mathematics, 1992.
- [24] J. E. Dennis and R. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, NJ, 1983.
- [25] L. C. W. Dixon. Automatic differentiation and parallel processing in optimisation. Technical Report No. 180, The Numerical Optimisation Center, Hatfield Polytechnic, Hatfield, UK, 1987.
- [26] L. C. W. Dixon and M. Mohseninia. The use of the extended operations set of Ada with automatic differentiation and the truncated Newton method. Technical Report NOC TR176, The Numerical Optimisation Center, Hatfield Polytechnic, Hatfield, UK, April 1987.
- [27] H. Fischer. Automatic differentiation: How to compute the Hessian matrix. Report No. 26, Schwerpunktprogramm der Deutschen Forschungsgemeinschaft Anwendunsbezogene Optimierung und Steuerung, 1987.
- [28] H. Fischer. Automatic differentiation: Parallel computation of function, gradient and Hessian matrix. *Parallel Computing*, 13:101 – 110, 1990.
- [29] H. Fischer. Special problems in automatic differentiation. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.

- [30] J. C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1(1):13 – 21, 1992.
- [31] V. V. Goldman, J. Molenkamp, and J. A. van Hulzen. Efficient numerical program generation and computer algebra environments. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.
- [32] A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83 – 108. Kluwer Academic Publishers, 1989.
- [33] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. Preprint MCS-P228-0491, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439-4801, 1991.
- [34] A. Griewank and S. Rees. On the calculation of Jacobian matrices by the Markowitz rule. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.
- [35] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.
- [36] Andreas Griewank, D. Juedes, and J. Srinivasan. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. Preprint MCS-P180-1190, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439-4801, 1990.
- [37] A. C. Hearn. *REDUCE User's Manual, Version 3.3*. The Rand Corporation, Santa Monica, CA, 1987.
- [38] K. E. Hillstrom. User's guide for JAKEF. Technical Memorandum ANL/MCS-TM-16, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4801, 1985.

- [39] J. E. Horwedel, R. J. Raridon, and R. Q. Wright. Sensitivity analysis of AIRDOS-EPA using ADGEN with matrix reduction algorithms. Technical Memorandum ORNL/TM 11373, Martin Marietta Energy Systems, Inc., Oak Ridge National Laboratory, Oak Ridge, TN 37830, 1989.
- [40] J. E. Horwedel, B. A. Worley, E. M. Oblow, and F. G. Pin. GRESS version 1.0 users manual. Technical Memorandum ORNL/TM 10835, Martin Marietta Energy Systems, Inc., Oak Ridge National Laboratory, Oak Ridge, TN 37830, 1988.
- [41] J. E. Horwedel, R. Q. Wright, and R. E. Maerker. Sensitivity analysis of EQ3. Technical Memorandum ORNL/TM 11407, Oak Ridge National Laboratory, Oak Ridge, TN 37830, 1990.
- [42] M. Iri. Simultaneous computation of functions, partial derivatives and estimates of rounding errors — Complexity and practicality. *Japan Journal of Applied Mathematics*, 1(2):223 – 252, 1984.
- [43] M. Iri. History of automatic differentiation and error estimation. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.
- [44] M. Iri and K. Kubota. Methods of fast automatic differentiation and applications. Research Memorandum RMI 87 – 02, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, 1987.
- [45] M. Iri and K. Kubota. Padre2, version 1 – user’s manual. Research Memorandum RMI 90 – 01, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, 1990.
- [46] M. Iri, T. Tsuchiya, and M. Hoshi. Automatic computation of partial derivatives and rounding error estimates with applications to large-scale systems of nonlinear equations. *Journal of Computational and Applied Mathematics*, 24:365 – 392, 1988. Original Japanese version appeared in *Journal of Information Processing*, 26 (1985), pp. 1411 – 1420.
- [47] D. Juedes. A taxonomy of automatic differentiation tools. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.

- [48] H. Kagiwada, R. Kalaba, N. Rasakhoo, and Spingarn K. *Numerical Derivatives and Nonlinear Analysis*, volume 31 of *Mathematical Concepts and Methods in Science and Engineering*. Plenum Press, Inc., New York, 1985.
- [49] R. Kalaba, Leigh Tesfatsion, and J.-L. Wang. A finite algorithm for the exact evaluation of higher-order partial derivatives of functions of many variables. *Journal of Mathematical Analysis and Applications*, 12:181 – 191, 1983.
- [50] L. V. Kantorovich. Ob odnoĭ matematicheskoi simvolike, udobnoi pri provedenii vychislenii na mashinakh. *Doklady Akademii Nauk SSSR*, 113(4):738 – 741, 1957.
- [51] G. Kedem. Automatic differentiation of computer programs. *ACM Trans. Math. Software*, 6(2):150 – 165, June 1980.
- [52] L. Michelotti. MXYZPTLK: A C++ Hacker's implementation of automatic differentiation. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.
- [53] W. Miller. Software for roundoff analysis. *ACM Trans. Math. Software*, 1(2):108 – 128, 1975.
- [54] J. J. Moré. The Levenberg-Marquardt algorithm: implementation and theory. In G. A. Watson, editor, *Numerical Analysis*, pages 105 – 116. Lecture Notes in Math. 630, Springer Verlag, Berlin, 1977.
- [55] J. J. Moré, B. S. Garbow, and K. E. Hillstom. Implementation guide for MINPACK-1. Technical Report ANL-80-68, Mathematics and Computer Sciences Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4801, 1980.
- [56] J. J. Moré, B. S. Garbow, and K. E. Hillstom. User's guide for MINPACK-1. Technical Report ANL-80-74, Mathematics and Computer Sciences Division, Argonne National Laboratory, 9700 South Cass Ave., Argonne, IL 60439-4801, 1980.
- [57] I. M. Navon and U. Muller. FESW — A finite-element Fortran IV program for solving the shallow water equations. *Advances in Engineering Software*, 1:77 – 84, 1970.

- [58] Richard D. Neidinger. An efficient method for the numerical evaluation of partial derivatives of arbitrary order. Preprint, Davidson College, Davidson, NC 28036, 1990.
- [59] G. M. Ostrovskii, J. M. Wolin, and W. W. Borisov. Über die Berechnung von Ableitungen. *Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13(4):382 – 384, 1971.
- [60] R. Pavelle and P. S. Wang. MACSYMA from F to G. *Journal of Symbolic Computation*, 1(1):69 – 100, March 1985.
- [61] M. J. D. Powell. A hybrid method for nonlinear equations. In P. Rabinowitz, editor, *Numerical Methods for Nonlinear Algebraic Equations*, pages 87–114. Gordon and Breach, London, 1970.
- [62] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.
- [63] Louis B. Rall. Differentiation in Pascal-SC: Type GRADIENT. *ACM Trans. Math. Software*, 10(2):161 – 184, June 1984.
- [64] B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL 61801, January 1980.
- [65] G.W. Stewart. *Introduction to Matrix Computations*. Academic Press, New York, 1973.
- [66] B. Van Domselaar and P. W. Hemker. Nonlinear parameter estimation in initial value problems. Technical report, Mathematisch Centrum, 1975.
- [67] R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463 – 464, 1964.
- [68] P. Werbos. Applications of advances in nonlinear sensitivity analysis. In *Systems Modeling and Optimization*, pages 762 – 777, New York, 1982. Springer Verlag.
- [69] Anthony S. Wexler. An algorithm for exact evaluation of multivariate functions and their derivatives to any order. *Computational Statistics and Data Analysis*, 6:1 – 6, 1988.

- [70] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, Reading, MA, 1988.