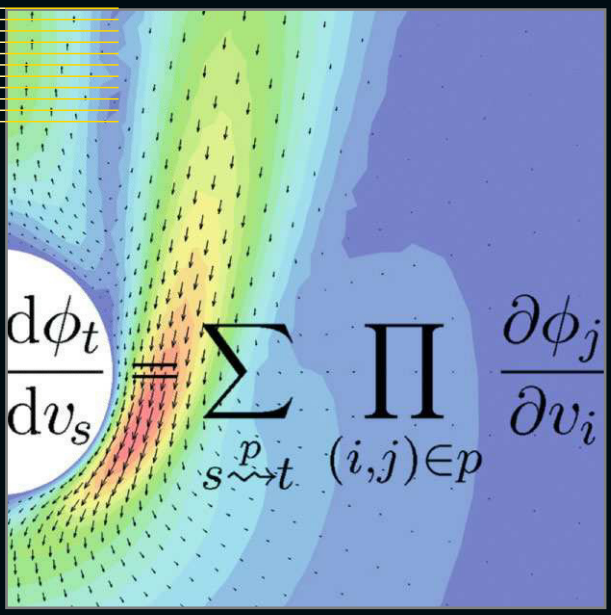
Lecture Notes in Computational Science and Engineering

50



Martin Bücker
George Corliss
Paul Hovland
Uwe Naumann
Boyana Norris
Editors

Automatic Differentiation: Applications, Theory, and Implementations

Springer

Lecture Notes
in Computational Science
and Engineering

50

Editors

Martin Bücker George Corliss Paul Hovland
Uwe Naumann Boyana Norris (Eds.)

Automatic Differentiation: Applications, Theory, and Implementations

With 108 Figures and 33 Tables

Springer

Editors

Martin Bücker
Institute for Scientific Computing
RWTH Aachen University
D-52056 Aachen, Germany
email: buecker@sc.rwth-aachen.de

Uwe Naumann
Software and Tools
for Computational Engineering
RWTH Aachen University
D-52056 Aachen, Germany
email: naumann@stce.rwth-aachen.de

George Corliss
Department of Electrical
and Computer Engineering
Marquette University
1515 W. Wisconsin Avenue
P.O. Box 1881
Milwaukee, WI 53201-1881, USA
email: george.corliss@marquette.edu

Paul Hovland
Boyana Norris
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S Cass Ave
Argonne, IL 60439, USA
email: hovland@mcs.anl.gov
norris@mcs.anl.gov

Library of Congress Control Number: 2005934452

Mathematics Subject Classification: 65Y99, 90C31, 68N19

ISBN-10 3-540-28403-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28403-1 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media
springer.com

Printed in The Netherlands

Typesetting: by the authors and TechBooks using a Springer LaTeX macro package

Cover design: *design & production* GmbH, Heidelberg

Printed on acid-free paper SPIN: 11360537 46/TechBooks 5 4 3 2 1 0

Preface

The Fourth International Conference on Automatic Differentiation was held July 20-23 in Chicago, Illinois. The conference included a one day short course, 42 presentations, and a workshop for tool developers. This gathering of automatic differentiation researchers extended a sequence that began in Breckenridge, Colorado, in 1991 and continued in Santa Fe, New Mexico, in 1996 and Nice, France, in 2000. We invited conference participants and the general automatic differentiation community to submit papers to this special collection. The 28 accepted papers reflect the state of the art in automatic differentiation.

The number of automatic differentiation tools based on compiler technology continues to expand. The papers in this volume discuss the implementation and application of several compiler-based tools for Fortran, including the venerable ADIFOR, an extended NAGWare compiler, TAF, and TAPENADE. While great progress has been made toward robust, compiler-based tools for C/C++, most notably in the form of the ADIC and TAC++ tools, for now operator-overloading tools such as ADOL-C remain the undisputed champions for reverse-mode automatic differentiation of C++. Tools for automatic differentiation of high level languages, including COSY and ADiMat, continue to grow in importance as the productivity gains offered by high-level programming are recognized.

The breadth of automatic differentiation applications also continues to expand. This volume includes papers on accelerator design, chemical engineering, weather and climate modeling, dynamical systems, circuit device modeling, structural dynamics, and radiation treatment planning. The last application is representative of a general trend toward more applications in the biological sciences. This is an important trend for the continued growth of automatic differentiation, as new applications identify novel uses for automatic differentiation and present new challenges to tool developers. The papers in this collection demonstrate both the power of automatic differentiation to facilitate new scientific discoveries and the ways in which application requirements can drive new developments in automatic differentiation.

Advances in automatic differentiation theory take many forms. Progress in mathematical theory expands the scope of automatic differentiation and its variants or identifies new uses for automatic differentiation capabilities. Advances in combinatorial theory reduce the cost of computing or storing derivatives. New compiler theory identifies analyses that reduce the time or storage requirements for automatic differentiation, especially for the reverse mode. This collection includes several papers on mathemetical and combinatorial theory. Furthermore, several of the tools papers document the compiler analyses that are required to construct an effective automatic differentiation tool.

This collection is organized as follows. The first two papers, by Rall and Werbos, provide an overview of automatic differentiation and place it in a historical context. The first section, comprising seven papers, covers advances in automatic differentiation theory. The second section, containing eight papers, describes the implementation of automatic differentiation tools. The final section, devoted to applications, includes eleven papers discussing new uses for automatic differentiation. Many papers include elements of two or more of the general themes of theory, tools, and applications. For example, in many cases successful application of an automatic differentiation tool in a new domain requires advances in theory or tools. A collected bibliography includes all of the references cited by one of the papers in this volume, as well as all of the papers from the proceeedings of the first three conferences. The bibliography was assembled from the BibTeX database at autodiff.org, an emerging portal for the automatic differentiation community. We thank Heiner Bach for his hard work on the autodiff.org bibliographic database.

While the last four years have seen many advances in automatic differentiation theory and implementation, many challenges remain. We hope that the next International Conference on Automatic Differentiation includes reports of reverse mode source transformation tools for templated C++, proofs that minimizing the number of operations in a Jacobian computation is NP-complete, advances in the efficient computation of Hessians, and many examples of new applications that identify research challenges and drive tool development forward. Together, researchers in automatic differentiation applications, theory, and implementation can advance the field in new and unexpected ways.

Martin Bücker
George Corliss
Paul Hovland
Uwe Naumann
Boyana Norris

Contents

List of Contributors

Mauricio Araya-Polo
INRIA Sophia-Antipolis, projet TROPICS
2004 route des Lucioles
BP 93
06902 Sophia-Antipolis
France
Mauricio.Araya@sophia.inria.fr

Paul I. Barton
Massachusetts Institute of Technology
Department of Chemical Engineering
77 Massachusetts Ave.
Cambridge, MA 02139
USA
pib@mit.edu

Martin Berz
Michigan State University
Department of Physics and Astronomy
East Lansing, MI 48824
USA
berz@msu.edu

Christian H. Bischof
Institute for Scientific Computing
RWTH Aachen University
D–52056 Aachen
Germany
bischof@sc.rwth-aachen.de

Frances Bodman
Cranfield University, RMCS Shrivenham
Computer Information Systems Engineering Dept.
Swindon
SN6 8LA
United Kingdom

H. Martin Bücker
Institute for Scientific Computing
RWTH Aachen University
D–52056 Aachen
Germany
buecker@sc.rwth-aachen.de

Richard Casey
Center for Applied Mathematics
Cornell University
Ithaca, NY 14853–2401
USA
rjc20@cornell.edu

William Castaings
Laboratoire de Modélisation et Calcul
Institut d'Informatique et Mathématiques Appliquées de Grenoble
BP 53
38041 Grenoble Cedex 9
France
william.castaings@imag.fr

Bruce Christianson
University of Hertfordshire, Hatfield
Computer Science
College Lane
Hatfield
Hatfield, Herts.
AL10 9AB
United Kingdom
B.Christianson@herts.ac.uk

Carlos E. Christoffersen
Department of Electrical Engineering
Lakehead University
955 Oliver Road
Thunder Bay, Ontario P7B 5E1
Canada
c.christoffersen@ieee.org

George F. Corliss
Electrical and Computer Engineering
Marquette University
P.O. Box 1881
Milwaukee WI 53201–1881
USA
George.Corliss@Marquette.edu

Maurice Cox
National Physical Laboratories
Teddington
TW11 0LW
United Kingdom
Maurice.Cox@npl.co.uk

Denis Dartus
Institut de Mécanique des Fluides de Toulouse
Hydrodynamique et Environnement
1 Allée du Professeur Camille Soula
31400 Toulouse
France
dartus@imft.fr

Ronald Errico
Global Modeling and Assimilation Office, Code 610.1
Goddard Space Flight Center
Greenbelt, MD 20771
USA
rerrico@gmao.gsfc.nasa.gov

Harley Flanders
University of Michigan
1867 East Hall
Ann Arbor, MI 48109–1043
USA
harley@umich.edu

Shaun Forth
Applied Mathematics and Operational Research
Engineering Systems Department
Cranfield University, Shrivenham Campus
Swindon
SN6 8LA
United Kingdom
S.A.Forth@cranfield.ac.uk

Benedick A. Fraass
The University of Michigan Medical School
Department of Radiation Oncology
1500 E. Medical Center Dr.
Ann Arbor, MI 48109–0010
USA

Jidong Gao
Center for Analysis and Prediction of Storms
University of Oklahoma
Sarkeys Energy Center, Suite 1110
100 East Boyd Street
Norman, OK 73019–1011
USA
jdgao@ou.edu

David M. Gay
Sandia National Labs
P.O. Box 5800, MS 0370
Albuquerque, NM 87185–0370
USA
dmgay@sandia.gov

Ronald Gelaro
Global Modeling and Assimilation Office, Code 610.1
Goddard Space Flight Center
Greenbelt, MD 20771
USA
gelaro@gmao.gsfc.nasa.gov

Ralf Giering
FastOpt
Schanzenstr. 36
D–20357 Hamburg
Germany
Rald@FastOpt.com

Andreas Griewank
Humboldt-Universität zu Berlin
Institut für Mathematik
Unter den Linden 6
D–10099 Berlin
Germany
griewank@mathematik.hu-berlin.de

Johannes Grote
Michigan State University
Department of Physics and Astronomy
East Lansing, MI 48824
USA
grotejoh@msu.edu

John Guckenheimer
Mathematics Department
Cornell University
Ithaca, NY 14853–2401
USA
gucken@cam.cornell.edu

Frank P. Hart
Department of Electrical and Computer Engineering
North Carolina State University
P.O. Box 7914
Raleigh, NC 27695–7914
USA
fphart@eos.ncsu.edu

Laurent Hascoët
INRIA Sophia-Antipolis, projet TROPICS
2004 route des Lucioles
BP 93
06902 Sophia-Antipolis
France
Laurent.Hascoet@sophia.inria.fr

Marc Honnorat
Laboratoire de Modélisation et Calcul
Institut d'Informatique et Mathématiques Appliquées de Grenoble
BP 53
38041 Grenoble Cedex 9
France
marc.honnorat@imag.fr

Shahadat Hossain
University of Lethbridge
4401 University Drive
T1K 3M4 Lethbridge, AB
Canada
shahadat.hossain@uleth.ca

Paul D. Hovland
University of Chicago / Argonne National Laboratory
9700 S. Cass Ave
Argonne, IL 60439
USA
hovland@mcs.anl.gov

Kyung-Wook Jee
The University of Michigan Medical School
Department of Radiation Oncology
1500 E. Medical Center Dr.
Ann Arbor, MI 48109–0010
USA
wook@umich.edu

Carol Johnstone
MS 221 Fermilab
Kirk Rd. & Wilson St.
Batavia, IL 60510
USA
cjj@fnal.gov

Thomas Kaminski
FastOpt
Schanzenstr. 36
D–20357 Hamburg
Germany
Thomas@FastOpt.com

Andy J. Keane
University of Southampton
School of Engineering Sciences
Mechanical Engineering
Highfield, Southampton
SO17 1BJ
United Kingdom
ajk@soton.ac.uk

Nikhil Kriplani
Department of Electrical and Computer Engineering
North Carolina State University
P.O. Box 7914
Raleigh, NC 27695–7914
USA
nmkripla@unity.ncsu.edu

Koichi Kubota
Dept. Information and System Engineering
Chuo University
1-13-27 Kasuga, Bunkyo-ku
112-8551 Tokyo
Japan
kubota@ise.chuo-u.ac.jp

François-Xavier Le Dimet
Laboratoire de Modélisation et Calcul
Institut d'Informatique et Mathématiques Appliquées de Grenoble
BP 53
38041 Grenoble Cedex 9
France
francois-xavier.le-dimet@imag.fr

Youssef Loukili
Laboratoire de Modélisation et Calcul
Institut d'Informatique et Mathématiques Appliquées de Grenoble
BP 53
38041 Grenoble Cedex 9
France
youssef.loukili@imag.fr

Sonali R. Luniya
Department of Electrical and Computer Engineering
North Carolina State University
P.O. Box 7914
Raleigh, NC 27695–7914
USA
srluniya@unity.ncsu.edu

Kyoko Makino
Michigan State University
Department of Physics and Astronomy
East Lansing, MI 48824
USA
makino@msu.edu

Shashikant L. Manikonda
Michigan State University
Department of Physics and Astronomy
East Lansing, MI 48823
USA
manikond@msu.edu

Wolfgang Marquardt
Process Systems Engineering
RWTH Aachen University
D–52056 Aachen
Germany
marquardt@lpt.rwth-aachen.de

William Martin
Center for Analysis and Prediction of Storms
University of Oklahoma
Sarkeys Energy Center, Suite 1110
100 East Boyd Street
Norman, OK 73019–1011
USA
wjmartin@ou.edu

Daniel L. McShan
The University of Michigan Medical School
Department of Radiation Oncology
1500 E. Medical Center Dr.
Ann Arbor, MI 48109–0010
USA

Jérôme Monnier
Laboratoire de Modélisation et Calcul
Institut d'Informatique et Mathématiques Appliquées de Grenoble
BP 53
38041 Grenoble Cedex 9
France
jerome.monnier@imag.fr

Uwe Naumann
RWTH Aachen University
LuFG Software and Tools for Computational Engineering
D–52056 Aachen
Germany
naumann@stce.rwth-aachen.de

Boyana Norris
University of Chicago / Argonne National Laboratory
9700 S. Cass Ave
Argonne, IL 60439
USA
norris@mcs.anl.gov

Derya B. Özyurt
Massachusetts Institute of Technology
Department of Chemical Engineering
77 Massachusetts Ave.
Cambridge, MA 02139
USA
derya@mit.edu

Valérie Pascual
INRIA Sophia-Antipolis, projet TROPICS
2004 route des Lucioles
BP 93
06902 Sophia-Antipolis
France
valerie.pascual@sophia.inria.fr

Monika Petera
Institute for Scientific Computing
RWTH Aachen University
D–52056 Aachen
Germany
petera@sc.rwth-aachen.de

Eric T. Phipps
Sandia National Laboratories
P.O. Box 5800 MS-0316
Albuquerque, NM 87185–0370
USA
etphipp@sandia.gov

John D. Pryce
Cranfield University, RMCS Shrivenham
Computer Information Systems Engineering Dept.

Swindon
SN6 8LA
United Kingdom
J.D.Pryce@cranfield.ac.uk

Louis Rall
University of Wisconsin-Madison
5101 Odana Road
Madison, WI 53711
USA
rall@math.wisc.edu

Jan Riehme
Humboldt-Universität zu Berlin
Institut für Mathematik
Unter den Linden 6
D–10099 Berlin
Germany
riehme@mathematik.hu-berlin.de

Sebastian Schlenkrich
Institute for Scientific Computing
Technical University Dresden
D–01062 Dresden
Germany
schlenk@math.tu-dresden.de

Pavel Snopok
MS 221 Fermilab
Kirk Rd. & Wilson St.
Batavia, IL 60510
USA
snopok@fnal.gov

Michael B. Steer
Department of Electrical and
Computer Engineering
North Carolina State University
P.O. Box 7914
Raleigh, NC 27695–7914
USA
m.b.steer@ieee.org

Trond Steihaug
Department of Informatics
University of Bergen
N-5020 Bergen
Norway
trond.steihaug@ii.uib.no

Julia Sternberg
Technical University Dresden
Department of Mathematics
Institute of Scientific Computing
D–01062 Dresden
Germany
jstern@math.tu-dresdent.de

Mohamed Tadjouddine
Applied Mathematics and
Operational Research
Engineering Systems Department
Cranfield University, Shrivenham
Campus
Swindon
SN6 8LA
United Kingdom
M.Tadjouddine@cranfield.ac.uk

Ricardo Todling
Global Modeling and Assimilation
Office, Code 610.1
Goddard Space Flight Center
Greenbelt, MD 20771
USA
rtodling@gmao.gsfc.nasa.gov

Jean Utke
University of Chicago / Argonne
National Laboratory
9700 S. Cass Ave
Argonne, IL 60439
USA
utke@mcs.anl.gov

Andre Vehreschild
Institute for Scientific Computing
RWTH Aachen University
D–52056 Aachen
Germany
vehreschild@sc.rwth-aachen.de

Michael Voßbeck
FastOpt
Schanzenstr. 36
D–20357 Hamburg
Germany
Michael@FastOpt.com

Andrea Walther
Technische Universität Dresden
Fachrichtung Mathematik
Institut für Wissenschaftliches Rechnen
D–01062 Dresden
Germany
awalther@math.tu-dresden.de

Paul J. Werbos
National Science Foundation
4201 Wilson Boulevard
ECS Division, Room 675
Arlington, VA 22203
USA
pwerbos@nsf.gov

Nathan Winslow
Global Modeling and Assimilation Office, Code 610.1
Goddard Space Flight Center
Greenbelt, MD 20771
USA

Jutta Wyes
Process Systems Engineering
RWTH Aachen University
D–52056 Aachen
Germany
wyes@lpt.rwth-aachen.de

Ying Xiao
School of Computer Science
University of Oklahoma
200 Felgar Street
Norman, OK 73019–6151
USA
ying_xiao@ou.edu

Ming Xue
Center for Analysis and Prediction of Storms
University of Oklahoma
Sarkeys Energy Center, Suite 1110
100 East Boyd Street
Norman, OK 73019–1011
USA
mxue@ou.edu

Perspectives on Automatic Differentiation: Past, Present, and Future?

Louis B. Rall

University of Wisconsin – Madison, Madison, WI, USA
rall@math.wisc.edu

Summary. Automatic (or algorithmic) differentiation (AD) is discussed from the standpoint of transformation of algorithms for evaluation of functions into algorithms for evaluation of their derivatives. Such finite numerical algorithms are commonly formulated as computer programs or subroutines, hence the use of the term "automatic." Transformations to evaluate derivatives are thus based on the well-known formulas for derivatives of arithmetic operations and various differentiable intrinsic functions which constitute the basic steps of the algorithm. The chain rule of elementary calculus then guarantees the validity of the process. The chain rule can be applied in various ways to obtain what are called the "forward" and "reverse" modes of automatic differentiation. These modes are described in the context of the early stages of the development of AD, and a brief comparison is given. Following this brief survey, a view of present tasks and future prospects focuses on the need for further education, communication of results, and expansion of areas of application of AD. In addition, some final remarks are made concerning extension of the method of algorithm transformation to problems other than derivative evaluation.

Key words: Numerical algorithms, algorithm transformation, history

The perspectives on automatic differentiation (AD) presented here are from a personal point of view, based on four decades of familiarity with the subject. No claim is made of comprehensive or complete coverage of this now large subject, such a treatment would require a much more extensive work; hence, the question mark in the title. In the time frame considered, AD has gone through the stages listed by Bell [30] in the acceptance of a useful technique: "It is utter nonsense; it is right and can be readily justified; everyone has always known it and it is in fact a trivial commonplace of classical analysis."

It is convenient to adopt as viewpoints on work in AD the classification given by Corliss in the preface to [42]: Methods, Applications, and Tools. Methods relate to the underlying theories and techniques, applications are

what motivate the work, and the final results in terms of computer software are the tools which actually produce solutions to problems.

1 The Algorithmic Approach

Before the middle of the 17th century, algebraic mathematics was based on *algorithms*, that is, step-by-step recipes for the solution of problems. A famous example is Euclid's algorithm for the g.c.d. of two integers, which goes back to the middle of the 3rd century B.C. The term "algorithm" comes from the name of the Islamic mathematician Mohammed ibn Mûsâ al-Khowârizmî, who around 825 A.D. published his book *Hisâb al-jabr w'al-muqâ-balah*, the title of which also gave us the word "algebra."

For centuries, geometers had the advantage of using intuitively evident visual symbols for lines, circles, triangles, etc., and could thus exploit the power of the human brain to process visual information. By contrast, human beings perform sequential processing, such as adding up long columns of numbers, rather slowly and poorly. This made what today are considered rather trivial algebraic operations opaque and difficult to understand when only algorithms were available. This changed in the 17th century with the development and general use of formulas to make algebra visible, and led to the rapid development of mathematics. In particular, I. Newton and G. Leibniz introduced calculus early in this age of formalism. Although their concept of derivative has been put on a sounder logical basis and generalized to broader situations, their basic formulas still underlie AD as practiced today.

The power of the choice of suitable notation and the manipulation of the resulting formulas to obtain answers in terms of formulas was certainly central to modern mathematics over the last 350 years, and will continue to be one of the driving forces of future progress. However, the introduction of the digital computer in the middle of the 20th century has brought the return of the importance of algorithms, now in the form of computer programs. This points to the problem of finding methods for manipulation of algorithms which are as effective as those for formulas.

In modern notation, a *finite* algorithm generates a sequence

$$s = (s_1, s_2, s_3 \ldots, s_n) \ , \tag{1}$$

where s_1 is its *input*, the result s_i of the ith *step* of the algorithm is given by

$$s_i = \phi_i(s_1, \ldots, s_{i-1}), \quad i = 2, \ldots, n \ , \tag{2}$$

where ϕ_i is a function with computable result, and the *output* s_n of the algorithm defines the function f such that $s_n = f(s_1)$. The number n of *steps* of the algorithm may depend on the input s_1, but this dependence will be ignored. For a *finite numerical algorithm* (FNA), the functions ϕ_i belong to a given set Ω of arithmetic operations and certain other computable (intrinsic)

functions. Arithmetic operations are addition, subtraction, multiplication, and division, including the cases of constant (literal) operands. With this in mind, it is sufficient to consider linear combinations, multiplication, and division. For brevity, elements of Ω will be called simply "operations."

This definition of an FNA is easily generalized slightly to the case that the input is a p-vector and the output is a q-vector, or, alternatively, the algorithm s has p inputs $s_1, \ldots, s_p$ and q outputs $s_{n-q+1}, \ldots, s_n$. Such algorithms model computer programs for numerical computations.

In general, it is much easier to grasp the significance of a formula such as

$$f(x, y) = (xy + \sin x + 4)(3y^2 + 6) \ , \tag{3}$$

for a function than a corresponding FNA for its evaluation:

$$\begin{array}{ll} s_1 = x, & s_6 = s_5 + 4, \\ s_2 = y, & s_7 = \mathrm{sqr}(s_2), \\ s_3 = s_1 \times s_2, & s_8 = 3 \times s_7, \\ s_4 = \sin(s_1), & s_9 = s_8 + 6, \\ s_5 = s_3 + s_4, & s_{10} = s_6 \times s_9 \ , \end{array} \tag{4}$$

sometimes called a *code list* for $f(x, y)$. In (4), $\mathrm{sqr}(y) = y2$ has been included as an intrinsic function. However, it is important to note that some functions are defined by computer programs which implement algorithms with thousands or millions of steps and do not correspond to formulas such as (3) in any meaningful way. It follows that algorithms provide a more general definition of functions than formulas.

2 Transformation of Algorithms

In general terms, the transformation of an FNA s into an FNA

$$S = (S_1, S_2, \ldots, S_N)$$

defines a corresponding function F such that $S_N = F(S_1)$. Such a transformation is *direct* if the functions ϕ_i in (1) are replaced by appropriate functions Φ_i on a one-to-one basis. For example, when an algorithm is executed on a computer, it is automatically transformed into the corresponding algorithm for finite precision (f.p.) numbers, which can lead to unexpected results. Other direct transformations from the early days of computing are from single to double or multiple precision f.p. numbers, real to complex, and so on. Another direct transformation is from real to interval, called the *united extension* of the function f by Moore (see [378, Sect. 11.2, pp. 108–113] and [379]): Here, $S_1 = [a, b]$ and $S_N = F(S_1) = [c, d]$, where a, b, c, d are computed f.p. numbers and $f(s_1) \in S_N$ for each $s_1 \in S_1$. The point here is that f.p. numbers can actually be computed which bound the results of exact real algorithms.

The bounds provided by the united extension are guaranteed, but not always useful. Considerable effort has gone into finding interval algorithms S which provide tighter bounds for the results of real algorithms s, but details are far beyond the scope of this paper. These and other more elaborate algorithm transformations make use of replacement of operations by corresponding FNAs if necessary.

From this perspective, AD consists of transformation of algorithms for functions into algorithms for their derivatives. An immediate consequence of the definition of FNAs and the chain rule (which goes back to Newton and Leibniz) is the following

Theorem 1. *If the derivatives s_i' of the steps s_i of the FNA s can be evaluated by FNAs with operations in Ω', then the derivative*

$$s_n' = f'(s_1)s_1'$$

can be evaluated by an FNA with operations in Ω'.

Note: Substitute "formula" for FNA to get symbolic differentiation as taught in school.

As far as terminology is concerned, obtaining an FNA for the derivative is essentially *algorithmic differentiation* [225]. The intent to have a computer do the work of evaluation led to calling this *automatic differentiation* [136, 227, 450], and *computational differentiation* [42] is also perfectly acceptable. In the following, AD refers to whichever designation one prefers.

3 Development of AD

The basic mathematical ideas behind AD have been around for a long time. The methodologies of AD have been discovered independently a number of times by different people at various times and places, so no claim is made here for completeness. Other information can be found in the paper by M. Iri [281] on the history of AD. Although the methodology of AD could well have been used for evaluation of derivatives by hand or with tables and desk calculators, the circuitous method of first deriving formulas for derivatives and then evaluating those seems to have been almost universally employed. Consequently, the discussion here will be confined to the age of the digital computer.

Starting about 1962, the development of AD to the present day can be divided approximately into four decades. In the first, the simple-minded direct approach known as the forward mode was applied to a number of problems, principally at the Mathematics Research Center (MRC) of the University of Wisconsin-Madison as described later in [450]. There followed a slack period marked by lack of acceptance of AD for reasons still not entirely clear. However, interest in AD had definitely revived by 1982 due to improvements in programming techniques and the introduction of the efficient reverse mode.

Much of the progress in this era is due to the work of Andreas Griewank and his colleagues at Argonne National Laboratory (ANL). This was followed by explosive growth of work in techniques, tools, and applications of AD, as recorded in the conference proceedings [42, 136, 227], the book [225] by Griewank, and the present volume. A useful tool in the development of AD following 1980 is the *computational graph*, which is a way of visualizing an algorithm or computer program different from formulas. For example, Fig. 1 shows a computational graph for the algorithm (4). This type of graph is technically known as a directed acyclic graph (DAG), see [225]. Transformations of the algorithm such as differentiation in forward or reverse mode can be expressed as modifications of the computational graph, see for example, [280]. As indicated above, the forward and reverse modes reflect early and later stages in the development of AD, and will be considered below in more detail.

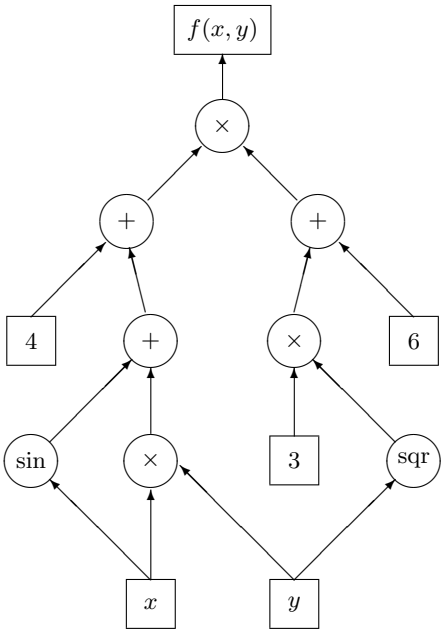


Fig. 1. A Computational Graph for $f(x, y)$.

3.1 The Forward Mode

This mode of AD consists essentially of step-by-step replacement of the operations in the FNA s for the function f by operations or FNAs for their corresponding derivatives. The result will thus be an FNA for derivatives of the result $s_n = f(s_1)$. This follows the order in which the computer program for evaluation of the function f is executed. In fact, before the introduction of

compilers with formula translation, programmers using machine or assembly language had to program evaluation of functions such as (3) in the form of a code list (4). The forward mode of AD reflects this early method of computer programming.

Early workers in AD were R. E. Moore at Lockheed Missiles and Space Company and, later and independently, R. E. Wengert and R. D. Wilkins of the Radio Guidance Operation of the General Electric Company. The work at these two locations had different motivations, but both were carried out in forward mode based on direct conversion of a code list into a sequence of subroutine calls. In reverse historical order, the method of Wengert [530] and the results of Wilkins [555] will be discussed first.

The group at General Electric was interested in perturbations of satellite motion due to nonuniformities in the gravitational field of the earth and checking computer programs which used derivatives obtained by hand. For a function $f(x_1, \ldots, x_d)$, it is often useful to approximate the difference

$$\Delta f = f(x_1 + \Delta x_1, \ldots, x_d + \Delta x_d) - f(x_1, \ldots, x_d) \; , \tag{5}$$

by the *differential*

$$df = \frac{\partial f}{\partial x_1} \Delta x_1 + \cdots + \frac{\partial f}{\partial x_d} \Delta x_d \; , \tag{6}$$

a linearization of (5) which is accurate for sufficiently small values of the increments $\Delta x_1, \ldots, \Delta x_d$. (It is customary to write dx_j instead of Δx_j in (6) to make the formula look pretty.) Leaving aside the situation that one or more of the increments may not be sufficiently small enough to make (6) as accurate as desired, the values of the partial derivatives $\partial f / \partial x_j$ give an idea of how much a change in the jth variable will perturb the value of the function f, and in which direction. Consequently, the values of these partial derivatives are known as "sensitivities."

Wengert's method used ordered pairs and does not calculate partial derivatives directly. Rather, after initialization of the values (x_j, x_j') of the independent variables and their derivatives, the result obtained is the pair (f, f'), where f is the function value and f' the *total* (or *directional*) derivative

$$f' = \frac{\partial f}{\partial x_1} x_1' + \cdots + \frac{\partial f}{\partial x_d} x_d' \; . \tag{7}$$

Values of individual partial derivatives $\partial f / \partial x_k$ are thus obtained by the initialization (x_j, δ_{jk}), δ_{jk} being the Kronecker delta. Wengert notes that higher partial derivatives can be obtained by applying the product rule to (7) and repeated evaluations with suitable initializations of (x_j, x_j'), (x_j', x_j''), and so on to obtain systems of linear equations which can be solved for the required derivatives.

As an example of the line-by-line programming required (called the "key to the method" by Wengert), starting with $\mathtt{S1}(1) = x$, $\mathtt{S1}(2) = 1$, $\mathtt{S2}(1) =$

y, S2(2) = 0, the computation of (f,$\partial f/\partial x$) of the function (3) would be programmed as

```
CALL PROD(S1, S2, S3)
CALL SINE(S1, S4)
...  ...  ...  ...                    (8)
CALL ADD(S8, 6, S9)
CALL PROD(S6, S9, S10)
```

following the code list (4), and then repeated switching the initializations to $x' = 0$ and $y' = 1$ to obtain $(f, \partial f/\partial y)$. The example given by Wilkins [555] is a function for which 21 partial derivatives are desired. The computation, after modification to avoid overflow, is repeated 21 times, and Wilkins notes the function value is evaluated 20 more times than necessary. The overflow was due to the use of the textbook formula for the derivative of the quotient by Wengert [530]. Wilkins notes an improvement suggested by his coworker K. O. Johnson to differentiate $u/v = uv^{-1}$ as a product was helpful with the overflow problem, and finally Wengert suggested the efficient expression $(u/v)' = (u' - (u/v)v')/v$ which uses the previously evaluated quotient. Also, since the function considered also depends on the time t and contains derivatives w.r.t. t, it is not clear which derivative was calculated, the ordinary total derivative (7) or the ordered derivative

$$\frac{\partial^+ f}{\partial t} = \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial t} + \cdots + \frac{\partial f}{\partial x_d}\frac{\partial x_d}{\partial t} \ ,$$

denoted by $Df/\partial t$ in [483].

Although inefficient, the program using Wengert's method, once corrected, showed there were errors in the program using derivatives obtained by hand. When the latter was corrected, both took about the same computer time to obtains answers which agreed. That Wilkins had to battle f.p. arithmetic shows that "automatic" in the sense of "plug-and-play" does not always hold for AD, as is also well-known in the case of interval arithmetic. Wilkins predicted a bright future for AD as a debugging tool and a stand-alone computational technique. However, AD seems to have hit a dead end at General Electric; nothing more appeared from this group as far as is known. As a matter of fact, the efforts of Wengert and Wilkins had no influence on the subsequent work using AD done at MRC off and on over the next ten years.

Prior to Wengert and Wilkins, R. E. Moore worked on the initial-value problem

$$\dot{x} = f(x,t), \quad x(t_0) = x_0 \ , \tag{9}$$

the goal being to compute f.p. vectors $a(t)$ and $b(t)$ such that the bounds $a(t) \leqq x(t) \leqq b(t)$ are guaranteed. Moore used recurrence relations for the Taylor series expansion of $f(x,t)$ to obtain the Taylor expansion

$$x(t_0 + \tau) = \sum_{k=0}^{m} x_k + R_m \ , \tag{10}$$

of the solution, where

$$x_k = \frac{1}{k!}\frac{d^k x(t_0)}{dt^k}\tau^k \tag{11}$$

is the kth normalized Taylor coefficient of the function $x(t)$, and the remainder term R_m is given by

$$R_m = \frac{1}{(m+1)!}\frac{d^{m+1}x(\vartheta)}{dt^{m+1}}\tau^{m+1}, \quad t_0 \leqq \vartheta \leqq t_0 + \tau\ . \tag{12}$$

Moore used interval arithmetic to bound the round-off error in the computation of the Taylor coefficients (11) and the truncation error (12) on the interval $[t_0, t_0 + \tau]$. In this way, valid assertions could be made about the results of an algorithm carried out in f.p. interval arithmetic. As in the later work of Wengert, the expansion (10) was based on representation of the function $f(x,t)$ by a code list and was programmed as a sequence of subroutine calls such as (8).

Moore presented his results to conferences on error in digital computation held at MRC in 1964 and 1965, see [376,377]. It was recognized immediately that Moore's method also applied to the direct evaluation of partial derivatives of functions of several variables, rather than via total derivatives as done by Wengert. The motivation was automation of Newton's method in d dimensions for approximate solution of $F(x) = 0$ by solving the sequence of linear equations

$$F'(x)(x_{m+1} - x_m) = -F(x_m), \quad m = 0, 1, \dots\ , \tag{13}$$

where the coefficient matrix is the Jacobian $F'(x) = (\partial F_i/\partial x_j)$ of the system of functions $F_i(x),\ i = 1, \dots, d$. The rows of the matrix $F'(x)$ are the gradients $\nabla F_i(x)$ of the corresponding functions $F_i(x)$. Furthermore, in order to apply the theorem of Kantorovich (see [449]) on the convergence of Newton's method, a Lipschitz constant for $F'(x)$ is required. This can be obtained from an upper bound for the ∞-norm of the Hessian operator

$$K \geqq \|F''(x)\| = \left\| \frac{\partial^2 F_i}{\partial x_j \partial x_k} \right\|\ .$$

The necessary bounds were computed using interval arithmetic, so that valid assertions regarding the existence of a solution and a region containing it were obtained as well as the convergence of the Newton sequence (13) when successful.

This program and others written at MRC by an outstanding programming staff supervised by L. Rall incorporated a number of advances over previous efforts in several respects. First of all, the programs accepted expressions (functions) as input and produced the corresponding sequences of subroutine calls internally, thus relieving the user of this unnecessary task. Secondly, gradients and Hessians were vectorized, so only one pass was required to obtain

the value of a function, its gradient vector, and Hessian matrix. First and second derivatives of operations and intrinsic functions were coded explicitly, rather than using Taylor coefficients or the product rule and linear equations as indicated by Wengert. Moore's program for initial-value problems was also modified to accept expressions as inputs. Finally, a program for numerical integration with guaranteed error bounds was written to accept subroutines (which could be single expressions) as input. For more details on the programs written at MRC, see [450].

Also at the University of Wisconsin, G. Kedem wrote his 1974 Ph.D. thesis on automatic differentiation of programs in forward mode, supervised by C. de Boor. It was published in 1980 [302]. For various reasons, work on AD at MRC came to a pause in 1974, and was not taken up again until 1981. This followed lectures given at the University of Copenhagen [450] and a visit to the University of Karlsruhe to learn about the computer language Pascal-SC, developed by U. Kulisch and his group (see [66] for a complete description). This extension of the computer language Pascal permits operator overloading and functions with arbitrary result types, and thus presents a natural way to program AD in forward mode, see [451] for example. Much of this work was done in collaboration with G. Corliss of Marquette University [133]. Another result was an adaptive version of the self-validating numerical integration program written earlier at MRC in nonadaptive form [137]. Funding of MRC was discontinued in 1985, which brought an end to this era of AD.

Another result of the technique of operator overloading was the concept of *differentiation arithmetic*, introduced in an elementary paper by Rall [452]. This formulation was based on operations on ordered pairs (a, a') (as in [530]). In algebraic terms, this showed that AD could be considered to be a derivation of a commutative ring, the rule for multiplication being the product rule for derivatives. Furthermore, M. Berz noticed that in the definition $(a, a') = a(1, 0) + a'(0, 1)$, the quantity $(1, 0)$ is a basis for the real numbers and, in the lexicographical ordering, $(0, 1)$ is a nonzero quantity less than any positive number and hence satisfies the classical definition of an infinitesimal. Starting from this observation, Berz was able to frame AD in terms of operations in a Levi-Civita field [37].

3.2 The Reverse Mode

Along with the revival of the forward mode of AD after 1980, the reverse mode came into prominence. As in the case of the forward mode, the history of the reverse mode is somewhat murky, featured by anticipations, publication in obscure sources [420], Ph.D. theses which were unpublished [487] or not published until much later. For example, the thesis of P. Werbos [532] was not published until twenty years later [543]. Fortunately, the thesis of B. Speelpenning [487] attracted the attention of A. Griewank at ANL, and further notice was brought to the reverse mode by the paper of M. Iri [280].

The basic idea of the reverse mode for the case the algorithm (1) has d inputs and one output is to apply the chain rule to calculate the "adjoints,"

$$\frac{\partial s_n}{\partial s_n}, \frac{\partial s_n}{\partial s_{n-1}}, \ldots, \frac{\partial s_n}{\partial s_d}, \ldots, \frac{\partial s_n}{\partial s_1} \ ,$$

which provide in reverse order the components of the gradient vector

$$\nabla f = \nabla s_n = \left(\frac{\partial s_n}{\partial s_1}, \ldots, \frac{\partial s_n}{\partial s_d} \right) \ .$$

The reverse mode resembles symbolic differentiation in the sense that one starts with the final result in the form of a formula for the function and then applies the rules for differentiation until the independent variables are reached. For example, (3) is a product, so the factors

$$\frac{\partial s_{10}}{\partial s_9} = s_6 = xy + \sin x + 4, \quad \frac{\partial s_{10}}{\partial s_6} = s_9 = 3y2 + 6 \ ,$$

are taken as new differentiation problems, with the final results of each composed by the product rule to obtain

$$\begin{aligned} \frac{\partial f}{\partial x} &= \frac{\partial s_{10}}{\partial s_1} = (y + \cos x)(3y2 + 6), \\ \frac{\partial f}{\partial y} &= \frac{\partial s_{10}}{\partial s_2} = x(3y2 + 6) + 6y(xy + \sin x + 4) \ , \end{aligned} \tag{14}$$

which can be "simplified" further if desired. Applied to the example code list (4), the reverse form yields

$$\begin{aligned} \frac{\partial s_{10}}{\partial s_{10}} &= 1, & \frac{\partial s_{10}}{\partial s_6} &= s_9, \\ \frac{\partial s_{10}}{\partial s_9} &= s_6, & \frac{\partial s_{10}}{\partial s_5} &= \frac{\partial s_{10}}{\partial s_6}\frac{\partial s_6}{\partial s_5} = s_9 \times 1, \\ \frac{\partial s_{10}}{\partial s_8} &= \frac{\partial s_{10}}{\partial s_9}\frac{\partial s_9}{\partial s_8} = s_6 \times 1, & \frac{\partial s_{10}}{\partial s_4} &= \frac{\partial s_{10}}{\partial s_5}\frac{\partial s_5}{\partial s_4} = s_9 \times 1, \\ \frac{\partial s_{10}}{\partial s_7} &= \frac{\partial s_{10}}{\partial s_8}\frac{\partial s_8}{\partial s_7} = s_6 \times 3, & \frac{\partial s_{10}}{\partial s_3} &= \frac{\partial s_{10}}{\partial s_5}\frac{\partial s_5}{\partial s_3} = s_9 \times 1 \ , \end{aligned}$$

with the final results

$$\begin{aligned} \frac{\partial s_{10}}{\partial s_2} &= \frac{\partial s_{10}}{\partial s_7}\frac{\partial s_7}{\partial s_2} + \frac{\partial s_{10}}{\partial s_3}\frac{\partial s_3}{\partial s_2} = (3s_6)(2s_2) + s_9 s_1, \\ \frac{\partial s_{10}}{\partial s_1} &= \frac{\partial s_{10}}{\partial s_4}\frac{\partial s_4}{\partial s_1} + \frac{\partial s_{10}}{\partial s_3}\frac{\partial s_3}{\partial s_1} = s_9 s_1 + s_9 s_2 \ , \end{aligned}$$

the same values as given by (14). Even in this simple case, fewer operations are required by this computation than forward evaluation of the algorithm (4) using the pairs $S_i = (s_i, \nabla s_i)$. Programming of the reverse mode is more elaborate than the forward mode, however, operator overloading can be done in reverse mode as in ADOL-C [288].

3.3 A Comparison

Both forward and reverse modes have their places in the repertoire of computational differentiation, and are sometimes used in combination. The efficiency of the reverse mode is sometimes offset by the necessity to store results of a long algorithm, see [429,550], for example. A theoretical comparison has been given by Rall [174, pp. 233–240] based on the matrix equivalent of the computational graph. If the algorithm (1) is differentiable, then its Jacobian matrix $J = (\partial s_i/\partial s_j)$ is of the form $J = I - K$, where K is lower-triangular and sparse. The eigenvalues of J are all equal to 1, and $K^{\nu+1} = 0$ for some index ν. The row vector

$$R = [0 \cdots 0 \ \nabla s_n]$$

is a left eigenvector of J, and the columns of the $n \times d$ matrix C with rows $\nabla s_1, \nabla s_2, \ldots, \nabla s_n$ are right eigenvectors of J. The reverse and forward modes consist of calculating these eigenvectors by the power method. The reverse mode starts with $R_0 = [0 \ \cdots \ 0 \ 1]$ and proceeds by $R_k = R_{k-1}J$ and terminates with $R_\mu = R$. Similarly, the forward mode starts with $C_0 = [\nabla s_1^T \ \cdots \ \nabla s_d^T \ 0 \ \cdots \ 0]^T$, proceeds by $C_k = JC_{k-1}$, and terminates with $C_\mu = C$. The difference in computational effort is immediately evident.

4 Present Tasks and Future Prospects

The future of AD depends on what is done now as well as what transpired in the past. Current tasks can be divided into four general categories: Techniques, education, communication, and applications. Some brief remarks will be devoted to each of these topics.

4.1 Techniques

The basic techniques of AD are well understood, but little attention has been devoted to accuracy, the assumption being that derivatives are obtained about as accurately as function values. The emphasis has been on speed and conservation of storage. Increasing speed by reducing the number of operations required is of course helpful, since the number of roundings is also decreased. Advantage can also be taken of the fact that $ab + c$ is often computed with a single rounding. Even more significant would be the provision of a long accumulator to evaluate the dot product

$$u \cdot v = \sum_{i=1}^{d} u_i v_i$$

of d-vectors u and v with a single rounding as implemented originally in Pascal-SC [66]. This enables many algebraic operations including solution of

linear systems to be carried out with high accuracy. For example, the components of the gradient $\nabla(u \cdot v)$ of a dot product can be expressed as dot products and thus computed with a single rounding. Also, if $x = L^{-1}y$ is the solution of a nonsingular system of equations $Lx = y$, then its gradient ∇x is given by the generalization of the division formula

$$\nabla x = L^{-1}\nabla y - L^{-1}(\nabla L)L^{-1}y = L^{-1}(\nabla y - (\nabla L)x) \ ,$$

(see [449]), where ∇L is a $d \times d$ matrix of gradients and ∇y is a d-vector of gradients. Of course, it is unnecessary to invert L, the system of equations $L\nabla x = \nabla y - (\nabla L)x$ can be solved by the same method as for $Lx = y$.

4.2 Education

It was discouraging throughout the 1970's that the work done on AD by Moore, Wengert, and the then state of the art programs written by the MRC programming staff were ignored and even disparaged. Presentations at conferences were met with disinterest or disbelief. One reason advanced for this was the wide-spread conviction that if a function was represented by a formula, then a formula for its derivative was necessary before its derivative could be evaluated. Furthermore, the differentiation of a function defined only by an algorithm and not by a formula seemed beyond comprehension. A few simple examples could be incorporated into elementary calculus courses to combat these fallacies. As mentioned above, the standard method taught for differentiation of functions defined by formulas essentially proceeds in reverse mode. The forward mode uses the way the final result $f(x)$ is computed from the given value of x and shows that $f'(x)$ can be evaluated in the same step-by-step fashion. Furthermore, given the definitions (2), the values $x = s_1, s_2, \ldots, s_n = f(x)$ of the steps in the evaluation of $f(x)$ can be used in the reverse mode to obtain the same value of $f'(x)$. Then, for example, Newton's method can be introduced as an application of use of derivative values without the necessity to obtain formulas for derivatives. All of this can be done once the basic formulas for differentiation of arithmetic operations and some elementary functions have been taught.

It is easy to prepare a teaching "module" for AD on an elementary level. The problem is to have it adopted as part of an increasingly crowded curriculum in beginning calculus. This means that teachers and writers of textbooks on calculus have to first grasp the idea and then realize it is significant. Thus, practitioners of AD will have to reach out to educators in a meaningful way. Otherwise, there will continue to be a refractory "formulas only" community in the computational sciences who could well benefit from AD.

Opportunities to introduce AD occur in other courses, such as differential equations, optimization, and numerical analysis. Reverse mode differentiation is a suitable topic for programming courses, perhaps on the intermediate level. An informal survey of numerical analysis and other textbooks reveals that

most recommend *against* the use of derivatives, in particular regarding Newton's method and Taylor series solution of differential equations. The reason advanced is the complexity of obtaining the "required" formulas for derivatives and Taylor coefficients by hand. An exception is the recent textbook on numerical analysis by A. Neumaier [411], which begins with a discussion of function evaluation and automatic differentiation. A definite opportunity exists to introduce AD at various levels in the curricula of various fields, including business, social and biological sciences as well as the traditional physical sciences and engineering fields. This is particularly true now that most instruction is backed up by software pertinent to the subject.

4.3 Communication and Applications

An additional reason for the slow acceptance of AD in its early years was the lack of publication of results after the papers of Wengert and Wilkins [530,555]. For example, the more advanced programs written at MRC were described only in technical reports and presented at a few conferences sponsored by the U. S. Army, but not widely disseminated. The attitude of journal editors at the time seemed to be that AD was either "a trivial commonplace of classical analysis," or the subject was completely subsumed in the paper by Wengert [530]. In addition, the emphasis on interval arithmetic and assertions of validity in the MRC approach had little impact on the general computing community, which was more interested in speed than guarantees of accuracy. Furthermore, the MRC programs were tied rather closely to the computer available at the time, standards for computer and interval arithmetic had not yet been developed. The uses of AD for Taylor series in Moore's 1966 book [378] and Newton's method in Rall's 1969 book [449] were widely ignored.

A striking example of lack of communication was shown in the survey paper by Barton, Willers, and Zahar, published in 1971 [461, pp. 369–390]. This valuable and interesting work on Taylor series methods traced the use of recurrence relations as employed by Moore back to at least 1932 and included the statement, "... adequate software in the form of automatic programs for the method has been nonexistent." The authors and the editor of [461] were obviously unaware that Moore had such software running about ten years earlier [375], and his program was modified by Judy Braun at MRC in 1968 to accept expressions as input, which made it even more automatic.

Another impediment to the ready acceptance of Moore's interval method for differential equations was the "wrapping effect" [377]. This refers to unreasonably rapid increase in the width of the interval $[a(t), b(t)]$ to make these bounds for the solution useless. Later work by Lohner [343] and in particular the Taylor model concept of Berz and Makino [266,346,347] have ameliorated this situation to a great extent.

Fortunately, publication of the books [225, 450], and the conference proceedings [42, 136, 227], and the present volume have brought AD to a much

wider audience and increased its use worldwide. The field received an important boost when the precompiler ADIFOR 2.0 by C. Bischof and A. Carle was awarded the J. H. Wilkinson prize for mathematical software in 1995 (see SIAM News, Vol. 28, No. 7, August/September 1995). The increasing number of publications in the literature of various fields of applications is likewise very important, since these bring AD to the attention of potential users instead of only practitioners. These books and articles as cited in their extensive bibliographies show a large and increasing sphere of applications of AD.

5 Beyond AD

Perhaps the bright future for AD predicted 40 years ago by Wilkins has arrived or is on the near horizon. There is general acceptance of AD by the optimization and interval computation communities. With more effort directed toward education, the use of AD will probably become routine. Perhaps future generations of compilers will offer differentiation as an option, see [400]. Directions for further study are to use the lessons learned from AD to develop other algorithm transformations. A step in this direction by T. Reps and L. Rall [459] is the algorithmic evaluation of divided differences

$$[x, h]f = \frac{f(x+h) - f(x)}{h} \ . \tag{15}$$

Direct evaluation of (15) in f.p. arithmetic is problematical, whereas algorithmic evaluation is stable over a wide range of values, and approaches the value of the AD derivative $f'(x)$ as $h \to 0$. In fact, for $h = 0$, the divided difference formulas reduce to the corresponding formulas for derivatives. In the use of divided differences to approximate derivatives, (15) is inaccurate due to truncation error for h large, and due to roundoff error for h small. On the other hand, the use of differentials (6) obtained by AD to approximate differences (5) has the same problems. Thus, it is useful to have a method to compute differences which does not suffer loss of significant digits by cancelation to the extent encountered in direct evaluation.

Other goals for algorithm transformation are suggested by the "ultra-arithmetic" proposed by W. Miranker and others [295]. Algorithms for functions represented by Fourier-type series can be used to obtain the coefficients of the series expansions, much like what has already been done for Taylor series. In other words, the transformation of an FNA can be accomplished once the appropriate transformations of arithmetic operations and intrinsic functions involved are known. As initially realized by Wengert [530], this is indeed the key to the method.

Backwards Differentiation in AD and Neural Nets: Past Links and New Opportunities*

Paul J. Werbos

National Science Foundation, Arlington, VA, USA
pwerbos@nsf.gov

Summary. Backwards calculation of derivatives – sometimes called the reverse mode, the full adjoint method, or backpropagation – has been developed and applied in many fields. This paper reviews several strands of history, advanced capabilities and types of application – particularly those which are crucial to the development of brain-like capabilities in intelligent control and artificial intelligence.

Key words: Reverse mode, backpropagation, intelligent control, reinforcement learning, neural networks, MLP, recurrent networks, approximate dynamic programming, adjoint, implicit systems

1 Introduction and Summary

Backwards differentiation or "the reverse accumulation of derivatives" has been used in many different fields, under different names, for different purposes. This paper will review that part of the history and concepts which I experienced directly. More importantly, it will describe how reverse differentiation could have more impact across a much wider range of applications.

Backwards differentiation has been used in four main ways known to me:

1. In automatic differentiation (AD), a field well covered by the rest of this book. In AD, reverse differentiation is usually called the "reverse method" or the "adjoint method." However, the term "adjoint method" has actually been used to describe two different generations of methods. Only the newer generation, which Griewank has called "the true adjoint method," captures the full power of the method.
2. In neural networks, where it is normally called "backpropagation" [532, 541, 544]. Surveys have shown that backpropagation is used in a majority

* The views herein are those of the author, not the official views of NSF. However – as work done by a government employee on government time, it is in the open government domain.

of the real-world applications of artificial neural networks (ANNs). This is the stream of work that I know best, and may even claim to have originated.
3. In hand-coded "adjoint" or "dual" subroutines developed for specific models and applications, e.g., [534, 535, 539, 540].
4. In circuit design. Because the calculations of the reverse method are all local, it is possible to insert circuits onto a chip which calculate derivatives backwards physically on the same chip which calculates the quantit(ies) being differentiated. Professor Robert Newcomb at the University of Maryland, College Park, is one of the people who has implemented such "adjoint circuits." Some of us believe that local calculations of this kind must exist in the brain, because the computational capabilities of the brain require some use of derivatives and because mechanisms have been found in the brain which fit this idea.

These four strands of research could benefit greatly from greater collaboration. For example – the AD community may well have the deepest understanding of how to actually calculate derivatives and to build robust dual subroutines, but the neural network community has worked hard to find many ways of *using* backpropagation in a wide variety of applications.

The gap between the AD community and the neural network community reminds me of a split I once saw between some people making aircraft engines and people making aircraft bodies. When the engine people work on their own, without integrating their work with the airframes, they will find only limited markets for their product. The same goes for airframe people working alone. Only when the engine and the airframe are combined together, into an integrated product, can we obtain a real airplane – a product of great power and general interest.

In the same way, research from the AD stream and from the neural network stream could be combined together to yield a new kind of modular, integrated software package which would *integrate* commands to develop dual subroutines *together with* new more general-purpose systems or structures making use of these dual subroutines.

At the AD2004 conference, some people asked why AD is not used more in areas like economics or control engineering, where fast closed-form derivatives are widely needed. One reason is that the proven and powerful tools in AD today mainly focus on differentiating C or Fortran programs, but good economists only rarely write their models in C or in Fortran. They generally use packages such as Troll or TSP or SPSS or SAS, which make it easy to perform statistical analysis on their models. Engineering students tend to use MatLab. Many engineers are willing to try out very complex designs requiring fast derivatives, when using neural networks but not when using other kinds of nonlinear models, simply because backpropagation for neural networks is available "off the shelf" with no work required on their part. A more general kind of integrated software system, allowing a wide variety of user-specified

modeling modules, and compiling dual subroutines for each module type and collections of modules, could overcome these barriers. It would not be necessary to work hard to wring out the last 20 percent reduction in run time, or even to cope with strange kinds of spaghetti code written by users; rather, it would be enough to provide this service for users who are willing to live with natural and easy requirements to use structured code in specifying econometric or engineering models, etc. Various types of neural networks and elastic fuzzy logic [542] should be available as choices, along with user-specified models. Methods for combining lower-level modules into larger systems should be part of the general-purpose software package.

The remainder of this paper will expand these points and – more importantly – provide references to technical details. Section 2 will discuss the motivation and early stages of my own strand of the history. Section 3 will summarize backwards differentiation capabilities we have developed and used.

For the AD community, the most important benefit of this paper may be the new ways of *using* the derivatives in various applications. However, for reasons of space, I will weave the discussion of those applications into Sects. 2 and 3 and provide citations and URLs to more information.

This paper does not represent the official views of NSF. However, many parts of NSF would be happy to receive more proposals to strengthen this important emerging area of research. For example, consider the programs listed at `www.eng.nsf.gov/ecs`. Success rates all across the relevant parts of NSF were cut to about 10% in fiscal year 2004, but more proposals in this area would still make it possible to fund more work in it.

2 Motivations and Early History

My personal interest in backwards differentiation started in the 1960s, as an outcome of my desire to better understand how intelligence works in the human brain.

This goal still remains with me today. NSF has encouraged me to explain more clearly the same goals which motivated me in the 1960s! Even though I am in the Engineering Directorate of NSF, I ask my panelists to evaluate each proposal I receive in the program for Controls, Networks, and Computational Intelligence (CNCI) by considering (among other things) how much it would contribute to our ability to someday understand and replicate the kind of intelligence we see in the higher levels of the brains of all mammals.

More precisely, I ask my panelists to treat the ranking of proposals as a kind of strategic investment decision. I urge them to be as tough and as complete about focusing on the bottom line as any industry investor would be, except that the bottom line, the objective function, is not dollars. The bottom line is the *sum* of the potential benefits to fundamental scientific understanding, plus the potential broader benefits to humanity. The emphasis is on *potential* – the risk of losing something really big if we do *not* fund a

particular proposal. The questions "What is mind? What is intelligence? How can we replicate and understand it as a whole system?" are at the top of my list of what to look for in CNCI. But we are also looking for a wide spectrum of technology applications of strategic importance to the future of humanity. See my chapter in [479] for more details and examples.

Before we can reverse-engineer brain-like intelligence as a kind of computing system, we need to have some idea of what it is trying to compute. Figure 1 illustrates what that is:

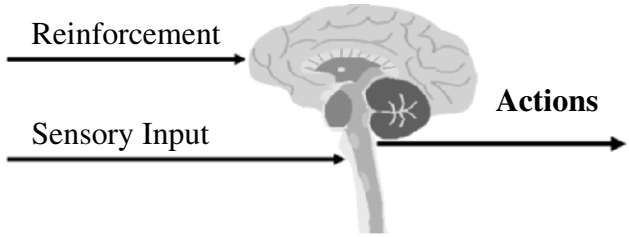


Fig. 1. The brain as a whole system is an intelligent controller.

Figure 1 reminds us of simple, trivial things that we all knew years ago, but sometimes it pays to think about simple things in order to make sure that we understand all of their implications. To begin with, Fig. 1 reminds us that the entire output of the brain is a set of nerve impulses that control *actions*, sometimes called "squeezing and squirting" by neuroscientists. The entire brain is an information processing or computing device. The purpose of any computing device is to compute its outputs. Thus the function of the brain *as a whole system* is to learn to compute the actions which best serve the interests of the organism over time. The standard neuroanatomy textbook by Nauta [404] stresses that we cannot really say *which* parts of the brain are involved in computing actions, since *all* parts of the brain feed into that computation. The brain has many interesting capabilities for memory and pattern recognition, but these are all *subsystems* or even *emergent dynamics* within the larger system. They are all subservient to the goal of the overall system – the goal of computing effective actions, ever more effective as the organism learns. Thus the design of the brain as a whole, as a computational system, is within the scope of what we call "intelligent control" in engineering. When we ask how the brain works, as a functioning engineering system, we are asking how a system made up of neurons is capable of performing learning-based intelligent control. This is the species of mathematics that we have been working to develop – along with the subsystems and tools that we need to make it work as an integrated, general-purpose system.

Many people read these words, look at Fig. 1, and immediately worry that this approach may be a challenge to their religion. Am I claiming that all human consciousness is nothing but a collection of neurons working like

a conventional computer? Am I assuming that there is nothing more to the human mind – no "soul?" In fact, this approach does not require that one agree or disagree with such statements. We need only agree that mammal brains actually do exist, and do have interesting and important computational capabilities. People working in this area have a great diversity of views on the issue of "consciousness." Because we do not need to agree on that complex issue, in order to advance this mathematics, I will not say more about my own opinions here. Those who are interested in those opinions may look at [532, 548, 549], and at the more detailed technical papers which they cite.

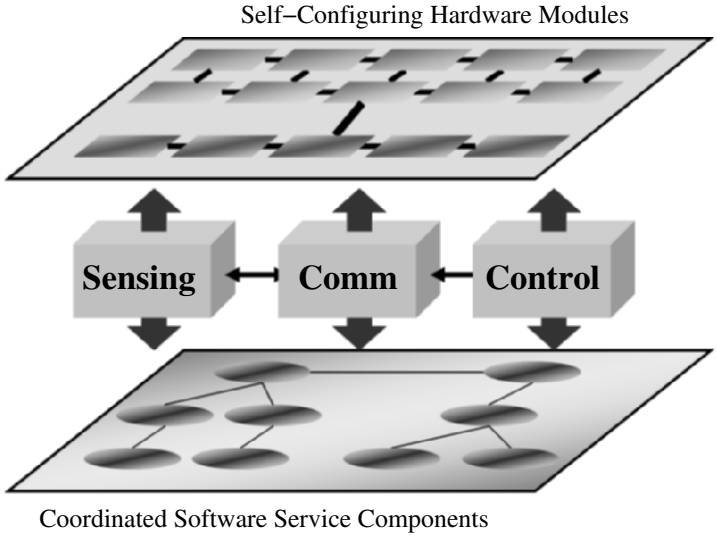


Fig. 2. Cyberinfrastructure: the entire web from sensors to decisions/action/control designed to self-heal, adapt and learn to maximize overall system performance.

Figure 2 depicts another important goal which has emerged in research at NSF and at other agencies such as the Defense Advanced Projects Agency (DARPA) and the Department of Homeland Security (DHS) Critical Infrastructure Protection. More and more, people are interested in the question of how to design a new kind of "cyberinfrastructure" which has the ability to integrate the entire web of information flows from sensors to actuators, in a vast distributed web of computations, which is capable over time to learn to optimize the performance of the actual physical infrastructure which the cyberinfrastructure controls. DARPA has used the expression "end-to-end learning" to describe this. Yet this is *precisely the same design task* we have been addressing all along, motivated by Fig. 1! Perhaps we need to replace the word "reinforcement" by the phrase "current performance evaluation" or the like, but the actual mathematical task is the same.

Many of the specific computing applications that we might be interested in working on can best be seen as *part* of a larger computational task, such as the tasks depicted in Figs. 1 or 2. These tasks can provide a kind of *integrating framework* for a general purpose software package – or even for a hybrid system composed of hardware and software together. See `www.eng.nsf.gov/ecs/` for a link to recent NSF discussions of cyberinfrastructure.

Figure 3 summarizes the origins of backpropagation and of Artificial Neural Networks (ANNs). The figure is simplified, but even so, one could write an entire book to explain fully what is here.

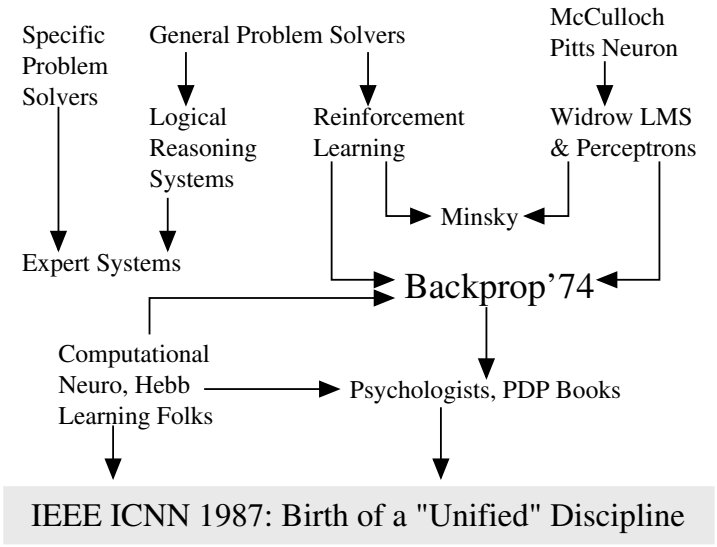


Fig. 3. Where did ANNs and backpropagation come from?

Within the ANN field proper, it is generally well-known that backpropagation was first spelled out explicitly (and implemented) in my 1974 Harvard Ph.D. thesis [532]. (For example, the IEEE Neural Network Society cited this in granting me their Pioneer Award in 1994.)

Many people assume that I developed backpropagation as an answer to Marvin Minsky's classic book *Perceptrons* [370]. In that book, Minsky addressed the challenge of how to train a specific type of ANN – the Multilayer Perceptron (MLP) – to perform a task which we now call Supervised Learning, illustrated in Fig. 4.

In supervised learning, we try to learn the nonlinear mapping from an input vector $\underline{X}$ to an output vector $\underline{Y}$, when given *examples* $\underline{X}(t)$, $\underline{Y}(t)$, $t = 1$ to T of the relationship. There are many varieties of supervised learning, and it remains a large and complex area of ANN research to this day, with links to statistics, machine learning, data mining, and so on.

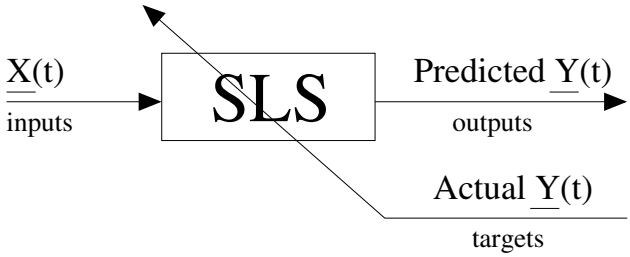


Fig. 4. What a Supervised Learning System (SLS) does.

Minsky's book was best known for arguing that 1) we need to use an MLP with a hidden layer even to represent simple nonlinear functions such as the XOR mapping; and 2) no one on earth had found a viable way to *train* MLPs with hidden layers good enough even to learn such simple functions. Minsky's book convinced most of the world that neural networks were a discredited dead-end – the worst kind of heresy. Widrow has stressed that this pessimism, which squashed the early "perceptron" school of AI, should not really be blamed on Minsky. Minsky was merely summarizing the experience of hundreds of sincere researchers who had tried to find good ways to train MLPs, to no avail. There had been islands of hope, such as the algorithm which Rosenblatt called "backpropagation" (not at all the same as what we now call backpropagation!), and Amari's brief suggestion that we might consider least squares as a way to train neural networks (without a discussion of how to get the derivatives, and with a warning that he did not expect much from the approach). But the pessimism at that time became terminal.

In the early 1970s, I visited Minsky at MIT and proposed a joint paper showing that MLPs can overcome the earlier problems if 1) the neuron model is slightly modified [534] to be differentiable; and 2) the training is done in a way that uses the reverse method, which we now call backpropagation [532, 544] in the ANN field. But Minsky was not interested [8]. In fact, no one at MIT or Harvard or any place else I could find was interested at the time.

There were people at Harvard and MIT then who had used, in control theory, a method very similar to the *first-generation* adjoint method, where calculations are carried out backwards from time T to $T-1$ to $T-2$ and so on, but where derivative calculations *at any time* are based on classical forwards methods. (In [532], I discussed first-generation work by Jacobsen and Mayne [282], by Bryson and Ho [75], and by Kashyap, which was particularly relevant to my larger goals.) Some later debunkers have argued that backpropagation was essentially a trivial and obvious extension of that earlier work. But in fact, some of the people doing that work actually controlled computer resources at Harvard and MIT at that time, and would not allow those resources to be used to test the ability of true backpropagation to train ANNs for supervised learning; they did believe there was enough evidence in 1971 that true backpropagation could possibly work.

In actuality, the challenge of supervised learning was not what really brought me to develop backpropagation. That was a later development. My initial goal was to develop a kind of universal neural network learning device to perform a kind of "Reinforcement Learning" (RL) illustrated in Fig. 5.

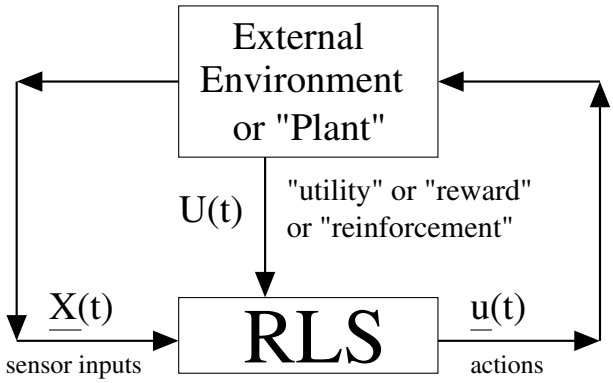


Fig. 5. A Concept of Reinforcement Learning. The environment and the RLS are both assumed to have memory at time t of the previous time $t-1$. The goal of the RLS is to learn how to maximize the sum of expected U ($\langle U \rangle$) over all future time.

Ironically, my efforts here were inspired in part by an earlier paper of Minsky [172], where he proposed reinforcement learning as a pathway to true general-purpose AI. Early efforts to build general-purpose RL systems were no more successful than early efforts to train MLPs for supervised learning, but in 1968 [531] I proposed what was then a new approach to reinforcement learning. Because the goal of RL is to maximize the sum of $\langle U \rangle$ over future time, I proposed that we build systems explicitly designed to learn an approximation to *dynamic programming*, the only exact and efficient method to solve such an optimization problem in the general case. The key concepts of classical dynamic programming are shown in Fig. 6.

In classical dynamic programming, the user supplies the utility function to be maximized (this time as a function of the state $\underline{x}(t)$!), and a stochastic model of the environment used to compute the expectation values indicated by angle brackets in the equation. The mathematician then finds the function J which solves the equation shown in Fig. 6, a form of the Bellman equation. The key theorem is that (under the right conditions) any system which chooses $\underline{u}(t)$ to solve the simple, static maximization problem within that equation will automatically provide the optimal strategy over time to solve the difficult problem in optimization over infinite time. See [479,546,553] for more complete discussions, including discussion of key concepts and notation in Figs. 6 and 7.

My key idea was to use a universal function approximator – like a neural network – to *approximate* the function J or something very similar to it, in

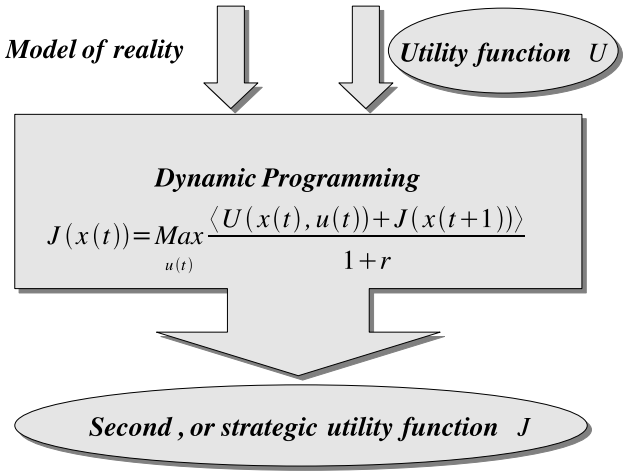


Fig. 6. The key concepts in classical dynamic programming.

order to overcome the curse of dimensionality which keeps classical dynamic programming from being useful on large problems.

In 1968, I proposed that we somehow imitate Freud's concept of a backwards flow of credit assignment, flowing back from neuron to neuron, to implement this idea. I did not provide a practical way to do this, but in my thesis proposal to Harvard in 1972, I proposed the following design, including the flow chart (with less modern labels) and the specific equations for how to use the reverse method to calculate the required derivatives indicated by the dashed lines.

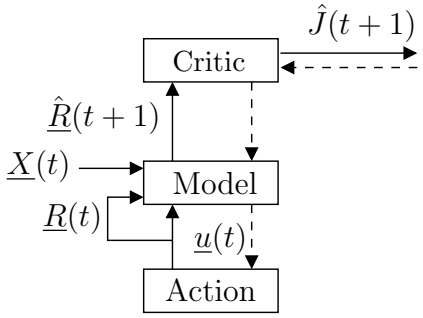


Fig. 7. RLS design proposed to Harvard in my 1972 thesis proposal.

I explained the reverse calculations using a combination of intuition and examples and the ordinary chain rule, though it was almost exactly a translation into mathematics of things that Freud had previously proposed in his theory of psychodynamics! Because of my difficulties in finding support for this

kind of work, I printed up many copies of this thesis proposal and distributed them very widely.

In Fig. 7, all three boxes were assumed to be filled in with ANNs – with ordered computational systems containing parameters or weights that would be adapted to approximate the behavior called for by the Bellman equation. For example, to make the actions $\underline{u}(t)$ actually perform the maximization which appears in the Bellman equation, we needed to know the derivatives of J with respect to every action variable (actually, every parameter in the action network). The derivatives would provide a kind of specific feedback to each parameter, to signal whether the parameter should be increased or decreased. For this reason, I called the reverse method "dynamic feedback" in [532]. The reverse method was needed to compute all the derivatives of J with respect to all of the parameters of the action network in just one sweep through the system. At that time, I focused on the case where the utility function U depends only on the state $\underline{x}$, and not on the current actions $\underline{u}$. I discussed how the reverse calculations could be implemented in a local way, in a distributed system of computing hardware like the brain.

Harvard responded as follows to this proposal and to later discussions. First, they would not allow ANNs as such to be a major part of the thesis, since I had not found anyone willing to act as a mentor for that part. (I put a few words into Chapter 5 to specify essential ideas, but no more.) Second, they said that backwards differentiation was important enough by itself for a Ph.D. thesis, and that I should postpone the reinforcement learning concepts for research after the Ph.D. Third, they had some skepticism about reverse differentiation itself, and they wanted a really solid, clear, rigorous proof of its validity in the general case. Fourth, they agreed that this would be enough to qualify for a Ph.D. if, in addition, I could show that the use of the reverse method would allow me to use more sophisticated time-series prediction methods which, in turn, would lead to the first successful implementation of Karl Deutsch's model of nationalism and social communications [146]. All of this happened [532], and is a natural lead-in to the next section.

The computer work in [532] was funded by the Harvard-MIT Cambridge Project, funded by DARPA. The specific multivariate statistical tool described in [532], made possible by backpropagation, was included as a general command in the MIT version of the TSP package in 1973-74 and, of course, described in the MIT documentation. The TSP system also included a kind of small compiler to convert user-specified formulas into Polish form for use in nonlinear regression. By mid-1974 we had almost finished coding a new set of commands (almost exactly paralleling [532]) which: (1) would allow a TSP user to specify a "model" as a set of user-specified formulas; (2) would consolidate all the Polish forms into a single compact structure; (3) would provide the obvious kinds of capabilities for testing a whole model, similar to capabilities in Troll; and (4) would automatically create a reverse code for use in prediction and optimization over time. The complete system in Fortran was almost ready for testing in mid-1974, but there was a complete reorganization

of the Cambridge Project that summer, reflecting new inputs from DOD and important improvements in coding standards based on PL/1. As I result, I graduated and moved on before the code could be moved into the new system.

3 Types of Differentiation Capability We Have Developed

3.1 Initial (1974) Version of the Reverse Method

My thesis showed how to calculate all the derivatives of a *single* computed quantity Y with respect to *all* of the inputs and parameters which fed into that computation in *just one sweep* backwards (see Fig. 8).

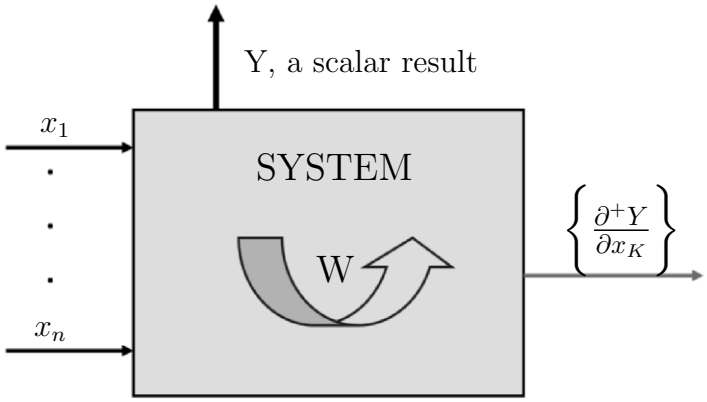


Fig. 8. Concept of the reverse method.

The first version of the reverse method required that the computational system be what I called an "ordered system." My definition of an "ordered system" in Chapter 2 of [532] was almost identical to the definition of an explicit computational algorithm given by Louis Rall in his chapter in this book [454]. At each time when we compute the scalar result Y, we need to be able to specify a sequence of intermediate computations f_1 through f_N which lead up to $Y = f_{N+1}$, where each computation is specified as a differentiable (and hopefully simple) function of what preceded it. In practice, these computations may form a kind of lattice of computations performed in parallel. However, that is just a useful and important special case of the general mathematics.

In order to specify and prove the validity of the reverse method, in the general case, I needed to define the concept of an *ordered derivative*. As shown in Fig. 8, the reverse method calculates the *entire set* of ordered derivatives of Y with respect to the set of inputs x_1 through x_n.

Many people at AD2004 asked how the reverse method could be better taught in schools. I would propose that *the very first course in calculus* that teaches partial derivatives should teach that there are at least three different *types* of partial derivative. The three different types make different assumptions, and need to be treated as distinct cases with distinct rules, to avoid confusion in the practical use of partial derivatives. I have seen enough confusion about partial derivatives in the study of complex systems, all across social sciences and basic science and engineering, that I believe it would save a lot of time in the end to be clear about these distinctions from the first.

The three basic concepts are: 1) the *algebraic* partial derivative, whose value (as an algebraic expression) depends on the *explicit algebraic expression* for the quantity being differentiated; 2) the *field* or functional partial derivative, whose value is well-defined only for a specific *set* of coordinate variables or input vector; and 3) the *ordered* derivative, which represents the *total* change in a *later* quantity which results when the value of an *earlier* quantity is changed, in an ordered system. Ordered derivatives occur in practice across all fields of science, but a confusing multitude of ad hoc terms and partial methods have been developed to deal with them. Again, it would save time to deal with the concept in a more unified and general way in basic calculus courses.

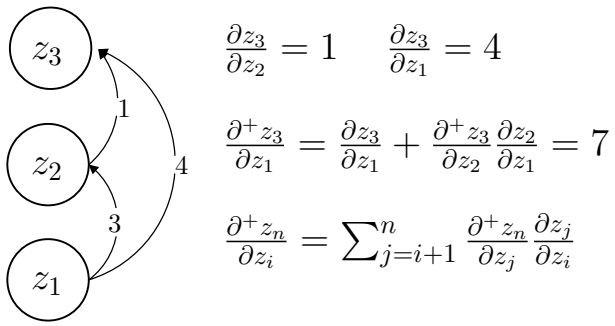


Fig. 9. The chain rule for ordered derivatives.

Figure 9 illustrates the relation between direct or algebraic partial derivatives and ordered derivatives, and gives the chain rule for ordered derivatives. In my view, the chain rule for ordered derivatives should be taught in second-year calculus classes. The proof of the chain rule in Chapter 2 of [532] (reprinted in [540]) is the proof of the validity of the reverse method. The reverse method *is* the use of this chain rule for the case of ordered systems. Notice that the direct or algebraic derivative of z_3 with respect to z_1 is only 4, because that is the direct impact along the outer arrow. However, the total or ordered derivative is 7.

For a system with n inputs as in Fig. 8, the reverse method allows one to compute all the required derivatives exactly in one pass, instead of the n passes needed with older methods. Thus it reduces costs by a factor of n. The person funding my work in the late 1970s argued that reductions in computational cost were growing less and less important, as computer costs fell. I replied that greater computing capacity is properly leading us to build ever larger models and modeling systems and control systems; thus as n grows larger and larger, the cost reduction becomes more and more important. For systems as large as the brain, the cost reduction is indispensable.

3.2 Extensions of the Reverse Method (1974-86)

During 1974-1986, I developed three kinds of extension to the reverse method: 1) extensions to calculate derivatives through "recurrent" or "implicit" systems; 2) extensions to calculate selected higher-order derivatives or even derivatives of eigenvalues or eigenvectors; and 3) extensions to manage block structured or modular computer systems. I used and published the method in several specialized areas – but interest became much broader after a detailed 1980 DOE/EIA Validation Report summarizing their capabilities and a condensed summary [534], which we distributed very widely.

Recurrent or Implicit Systems

The neural network community talks a lot about "feedforward networks," which sound identical at first to "ordered systems." A feedforward network would contain N elementary processing elements or "neurons," like the functions f_k above. At each time, the network would take n inputs (as in Fig. 8) and work forward step by step to compute its outputs. There may be more than one output, but still it is an ordered system. Neural network people often picture such a network as a kind of computational graph made up of circles and arrows (for example, see [534] or `www.nd.com`). Each circle represents the calculation of an intermediate variable, and the arrows flowing into any circle show us which earlier results are directly used in that calculation.

A "recurrent network," in neural network language, is a network which cannot be ordered, because the graph contains arrows "pointing backwards" (or looping back to the same level they start in.) The idea of recurrent or recursive neural networks was known back in Minsky's time [370]. The commonest form of recurrence is a loop *from* neuron number k *back to itself.*

The literature on recurrent networks has become very confused and often inaccurate, in part because there are different interpretations of what it means when people insert a backwards arrow into the computational graph. There are three common versions of what a backwards loop might mean: (1) a time-lagged flow of information – for example, when the calculation of neuron k at time t depends on the previous output at time $t-1$ of the same neuron; (2) an instantaneous flow of information, such that the network must

be interpreted as an *implicit system*, as a system of nonlinear simultaneous equations such that the output of the system is defined as the result of solving those equations; or (3) a flow of information in continuous time, governed by ordinary differential equations (ODE).

I have defined a Time-Lagged Recurrent Network (TLRN) as a feedforward system augmented by the first kind of recurrence. I have defined a Simultaneous Recurrent Network (SRN) as a feedforward system augmented by the second kind of recurrence. The most general case, for systems based on discrete time, is a *hybrid* TLRN/SRN, where both kinds of recurrence are present. I have worked at times with the ODE versions [539], but this usually causes more trouble than it is worth (except in certain stability proofs in control [546]). The current lack of reliable software to handle TLRN/SRN hybrids effectively is a major barrier to progress in making better use of ANNs, in my view. Time-lagged recurrence and simultaneous recurrence each provide fundamentally different kinds of modeling or computational capability. For maximum (brain-like) overall capability, it is essential to be able to combine these two capabilities without blurring the distinction between them.

TLRNs are still ordered systems, if one considers the entire web of calculations across time. Later, I defined the term "backpropagation through time"(BPTT) [541] to refer to the use of backpropagation across an ordered space-time system. Of course, the cost of a complete and exact backwards sweep to get all the derivatives is still of the same order as the cost of a forwards sweep. BPTT was implemented in [532], and numerous examples were given of ways to use it. TLRNs trained using BPTT, along with sophisticated ways of *using* the derivatives, are the core of some of the most powerful applications of ANNs today. For example, the work by Feldkamp, Prokhorov, and others at Ford Research contains many examples of the effective use of TLRNs.

True implicit systems are a more difficult case. Perhaps the easiest way to think about implicit systems is to use the definition of SRN given in Sect. 3.2.4 of [553], with minor revision. An SRN may be defined as a vector-valued mapping $\underline{F}$:

$$\underline{Y} = \underline{F}(\underline{X}, W) \tag{1}$$

defined as the result of applying a "read-out function" $\underline{g}$:

$$\underline{Y} = \underline{g}(\underline{y}^{(\infty)}, \underline{X}, W) \tag{2}$$

to the converged value $\underline{y}^{(\infty)}$ of a vector $\underline{y}$, which we update by an iteration rule

$$\underline{y}^{(n+1)} = \underline{f}(\underline{y}^{(n)}, \underline{X}, W)\ , \tag{3}$$

where $\underline{f}$ is a feedforward system, and W is a set of weights or parameters, together with some procedure for determining the initial iterate $\underline{y}^{(0)}$. I sometimes call $\underline{f}$ the "feedforward core" of the SRN.

In 1980, soon after starting work for the Office of Energy Information Validation at the Energy Information Administration (EIA) of the Department of Energy, I encountered two examples of such implicit systems: (1) an econometric model of the natural gas industry [539], which included time-lagged effects but was defined as a simultaneous-equation system, like most standard econometric models; and (2) the Long-Term Energy Analysis Package (LEAP) [535], which was a large simultaneous-equation system operating forwards and backwards through time. I had responsibility for managing two large contracts which included sensitivity analysis of such models, one at MIT [31] (for econometric models) and one at Oak Ridge National Laboratories (ORNL) evaluating LEAP.

The ORNL group had studied the best current literature on the first-generation adjoint sensitivity methods, some of which they forwarded to me. Extending that approach, they calculated "sensitivity coefficients" (ordered derivatives) for LEAP, by calculating the Jacobian of $\underline{f}$, in effect, and iterating over the gradient of (3).

Looking at this work, I realized immediately that I could *combine* their approach to addressing the simultaneous equations aspect, *together with* the use of the reverse method applied to the feedforward core in order to avoid Jacobian calculations, and together with BPTT to handle the time-lagged effects in a normal econometric model. I implemented this new unified method as follows. First, I translated the current EIA model of natural gas markets and natural gas regulation from FORTRAN into a model in the Troll system. (This took some time, but was much appreciated by EIA management, because it made it much easier for them to know precisely what was assumed inside this model.) Then I hand-coded the dual or adjoint code to go with the model, as another "model" in Troll, so that I could quickly compute the sensitivity of any model result to *all* of the many inputs and parameters of the system. The results were written up in an EIA report "published" as an energy validation report, distributed within DOE and ORNL and a few other places, and theoretically distributed to the general public. The resulting journal article [539] was delayed due to the (verified) finding that the predicted residential gas price could vary by $1 or more, in response to changes of only 0.001 in one of the elasticity parameters. The group which I managed at ORNL soon after became a primary source for the second-generation adjoint sensitivity methods. The person I exchanged papers with the most in this group retired after making a large amount of money on the stock market using neural network methods to guide his investments.

The method described in the final section of [539] is very close to the "white box method for implicit systems" as now used in the AD community. The presentation of the method in Chapters 3 and 10 of [553] may be somewhat easier to work with than [539].

SRNs are not widely used yet in ANN technology, even though many important applications will require them for real success. Part of the problem is a lack of suitable software and a need for research into how to speed up the

learning. (Kozma, Ilin, and I have recently had preliminary results cutting learning time by a factor of ten, compared with [426, 551], using the simplest partial version of some new approaches.)

Another part of the problem is a widespread lack of understanding of what SRNs can offer, if properly trained.

Most ANN users know that simple MLPs are "universal approximators." Andrew Barron of Yale [20] has proven theorems about *how many parameters* are needed to achieve a given level of approximation accuracy, when approximating smooth functions. In essence, he has proven that the required complexity grows exponentially with the number of inputs for linear basis function approximators (such as lookup tables, Taylor series, or radial basis function approximators). However, it grows much more slowly for the simple MLP. Many neural network people conclude: if MLPs are so effective in approximating any input-output mapping, why bother with the extra complexity of an SRN?

However, many tasks critical to intelligent systems require that we approximate nonsmooth functions or functions with high computational complexity. Minsky's "connectedness" function [370] is one example. Evaluating a position in a game like Go is another example. The SRN provides a kind of Turing-like capability [553] that ensures it has the most general kind of representation ability we need in practice, and we often do need it.

To try to demonstrate this, Pang and I [426, 551] showed how a simple SRN could learn to solve a kind of "generalized maze navigation" task, where MLPs and the Simple Recurrent Networks later proposed by psychologists both failed very badly. In [551] I showed how an SRN trained on six easy mazes could steadily improve its performance on six hard mazes on which it was never trained. In [426], we exhaustively discussed the five major approaches to computing the derivatives needed to train an SRN structure (four applicable to TLRNs as well). We actually used the "black box" approach in this early demonstration. The black box approach – treating the iterations of (3) as if they were time points, and using BPTT – takes more memory than the "white box" approach, but it was simple and exact, and did not lead into the tricky pitfalls of the white box method discussed at AD 2004.

In the work of [426, 551], we used a special kind of SRN, a "cellular SRN," suitable for situations where the inputs come from a kind of two-dimensional grid with translational symmetry. More recently I have developed and patented a more general special case of SRN, called an ObjectNet, for situations where the inputs may come from a more general class of relational networks (such as the state of electric power grids). ObjectNets can be used to train a single network to learn from a training set consisting of data from different power grids with different topologies and different numbers of state variables – and yet they are still an inherently distributed computational structure, like the cellular SRN.

One may also ask: how could the *brain* calculate the derivatives it needs to train time-lagged recurrences, since it cannot memorize its entire life history as

one time series, and it does not seem to sweep back through time in the same way that BPTT does? In [426,553], we describe an approach to approximating the required derivatives in forwards time, called the "Error Critic." Also, when systems need to learn over very long time intervals T, there may be a close relation between the kinds of multi-scale time representation we need for intelligent control [545] and the kind we need for memory management or "checkpointing." As a simple example, if $T = 2^n$, we could live with only n "memory records" in an exact BPTT, by allocating one record to hold $T/2$, one to hold $3T/4$ initially but later $T/4$, and so on, in a scheme similar to binary search. The brain may not be so limited in its memory capacity, but the organization of its information may lead to some interesting parallels.

Modular Structure, Higher-Order Derivatives and Eigenvalues

Up to now, I discussed how to calculate the ordered first derivatives of a scalar quantity of interest Y as depicted in Fig. 8, for systems which are ordered (Sect. 3.1) or recurrent (Sect. 3.2). But what if the system of interest has more than one output of interest? What if we need higher-order derivatives? For reasons of space, I cannot present all the extensions I have worked with, but I can give some examples and citations.

First, consider the example of Fig. 4. A supervised learning system usually has several outputs Y_1 through Y_n, forming a vector $\underline{Y}$. How can we apply the capability shown in Fig. 8 where there is only a single output of interest?

The SLS shown in Fig. 8 may be a neural network *or any other* system which may be represented by a vector-valued function $\underline{F}$:

$$\underline{\hat{Y}}(t) = \underline{F}(\underline{X}(t), W) \ , \tag{4}$$

where W represents the *set* $\{W_\alpha\}$ of weights or parameters to be adapted. There are many ways to adapt such systems. In "vanilla backpropagation" or in "basic backpropagation" [541, 544] in real-time, we adapt the weights W_α to reduce the square error of the prediction of $\underline{Y}(t)$ at the current time

$$E(t) = \sum_{i=1}^{n} \frac{1}{2} \left(\hat{Y}_i(t) - Y_i(t) \right)^2 \ . \tag{5}$$

For truly brain-like capability, it is very important to modify this approach by adding penalty terms (e.g., those of Phatak) and by accounting for the related issues addressed by various authors involving loss functions [448], robustness [31], empirical risk [518] or dynamic robustness [532, 552, 553] and by allowing for a kind of interplay between learning from current experience and learning from memory as in what I have called syncretism [553], which is related to Atkeson's memory-based learning.

In basic backpropagation, the scalar quantity of interest is $E(t)$. The system to be differentiated is not the SLS itself but the *combination* of the SLS

and (5). Thus we can apply the reverse method directly. As a practical matter, it is important to write clean modular code here. Modular code becomes ever more important as we work our way up to more complex applications.

In writing modular code, we would like to use a name for each variable as close as possible to the label we use in the mathematical papers that describe the system, but computer languages will not let us use "$\partial^+ E(t)/\partial W_\alpha$", for example, as a variable name. Thus I have used the shorthand notation "F_W_α", for example, to represent the $f$eedback to the quantity W_α, the ordered derivative of the current quantity of interest with respect to W_α. (In the AD community, people might use "ad_W_α" instead.)

For developing a modular version of basic backpropagation [541], I have shown how the reverse calculations can be split up into two parts. The first part, in the main program, calculates:

$$F\_\hat{Y}_i \equiv \frac{\partial^+ E(t)}{\partial \hat{Y}_i(t)} = \frac{\partial E(t)}{\partial \hat{Y}_i(t)} = \hat{Y}_i - Y_i \ , \tag{6}$$

for $i = 1$ to n. Then a dual subroutine, a dual or adjoint to the SLS, works back the implied ordered derivatives with respect to all of the inputs or weights. (To minimize run costs, we may sometimes code two versions or entries to the dual subroutine, one of which only calculates feedback to the weights, for cases where that is all we need.) The dual subroutine inputs the entire set of F_Y variables, but it implements a single reverse calculation aimed at a single scalar quantity of interest further downstream.

Only about 12 people have fully implemented structures like Fig. 7 so far, because it requires us to keep track of three main scalar quantities of interest, the specific error measure used in training the Critic, the error measure used in training the Model, and the estimate of J itself as used to train the Action network. People who use off-the-shelf neural network software without really understanding the reverse method find it difficult to keep track of the complexity. Often I explain the system by discussing how to adapt the three parts in three separate sections, so that I discuss only one error measure in each section. Many other explanations, examples, and applications appear in [479]. In my parts of [553], I give the more general case more explicitly, by using dual subroutines in the specification of algorithms, so that a user can select any mixture of neural networks, elastic fuzzy logic, or user-specified systems of equations for any of the components. I have sometimes wondered whether I should use notation like FJ_W_α, FEJ_W_α, and FEX_W_α to explicitly describe how systems like Fig. 7 require us to consider three quantities of interest at the same time. In 1986 [536], I described some early ideas for how one might implement capabilities like this as a user-friendly systems of commands in the SAS system. There are tutorial slides with text which progress from simple pattern recognition and data mining methods, through to diagnostics and time-series issues, through to many generations and types of methods for decision and control [547].

Chapter 10 of [553] discusses the issues which arise when we try to train "Models" (as in Fig. 7) which can predict partially observed systems over time, and chapter 13 discusses a stochastic extension (the "Stochastic Encoder/Decoder/Predictor") which may be thought of as a kind of nonlinear maximum likelihood factor analysis system in the special case where the Predictor is set to zero. Backwards differentiation is essential to making these kinds of complex capabilities workable in realistic computing systems or in realistic chip-level or distributed hardware implementations. Jose Principe at the University of Florida (`www.ece.ufl.edu/facultystaff/principe.html`) also has interesting ideas. Feldkamp and Prokhorov of Ford recently did benchmark studies where even simple TLRNs performed state estimation as well as more expensive "particle filter" methods and better than Extended Kalman Filters. Still, more work is needed to unify the pieces needed when unobserved variables are partly continuous and partly discrete.

As a further example, people sometimes use supervised learning to learn nonlinear relationships in a statistical way from real observed data, but supervised learning can also be useful as a way to develop a kind of reduced order model of a more complex model. For example, one may use it to *approximate* a large model running on a supercomputer by a neural network model which could fit on the $1-10 neural chip designed and tested by Raoul Tawel of the Jet Propulsion Laboratory and Mosaix LLC (funded by an NSF SBIR grant with encouragement from Ford). In such cases, however, you can get better results by developing an adjoint for the large model as well, so that the training set includes $\underline{X}(t)$, $\underline{Y}(t)$, and the Jacobian of $\underline{Y}(t)$ with respect to $\underline{X}(t)$ for each example t. One can minimize the augmented error function:

$$E(t) = \sum_{i=1}^{n} \frac{1}{2}\left(\hat{Y}_i(t) - Y_i(t)\right)^2 + \frac{1}{2}\sum_{j=1}^{m} C_j \sum_{i=1}^{n}\left(\frac{\partial^+ \hat{Y}_i}{\partial X_j} - \frac{\partial^+ Y_i}{\partial X_j}\right)^2, \qquad (7)$$

where the input vector $\underline{X}$ has m components, and the nonnegative weights C_j can be chosen in various ways.

I have called this method "Gradient Assisted Learning" (GAL). To minimize this error function, one must in effect calculate *its* derivatives, which involve ordered second derivatives. Chapter 10 of [553] discusses how to do so in some detail. (See also [538].) The easiest general approach is simply to note that the calculation of $E(t)$ is itself an ordered system, even though some of its intermediate calculations are *motivated* by derivative calculation. One can apply the reverse method directly to *that* ordered system. Similar issues arise in implementing a control method which I call Globalized Dual Heuristic Programming (GDHP), where I discussed (less clearly) how to get the second derivatives [533,534,537]. GDHP now seems most important as a way to handle decision or control problems where the actions $\underline{u}(t)$ include both discrete and continuous choices [479]. See [546] for some discussion of stability theory and links to control theory.

Sensitivity analysis and convergence of large models was a major application when I was at EIA, calling for many kinds of derivatives for many uses. See [534] and [535] for examples. For example, combining the reverse method with the Fadeev formulas for derivatives of eigenvalues and eigenvectors yielded interesting information. All of the methods here can provide important practical information in global modeling packages, such as one would use for long-range strategic planning where it is important to know the impact of current decisions or policies on long-term global outcomes, and to generate value measures or "shadow prices" to guide optimal decisions by tactical or distributed agents.

Solutions of ODEs with Removable Singularities*

Harley Flanders

University of Michigan, Ann Arbor, MI, USA
harley@umich.edu

Summary. We discuss explicit ODEs of the form $\dot{x} = R(t, x)$, where R is a polynomial or rational function, and the solution $x(t)$ has a removable singularity. We are particularly interested in functions built from elementary functions, such as $x(t) = t/\sin t$. We also consider implicit ODEs of the forms $P(t, x, \dot{x}) = 0$ and $P(t, x, \dot{x}, \ddot{x}) = 0$.

Key words: Ordinary differential equations, removable singularities, Taylor polynomial

1 Introduction

Automatic differentiation (AD) can be used to compute a Taylor polynomial approximation to a function $x(t)$ of a single variable at a base point t_0 to any desired degree. AD can handle functions with removable singularities, such as $x(t) = t^2/(1 - \cos t)$ at $t_0 = 0$. The standard convolution algorithm for computing the expansion of a quotient $H = F/G$ from $F = G * H$, given the expansions of F and G, must first be preprocessed by determining the lowest terms t^m and t^n appearing in F and G, respectively. Then if $m \geq n$, the algorithm, slightly modified, works.

AD also can be used to compute Taylor polynomial approximations to solutions of the IVP (Initial Value Problem) for a single ODE (Ordinary Differential Equation) or a system of ODEs of the form

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x}), \quad \mathbf{x}(t_0) = \mathbf{x}_0 \ .$$

It is known from work of Charney [104], Kerner [304], and Moore [378] that, by adding more dependent variables if necessary, the vector functions $\mathbf{f}$ may be restricted to polynomials (even polynomials of degrees at most 2).

* We are grateful to Alexander Gofen for suggesting this inquiry and for many helpful discussions and suggestions.

In this article we explore the question of whether a function with a removable singularity at t_0 may actually be the solution function of an ODE with polynomial or rational right side, so its Taylor approximations can be computed by AD. We also consider the same question for implicit ODEs of the form

$$P(t,\, x,\, \dot{x}) = 0\ ,$$

where P is a polynomial. Finally we discuss some explicit higher order ODEs.

2 Notation and Some Polynomial Algebra

All polynomials here have coefficients in the complex field $\mathbf{C}$. The letters T, X, Y, ... denote independent indeterminants over the field $\mathbf{C}$. They generate the polynomial ring

$$\mathbf{C}[T,\, X,\, Y,\, \ldots]\ .$$

This commutative ring has no zero divisors and is a *unique factorization domain.* That is, each non-constant polynomial factors into irreducible factors, unique up to nonzero constant multiples. (When we refer to the "only" irreducible polynomial with some property, "up to a nonzero constant factor" is understood.)

The letters x, y, ... denote functions of the complex variable t.

Lemma 1. *Let $x(t)$ be a non-constant periodic function, and let $P(T, X)$ be a polynomial such that $P(t,\, x(t)) = 0$. Then $P(T,\, X) = 0$. That is, P is the zero polynomial.*

Proof. Fix t_0. Then $P(T,\, x(t_0)) = 0$ is a polynomial with infinitely many zeros: $T = t_0 + np$, where p is a period of $x(t)$. Hence $P(T,\, x(t_0)) = 0$ identically, independent of t_0. As $x(t)$ is non-constant, it takes infinitely many values; hence $P(T,\, X) = 0$.

Lemma 2. *The same result holds for rational functions $R(T,\, X)$.*

3 Elementary Functions

3.1 Exponential Functions

Theorem 1. *The* only *irreducible polynomial $P(T, X, Y)$, such that the implicit differential equation*

$$P(t,\, x,\, \dot{x}) = 0$$

has solution

$$x(t) = \frac{e^t - 1}{t} \quad (x(0) = 1)$$

is

$$P(T,\, X,\, Y) = TY - TX + X - 1\ .$$

The only explicit *rational ODE satisfied by* $x(t)$ *is*

$$\dot{x} = \frac{tx - x + 1}{t}\ .$$

In particular, there does not exist a polynomial $P(T,\, X)$ *in* T *and* X *alone such that*

$$\dot{x} = P(t,\, x)\ .$$

Proof. Set $y = \dot{x}$ and differentiate $tx = e^t - 1$:

$$x + ty = e^t = tx + 1\ .$$

Hence $P(T,\, X,\, Y) = TY - TX + X - 1$ satisfies $P(t,\, x,\, \dot{x}) = 0$. Clearly P is irreducible, being of degree one in Y.

Suppose $Q(T,\, X,\, Y)$ is another polynomial such that $Q(t,\, x,\, \dot{x}) = 0$. Multiply Q by a sufficiently high power T^k of T, so the result has Y appearing only in products TY. Divide T^kQ by P, thought of as a polynomial in Y with coefficients in $\mathbf{C}[T,\, X]$:

$$T^kQ(T,\, X,\, Y) = P(T,\, X,\, Y) \cdot U(T,\, X,\, Y) + V(T,\, X)\ .$$

Replace T, X, and Y by t, x, and $\dot{x}$, with result $V(t,\, x) = 0$. This makes x into an algebraic, rather than transcendental, function, which is impossible. Consequently, $V(T,\, X) = 0$, so P divides T^kQ. By unique factorization, P divides Q, proving that P is the *only* irreducible polynomial with $P(t,\, x,\, \dot{x}) = 0$.

Another explicit rational ODE for X would imply another irreducible polynomial relation, which we know cannot be the case.

Remark 1. Suppose we wish to solve the implicit IVP

$$t\dot{x} - tx + x - 1 = 0, \quad x(0) = 1\ ,$$

but we do not know its solution in advance. Of course it can be solved by separating variables, but more to the point here is that it can be solved by AD. Actually, to get started, first differentiate the equation once and substitute $t = 0$, $x = 1$ to find $\dot{x}(0)$. Then repeatedly traverse the parse tree of the equation for successive terms of the solution's expansion. This works as usual for implicit ODEs without singularity.

This is related to Taylor series solutions of DAE (Differential Algebraic Equations); see Nedialkov and Pryce [406].

Theorem 2. *The only irreducible polynomial* $P(T,\, X,\, Y)$ *such that the implicit differential equation*

$$P(t,\, x,\, \dot{x}) = 0$$

has solution

$$x(t) = \frac{t}{e^t - 1} \quad (x(0) = 1)$$

is

$$P(T,\, X,\, Y) = TY + TX + X^2 - X\ .$$

The only explicit rational ODE satisfied by $x(t)$ is

$$\dot{x} = \frac{-tx + x - x^2}{t}\ .$$

In particular, there does not exist a polynomial $P(T,\ X)$ *such that*

$$\dot{x} = P(t,\, x)\ .$$

Proof. With notation $y = \dot{x}$ as in the previous proof,

$$(e^t - 1)x = t\ .$$

Differentiate, multiply by x, and substitute:

$$(e^t - 1)y + e^t x = 1, \quad (e^t - 1)xy + (e^t - 1)x^2 + x^2 = x, \quad ty + tx + x^2 = x\ .$$

An argument similar to that in the previous proof finishes this proof. (Alternative proof: apply Theorem 1 to $1/x$.)

Remark 2. What was said in Remark 1 applies here too.

3.2 Trigonometric Functions

Lemma 3. *If a polynomial $P(T,\, X,\, Y)$ satisfies $P(t,\, \cos t,\, \sin t) = 0$, then $P(T,\, X,\, Y)$ is a multiple of $Q = X^2 + Y^2 - 1$.*

Proof. Divide P by the quadratic $Q = Y^2 + (X^2 - 1)$ in Y; the remainder is linear in Y with coefficients polynomials in T and X:

$$P(T,\, X,\, Y) = QG + P_0(T,\, X) + P_1(T,\, X)Y\ ,$$

where $G = G(T, X, Y)$. Therefore

$$0 = P(t,\, \cos t,\, \sin t) = P_0(t,\, \cos t) + P_1(t,\, \cos t)\sin t\ .$$

Transpose and square:

$$P_0^2(t,\, \cos t) = P_1^2(t,\, \cos t)\sin^2 t = P_1^2(t,\, \cos t)(1 - \cos^2 t)\ .$$

It follows from Lemma 1 that

$$P_0^2(T,\, X) = P_1^2(T,\, X)(1 - X^2)\ .$$

If P_0 were not 0, then $1 - X$ would divide the left side an even number of times and the right side an odd number, clearly impossible by unique factorization. Hence $P_0 = P_1 = 0$, $P \equiv 0 \pmod{Q}$. That is, P is a multiple of Q.

Theorem 3. *There does not exist a rational function $R(T, X)$ such that the ODE*

$$\dot{x} = R(t, x)$$

is satisfied by $x(t) = \sin t$ or by $x(t) = \cos t$.

Proof. We use the notation $s = \sin t$, $c = \cos t$, and as above, $y = \dot{x}$. If $x = \sin t$, then

$$c = R(t, s) \ ,$$

which is impossible by Lemma 3. A similar argument works for $x(t) = \cos t$.

Theorem 4. *The only irreducible polynomial $P(T, X, Y)$ such that*

$$P(t, x(t), \dot{x}(t)) = 0 \quad \textit{for} \quad x(t) = \frac{\sin t}{t} \quad (x(0) = 1)$$

is

$$P(T, X, Y) = (X + TY)^2 + T^2X^2 - 1 \ .$$

There does not exist a rational function $R(T, X)$ such that the ODE

$$\dot{x} = R(t, x)$$

is satisfied by

$$x(t) = \frac{\sin t}{t} \ .$$

Proof. Differentiate $s = tx$:

$$c = x + ty, \quad (x + ty)^2 = c^2 = 1 - s^2 = 1 - t^2x^2 \ .$$

Therefore

$$P(T, X, Y) = (X + TY)^2 + T^2X^2 - 1 = 0$$

is satisfied by $(t, \ x, \ \dot{x})$. To prove P irreducible, write P as a quadratic in Y:

$$P = T^2Y^2 + (2TX)Y + (T^2X^2 + X^2 - 1) \ .$$

Were P reducible, a product of two factors linear in T, its discriminant

$$\Delta = (TX)^2 - T^2(T^2X^2 + X^2 - 1) = T^2(1 - T^2X^2)$$

would be the perfect square of a polynomial, which obviously it is not.

If there were an explicit rational ODE $\dot{x} = R(t, x)$, then a polynomial of the form $A(T, X)Y + B(T\,X)$ would be divisible by the irreducible P, which is impossible because P is quadratic in Y.

It remains to prove that if $Q(T, X, Y)$ is any nonzero polynomial such that $Q(t, x(t), \dot{x}(t)) = 0$, then P divides Q. To do so, write Q as a polynomial in Y with coefficients polynomials in T and X, and divide Q by P:

$$T^kQ = PU + A(T, X)Y + B(T, X) \ ,$$

where k is sufficiently high to eliminate denominators. Then $A(t, x)\dot{x} + B(t, x) = 0$, which we just ruled out. Hence $A = B = 0$, so $T^kQ = PU$. By unique factorization, Q divides P.

We can prove the following theorem in a similar way.

Theorem 5. *The only irreducible polynomial $P(T, X, Y)$ such that*

$$P(t, x(t), \dot{x}(t)) = 0 \quad \textit{for} \quad x(t) = \frac{t}{\sin t} \quad (x(0) = 1)$$

is

$$P(T, X, Y) = (X - TY)^2 - X^2(X^2 - T^2) \, .$$

There does not exist a rational function $R(T, X)$ such that the ODE

$$\dot{x} = R(t, x)$$

is satisfied by $x(t)$.

The three functions $(1 - \cos t)/t$, $(1 - \cos t)/(\frac{1}{2}t^2)$, and $\frac{1}{2}t^2/(1 - \cos t)$ have removable singularities at 0. We obtain the following theorem by similar methods as above. The normalizing factor $\frac{1}{2}$ is inserted because $1 - \cos t = \frac{1}{2}t^2 + \ldots$.

Theorem 6. *We have*

$x(t)$	*Irreducible $P(T, X, Y)$ such that $P(t, x, \dot{x}) = 0$*
$\frac{1}{2}\,(1 - \cos t)/t$	$(TY + X)^2 - TX(1 - TX)$
$(1 - \cos t)/(\frac{1}{2}t^2)$	$(TY + 2X)^2 + X(T^2X - 4)$
$(\frac{1}{2}t^2)/(\,1 - \cos t)$	$(TY - 2X)^2 + X^2(T^2 - 4X)$.

In each of these cases, there does not exist a rational function $R(T, X)$ such that $x(t)$ satisfies the explicit differential equation $\dot{x} = R(t, x)$.

Proof. The final statement follows in each case because the implicit polynomial ODE is irreducible and quadratic in Y.

The functions $(\tan t)/t$ and $t/(\tan t)$ have removable singularities at $t = 0$, but unlike the functions in Theorem 6, they satisfy explicit rational ODEs.

Theorem 7. *The only irreducible polynomial $P(T, X, Y)$ such that*

$$P(t, x(t), \dot{x}(t)) = 0 \quad \textit{for} \quad x(t) = \frac{\tan\, t}{t}$$

is

$$P(T, X, Y) = TY - T^2X^2 + X - 1 \, ,$$

and $x(t)$ satisfies the explicit rational ODE

$$\dot{x} = \frac{t^2x^2 - x + 1}{t} \, .$$

Proof. We have

$$x = \frac{\tan\ t}{t} = \frac{s}{ct}, \quad tx = \frac{s}{c}, \quad ctx = s\ .$$

Differentiate the third of these:

$$-stx + cx + cty = c, \quad -\frac{s}{c}tx + x + ty = 1, \quad -t^2x^2 + x + ty = 1\ .$$

Theorem 8. *The only irreducible polynomial* $P(T,\, X,\, Y)$ *such that*

$$P(t,\, x(t),\, \dot{x}(t)) = 0 \quad \textit{for} \quad x(t) = \frac{t}{\tan\ t}$$

is

$$P(T,\, X,\, Y) = TY + X^2 - X + T^2\ ,$$

and $x(t)$ *satisfies the explicit rational ODE*

$$\dot{x} = \frac{x - t^2 - x^2}{t}\ .$$

Proof. We have

$$x = \frac{t}{\tan\ t} = \frac{tc}{s}, \quad xs = tc\ .$$

Differentiate:

$$ys + xc = c - ts, \quad y + x\frac{c}{s} = \frac{c}{s} - t, \quad yt + x^2 = x - t^2\ .$$

The rest follows.

3.3 Inverse Trigonometric Functions

We simply list four results:

Theorem 9. *We have:*

$x(t)$	$P(T,\, X,\, Y)$	*Explicit Rational ODE*
$(\arcsin\ t)/t$	$(1-T^2)(X+TY)^2 - 1$	*None*
$t/(\arcsin\ t)$	$(X-TY)^2(1-T^2) - X^4$	*None*
$(\arctan\ t)/t$	$(X+TY)(1+T^2) - 1$	$\dot{x} = [1 - x(1+t^2)]/[t(1+t^2)]$
$t/(\arctan\ t)$	$(X-TY)(1+T^2) - X^2$	$\dot{x} = [x(1+t^2) - x^2]/[t(1+t^2)]$.

Proof. One will suffice, say $x = (\arctan\ t)/t$. We set

$$s = \sin(tx)\ , \quad c = \cos(tx)\ .$$

Then

$$tx = \arctan t, \quad t = \tan(tx), \quad s = tc$$

and

$$\dot{s} = (x + ty)c, \quad \dot{c} = -(x + ty)s \ .$$

Differentiate $s = tc$:

$$(x + ty)c = c - t(x + ty)s, \ x + ty = 1 - t^2(x + ty), \ (x + ty)(1 + t^2) = 1 \ .$$

This gives us P, and solving for Y gives the explicit rational ODE.

3.4 Hyperbolic Functions

Again, we summarize results in a table.

Theorem 10. *We have:*

$x(t)$	$P(T, X, Y)$	*Explicit Rational ODE*
$(\sinh t)/t$	$(X + TY)^2 - T^2X^2 - 1$	*None*
$t/(\sinh t)$	$(X - TY)^2 - X^4 - X^2T^2$	*None*
$(\tanh t)/t$	$X + TY + T^2X^2 - 1$	$\dot{x} = (1 - x - t^2x^2)/t$
$t/(\tanh t)$	$TY + X^2 - X - T^2$	$\dot{x} = (-x^2 + x + t^2)/t$
$(\operatorname{arcsinh} t)/t$	$(X + TY)^2(1 + T^2) - 1$	*None*
$t/(\operatorname{arcsinh} t)$	$(X - TY)^2(1 + T^2) - X^4$	*None*
$(\operatorname{arctanh} t)/t$	$(X + TY)(1 - T^2) - 1$	$\dot{x} = [1 - x(1 - t^{2)}]/[t(1 - t^2)]$
$t/(\operatorname{arctanh} t)$	$(X - TY)(1 - T^2) - X^2$	$\dot{x} = [x(1 - t^2) - x^2]/[t(1 - t^2)]$.

Proof. As before, one will suffice: $x = (\operatorname{arctanh} t)/t$. We set $s = \sinh(tx)$, $c = \cosh(tx)$. Then $t = \tanh(tx) = s/c$. We have

$$tc = s, \quad c + t(x + ty)s = (x + ty)c, \quad 1 + t^2(x + ty) = x + ty \ .$$

4 Other Functions

4.1 Review of Bessel Functions

We briefly review Bessel functions $J_n(t)$ for $n > 0$. See Abramowitz and Stegun [2] for details. The functions $J_n(t)$ satisfy an ODE with suitable initial data. We have

$$J_0\text{:} \quad t\ddot{x} + \dot{x} + tx = 0, \quad x(0) = 1, \quad \dot{x}(0) = 0 \ ,$$

$$J_1\colon \quad t^2\ddot{x} + t\dot{x} + (t^2 - 1)x = 0, \quad x(0) = 0, \quad \dot{x}(0) = \frac{1}{2} \ ,$$

and for $n > 1$:

$$J_n\colon \quad t^2\ddot{x} + t\dot{x} + (t^2 - n^2)x = 0, \quad x(0) = 0, \quad \dot{x}(0) = 0 \ .$$

Each of these is a singular ODE because of the t or t^2 multiplier of $\ddot{x}$. The solutions are regular at $t = 0$, indeed,

$$J_n(t) = \frac{t^n}{2^n} \sum_{k=0}^{\infty} \frac{(-1)^k t^{2k}}{4^k k!(n+k)!} \ .$$

4.2 The Function J_0

The Taylor expansion of J_0 can be found by AD. Substitute $x(t) = 1 + a_2t^2 + a_3t^3 + \ldots$ into the ODE:

$$(2a_2t + 6a_3t^2 + \ldots) + (2a_2t + 3a_3t^2 + \ldots) + (t + a_2t^3 + a_3t^4 + \ldots) = 0 \ .$$

This determines successively a_2, a_3, ...

Conjecture 1. The only irreducible polynomial $P(T, X, Y, Z)$ such that

$$P(t,\, x(t),\, \dot{x}(t),\, \ddot{x}(t)) = 0 \qquad \text{for} \qquad x(t) = J_0(t)$$

is

$$P(T,\, X,\, Y,\, Z) = TZ + Y + TX \ .$$

The function $J_0(t)$ does not satisfy any first order implicit polynomial ODE.

The two statements are closely related. For suppose $Q(T,\, X,\, Y,\, Z)$ is a polynomial such that $Q(t,\ x(t),\ \dot{x}(t),\, \ddot{x}(t)) = 0$. Eliminate Z:

$$T^kQ(T,\, X,\, Y,\, Z) = P(T,\, X,\, Y,\, Z)A(X,\, Y,\, Z) + B(T,\, X,\, Y) \ ,$$

so $B(t,\, x(t),\, \dot{x}(t)) = 0$. If we could conclude that $B(T,\, X,\, Y) = 0$, then P divides Q. To prove this appears a challenge, as do similar conjectures for other Bessel functions. Perhaps Ostrowski's [421] proof of a theorem of O. Hölder on Gamma functions offers an idea for attacking such questions.

4.3 Gamma Function

The Gamma function is defined by

$$\Gamma(t) = \int_0^{\infty} x^{t-1}e^{-x}\, dx$$

for $t > 0$. It extends as a meromorphic function to the whole complex t-plane with simple poles at all integers $t = n \leq 0$, and no other poles.

Hölder's theorem states: *The Gamma function does not satisfy an algebraic differential equation of any order.* There is a whole chapter on this theorem in an even older reference Nielsen [414]. The theorem has the interesting consequence that there is currently no way of using AD to compute a Taylor expansion of the Gamma function. But we do not know if there is an algebraic system of ODEs in which one component of a solution is the Gamma function.

5 Higher Order Equations

We explore other questions here. We ask if the function $x(t) = (\sin t)/t$ can satisfy an explicit polynomial ODE of higher order. We introduce a notation for successive derivatives:

$$x_0 = x(t), \quad x_1 = \dot{x}_0, \ \ldots, \ x_n = \dot{x}_{n-1} \ .$$

Theorem 11. *For each $n > 0$, there does not exist a polynomial*

$$P(T, X_0, X_1 \ , \ ..., X_{n-1})$$

such that

$$x_n = P(t, x_0, x_1, \ \ldots \ , x_{n-1}) \ .$$

Proof. For $n = 1$, this was proved in Theorem 4. We review part of that proof:

$$x_0 = \frac{s}{t}, \quad tx_0 = s, \quad tx_1 + x_0 = c \ .$$

Differentiate and eliminate s:

$$tx_2 + x_1 + x_1 = -s, \quad tx_2 + 2x_1 + tx_0 = 0 \ .$$

The polynomial $P_2 = TX_2 + 2X_1 + TX_0$ is clearly irreducible. If there were a polynomial ODE of the form $x_2 + F(t, x_0, x_1) = 0$, then

$$Q_2(T, X_0, X_1) = P_2 - T(X_2 + F(T, X_0, X_1))$$

would be a polynomial satisfying $Q_2(t, x, \dot{x}) = 0$, which we know is impossible unless $Q = 0$, so T divides P_2, which is impossible.

Differentiate again:

$$tx_3 + 3x_2 + tx_1 + x_0 = 0 \ .$$

Again, $P_3 = TX_3 + 3X_2 + TX_1 + X_0$ is irreducible which, in the same way, precludes an explicit polynomial ODE of the form $x_3 = F(t, x_0, x_1, x_2)$.

It is clear that repeated differentiation will result in

$$tx_n + Q(t,\, x_0,\, ...,\, x_{n-1}) = 0\ ,$$

where Q is a quadratic polynomial not divisible by t. Hence

$$P_n = TX_n + Q(T,\, X_0,\, ...,\, Xx_{n-1})$$

is irreducible, so an explicit polynomial ODE is impossible.

We look the corresponding result for one other of the functions we have studied above. We start with

$$x_0 = \frac{t}{e^t - 1}\ ,$$

and let x_1, x_2, etc. denote its successive derivatives.

Theorem 12. *For each $n > 0$, there does not exist a polynomial*

$$F(T,\, X_0,\, X_1,\ ...,\, X_{n-1})$$

such that

$$x_n = F(t,\, x_0,\, x_1,\ ...,\, x_{n-1})\ .$$

Proof. For $n = 1$, we know that $P(T,\, X,\, Y) = TY + TX + X^2 - X$ is the only irreducible polynomial such that $P(t,\, x_0,\, x_1) = 0$. This and the statement of Theorem 2 prove this theorem for $n = 1$. The same reasoning as in the proof of Theorem 11 establishes the result in general.

6 Open Questions

1) Do there exist polynomials $P(T,\, X,\, Y)$ and $Q(T,\, X,\, Y)$ and a regular function $y(t)$ in a neighborhood of 0 such that the system

$$\begin{cases} \dot{x} = P(t,\, x,\, y) \\ \dot{y} = Q(t,\, x,\, y)\ , \end{cases}$$

is satisfied by $x(t) = (\sin t)/t$ and $y(t)$? The same question may be asked for the functions $t/\sin t$, $(e^t - 1)/t$, $t/(e^t - 1)$ etc. We suspect the answer is "no" in all cases.

2) If the answer is indeed no, does there exist a system

$$\begin{cases} \dot{x}_1 = P_1(t,\, x_1,\, ...,\, x_n) \\ \dot{x}_2 = P_2(t,\, x_1,\, ...,\, x_n) \\ \qquad\qquad \cdots \\ \dot{x}_n = P_n(t,\, x_1,\, ...,\, x_n)\ , \end{cases}$$

with the P_j polynomials and a solution of functions $x_j(t)$ regular near $t = 0$, with $x_1(t) = (\sin t)/t$? The same question may be asked for the functions $t/\sin t$, $(e^t - 1)/t$, $t/(e^t - 1)$, etc.

Automatic Propagation of Uncertainties

Bruce Christianson[1] and Maurice Cox[2]

[1] University of Hertfordshire, Hatfield, United Kingdom
B.Christianson@herts.ac.uk
[2] National Physical Laboratories, Teddington, United Kingdom
Maurice.Cox@npl.co.uk

Summary. Motivated by problems in metrology, we consider a numerical evaluation program $y = f(x)$ as a model for a measurement process. We use a probability density function to represent the uncertainties in the inputs x and examine some of the consequences of using Automatic Differentiation to propagate these uncertainties to the outputs y. We show how to use a combination of Taylor series propagation and interval partitioning to obtain coverage (confidence) intervals and ellipsoids based on unbiased estimators for means and covariances of the outputs, even where f is sharply non-linear, and even when the level of probability required makes the use of Monte Carlo techniques computationally problematic.

Key words: Confidence interval, coverage, covariance, ellipsoid, GUM, kurtosis, metrology, singularities, Taylor series, unbiased estimators, validation

1 Introduction

Often we have a program which calculates the numerical values $y = f(x)$ of some outputs y from the values of the inputs x. For many applications the values of the inputs are not known with certainty. This may be because of a deficit in our knowledge, corresponding perhaps to indeterminacies in the measurement process, or it may be because the input values are themselves representatives of a population with a non-zero variance. These possibilities correspond naturally to the Bayesian and frequentist viewpoints respectively[3].

The evaluations of such uncertainties in variable values are referred to respectively as Type B and Type A evaluations of uncertainty in Clauses 2.3.3, 3.3.5 of the enormously influential methodology for uncertainty evaluation set out in the "Guide to the Expression of Uncertainty in Measurement" [245], published by ISO and popularly known as the GUM.

[3] As tool-writers we hope to design our product in such a way as to satisfy both camps that we have performed the correct calculation, and thus to escape involvement in the inevitable bickering about the interpretation and significance of the result.

We often wish to model the uncertainty in the values of the independent variables (inputs x) and to obtain corresponding estimates of the uncertainties in the values of the dependent variables (outputs y).

Interval analysis is one technique which may be used to do this. However in many applications, certainty is either not to be had, or comes at too high a price. For example, the value of a variable drawn from a normal distribution $N(\mu, \sigma)$ is theoretically unbounded, but in practice the coverage interval

$$[\mu - 5\sigma, \mu + 5\sigma]$$

will almost certainly suffice[4]. In other cases, certainty is not desired: for many applications in metrology, mean-centred 95% coverage intervals for the outputs are the information of primary interest.

Direct application of interval analysis to coverage intervals is not straightforward. Input value uncertainties are often correlated as a result of the processes used to obtain them, and we frequently desire to exploit correlation information about output uncertainties. In this paper we use a probability density function (pdf), as in the GUM, to model the input uncertainties, and we examine some of the consequences of using Automatic Differentiation (AD) to propagate these uncertainties to the outputs. Here we use AD terminology and notation, rather than that more usually followed by the uncertainty community. We hope this paper is a step toward drawing the two communities together.

The rest of this paper is organized as follows: in the next section we look at the case where f is linear, corresponding to Clause 8 of the GUM, and introduce the use of multivariate truncated Taylor series. In Sect. 3 we compare and contrast this approach with conventional interval analysis. In Sect. 4 we extend our Taylor series approach to cope with moderate non-linearities in f and show how to obtain unbiased estimators for uncertainty parameters in this case. Implementation issues are considered in Sect. 5. This section also considers how to truncate and partition distributions in order to cope with poles and other artifacts of severe non-linearity. In Sect. 6 we show how AD can be used to validate hypotheses about output distributions, and to construct coverage intervals at various levels of probability under a hierarchy of such assumptions. The final section includes some prospects for the future.

2 Linear Models

Many approaches to uncertainty modelling assume that the evaluated function f (known in the GUM as the *model* of a measurement) is linear, or very

[4] There always comes a point where events within the model become sufficiently unlikely, relative to significant events deliberately ignored by the model, that they can also prudently be disregarded. For example, most metrological analyses, do not consider the effect of the measuring apparatus being struck by a very small meteor at the crucial moment.

nearly linear, at least over sufficiently long intervals surrounding the anticipated values for the inputs[5].

Under this assumption, the straightforward approach is to represent all the program variables v_i as multivariate first order Taylor series, so that the variable

$$v = (v^{(0)}, v^{(1)}, \ldots, v^{(n)}) \quad \text{represents} \quad v^{(0)} + v^{(1)}\zeta_1 + \ldots v^{(n)}\zeta_n \ ,$$

where the ζ_i are independent random variables with zero mean and unit variance, so that

$$E(\zeta_i) = 0 \ ; \qquad E(\zeta_i^2) = 1 \ ,$$

where E denotes expectation.

Usually we will take ζ_i from either the normal distribution $N(0, 1)$ or from the student-t distribution with the appropriate number of degrees of freedom for the number of measurements involved. Other distributions including the uniform and the logarithmic distribution are also possible and are considered in what follows.

2.1 Uncorrelated Inputs

In the simplest case, where the uncertainties in the input values to the calculation are uncorrelated, we initialize the input variables x_i by setting

$$x_i = x_i^{(0)} + x_i^{(i)}\zeta_i \ ,$$

where

$$x_i^{(0)} = E\left[x_i\right] \quad \text{and} \quad \left(x_i^{(i)}\right)^2 = V\left[x_i\right] = E\left[(x_i - E[x_i])^2\right]$$

are the mean and variance respectively of x_i.

2.2 Correlated Inputs

In the more general case where the input uncertainties are correlated, we set

$$x_i = x_i^{(0)} + \sum_j x_i^{(j)}\zeta_j \ ,$$

where $x_i^{(j)} = r_{ij}$ are chosen so that $R = [r_{ij}]$ is the lower triangular decomposition of the input covariance matrix

$$V = RR^T = Cov[x_1, \ldots, x_n] \ .$$

[5] The GUM makes this assumption explicitly [245, Clause G.6.1] as a basis for the uncertainty budgeting (evaluation and expression) procedure set out in Clause 8 of the GUM.

In this case we have $E[x_i] = x_i^{(0)}$, and

$$\begin{aligned}\text{Covariance } (x_i, x_j) &= E\left[(x_i - E[x_i])(x_j - E[x_j])\right] \\ &= E\left[\sum_{k,\ell} x_i^{(k)} x_j^{(\ell)} \zeta_k \zeta_\ell\right] = \sum_{k,\ell} x_i^{(k)} x_j^{(\ell)} E\left[\zeta_k \zeta_\ell\right] \\ &= \sum_k x_i^{(k)} x_j^{(k)} = \sum_k r_{ik} r_{jk} = v_{ij} \ .\end{aligned}$$

since by independence $E[\zeta_k \zeta_\ell] = \delta_{k\ell}$.

2.3 Single Output

Automatic Differentiation [21, 110, 229, 302] can be used to propagate the numerical coefficients of the Taylor terms through the calculation $y = f(x)$. If the mainstream GUM assumptions [342] of a linear function model are satisfied, then to the required degree of approximation we will have

$$y = y^{(0)} + \sum_j y^{(j)} \zeta_j \ ,$$

with the mean and variance of y being given by

$$E[y] = y^{(0)} \quad \text{and} \quad V[y] = \sum_j (y^{(j)})^2 \ ,$$

respectively. In the case where the ζ_i are normal, the pdf for y is also normal, and the calculated value for $V[y]$ can be used directly to construct the required coverage interval centred upon $y^{(0)}$.

2.4 Multiple Outputs

For a linear model with several outputs, AD gives

$$y_i = y_i^{(0)} + \sum_j y_i^{(j)} \zeta_j \ ,$$

so

$$E[y_i] = y_i^{(0)}, \text{ and } \quad \text{Cov}(y_i, y_j) = \sum_k y_i^{(k)} y_j^{(k)} \ .$$

These values can be used to construct the required coverage ellipsoid; the values $y_i^{(k)}$ effectively give the covariance of the outputs in a factored form, which in the normal case can be used to construct the ellipsoid directly.

3 Contrast with Interval Analysis

This section describes a simple linear example, intended to illuminate some differences between the approach taken in this paper and that of conventional interval analysis. Consider the case where the uncertainties in the inputs x_1 and x_2 are modelled by independent (uncorrelated) normal distributions with zero mean and variances $1.0, 0.01$ respectively, and suppose $y_1 = x_1 + x_2$. Clearly taking 97.5% coverage intervals for x_1 and x_2 and applying interval analysis will give a 95% coverage interval $[-2.47, +2.47]$ for y_1, but not an optimal one. For example taking a 96% coverage interval for x_1 and a 99% coverage interval for x_2 will give the tighter 95% interval $I = [-2.31, +2.31]$ for y_1. However the variances of the independent inputs add to 1.01, which is therefore the variance of y_1, so our approach directly gives $y_1 \in [-1.97, +1.97]$ with 95% probability.

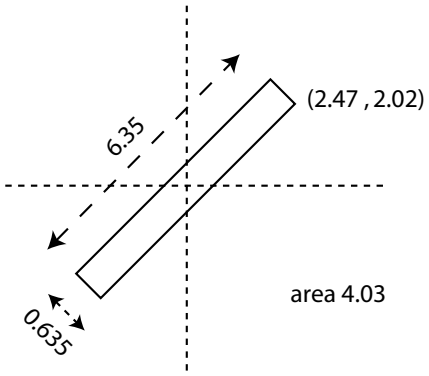


Fig. 1. The rectangle J.

Now suppose $y_2 = x_1 - x_2$. The square $I \times I$ is a 95% coverage box for $y = (y_1, y_2)$ with area 21.3, although the corresponding square with half-side 1.97 is not. But the thin rectangle J (Fig. 1) aligned at 45 degrees to the axes,

$$J = \{(y_1, y_2) : |y_1 + y_2| \leq 4.49, |y_1 - y_2| \leq 0.45\} ,$$

is a 95% coverage box and has the much smaller area 4.03. The optimal 95% ellipsoid E, given by the method of the previous section,

$$E = \{(y_1, y_2) : (y_1 + y_2)^2 + 100(y_1 - y_2)^2 \leq 23.96\} ,$$

has an area of just under 3.77.

Even in the exactly linear case, if rigorous bounds are sought for the output coverage intervals, then there are benefits to using the approach described here in conjunction with conventional interval analysis, rather than relying upon a naive use of the latter.

The method of this paper can be used to construct a sensible hypothesis for the rigorous approach to verify. The benefit of using the approach described here to "precondition" the hypothesis for rigorous validation is even more pronounced in the case of correlated inputs or, as we now discuss, non-linear evaluation functions f.

4 Nonlinear Models

The assumption of linearity of f over the relevant coverage interval is frequently not justified. In other cases, (approximate) linearity is an hypothesis which we wish to use our model to confirm, rather than a matter of blind faith. In these cases, a non-linear model of the effects of evaluating the function f is required.

In the case where the function model is significantly non-linear, the $y_i^{(0)}$ are generally biased estimators for the y_i. In other words, we can no longer assume $E[y_i] = y_i^{(0)}$. Indeed, in the non-linear case the outputs may not even be monotone functions of the inputs over the intervals in question, nor need the $y_i^{(0)}$ be maximum likelihood estimators for the y_i even in the smooth monotone case[6].

Although maximum likelihood estimators are appropriate for Bayesian inferences such as data assimilation, for many purposes we also require unbiased estimates of quantities associated with the outputs. In particular if we wish to construct coverage intervals (or ellipsoids) from a pdf model for y then we would like to have unbiased estimates for the relevant moment coefficients in order to construct percentile points of the cumulative output pdf.

Such unbiased estimators can be approximated accurately by using AD to propagate higher order Taylor terms.

4.1 Higher Order Program Variables

We can redefine the program variables to represent a pyramid of coefficients corresponding to a higher order truncated multivariate Taylor series, so that the variable

$$v_i = (v_i^{(0)}, (v_i^{(j)})_{0\le j\le n}, (v_i^{(jk)})_{0\le j\le k\le n}, (v_i^{(jk\ell)})_{0\le j\le k\le \ell\le n})$$

represents the truncated Taylor series

$$v_i^{(0)} + \sum_j v_i^{(j)}\zeta_j + \sum_{j\le k} v_i^{(jk)}\zeta_j\zeta_k + \sum_{j\le k\le \ell} v_i^{(jk\ell)}\zeta_j\zeta_k\zeta_\ell \ .$$

Initializing the input variables to be first order Taylor series as before, and using forward AD for general truncated multivariate Taylor series, we obtain

[6] Although in the smooth monotone case, if p and $q = p/f'$ are the pdf for x and y respectively, then at the maximum likelihood value for y, we have $p' = q \cdot f''$.

$$y = y^{(0)} + \sum_j y^{(j)}\zeta_j + \sum_{j\le k} y^{(jk)}\zeta_j\zeta_k + \sum_{j\le k\le \ell} y^{(jk\ell)}\zeta_j\zeta_k\zeta_\ell + \dots \ .$$

4.2 Taking Expectations

In general by independence of the ζ_i we have

$$E[\zeta_j^p\zeta_k^q\zeta_\ell^r] = E[\zeta_j^p]\cdot E[\zeta_k^q]\cdot E[\zeta_\ell^r] \quad \text{for } j<k<\ell \ ,$$

and we can usually evaluate terms such as $E[\zeta_j^p]$ from our knowledge of the distribution from which the ζ are drawn. For example for ζ in $N(0,1)$, we have

$$E[\zeta^{2p}] = (2p-1)E[\zeta^{2(p-1)}], \quad E[\zeta^{2p-1}] = 0 \quad \text{for } p>0 \ .$$

Corresponding moments can be pre-calculated for other initial distributions.

4.3 Unbiased Output Mean

Using these reductions, we see for example that an unbiased estimator of y is

$$E[y] = y^{(0)} + \sum_j y^{(jj)} + S_j y^{(jjj)} + K_j y^{(jjjj)}$$

$$+ \sum_{j<k} y^{(jjkk)} + \text{ 5th order terms} \ ,$$

where S_j is the skew $E[\zeta_j^3]$, and K_j is the kurtosis $E[\zeta_j^4]$. In the usual case where the ζ_j are symmetric, all S_j and all fifth order terms are zero, and the estimator is therefore accurate to order five. If ζ_j is normal then $K_j = 3$.

4.4 Unbiased Output Variance

Unbiased estimators for other quantities can be obtained by taking expectations of other variables. For example, an unbiased estimator for the variance of y is

$$V[y] = E\left[(y-E[y])^2\right]$$

$$= E\left[\left(\sum_j y^{(j)}\zeta_j + y^{(jj)}(\zeta_j^2-1) + \sum_{j<k} y^{(jk)}\zeta_j\zeta_k + \sum_{j\le k\le \ell} y^{(jk\ell)}\zeta_j\zeta_k\zeta_\ell + \dots\right)^2\right]$$

$$= \sum_j (y^{(j)})^2 + 2S_j y^{(j)}y^{(jj)} + (K_j-1)(y^{(jj)})^2 + 2K_j y^{(j)}y^{(jjj)}$$

$$+ \sum_{j<k} (y^{(jk)})^2 + 2y^{(j)}y^{(jkk)} + 2y^{(jjk)}y^{(k)} + \dots \ ,$$

where ζ_j has skew S_j and kurtosis K_j. When ζ_j is normal, $S_j = 0$ and $K_j = 3$.

4.5 Unbiased Output Covariances

Similarly in the case of more than one output, the covariances correspond to inner products:

$$\mathrm{Cov}[y_p, y_q] = E[(y_p - E[y_p])(y_q - E[y_q])]$$

$$\begin{aligned}
&= \sum_j y_p^{(j)} y_q^{(j)} + S_j(y_p^{(jj)} y_q^{(j)} + y_p^{(j)} y_q^{(jj)}) \\
&+(K_j - 1) y_p^{(jj)} y_q^{(jj)} + K_j(y_p^{(j)} y_q^{(jjj)} + y_p^{(jjj)} y_q^{(j)}) \\
&+ \sum_{j<k} y_p^{(jk)} y_q^{(jk)} + y_p^{(j)} y_q^{(jkk)} + y_p^{(jkk)} y_q^{(j)} + y_p^{(jjk)} y_q^{(k)} + y_p^{(k)} y_q^{(jjk)} + \dots
\end{aligned}$$

4.6 Adjoint Expectations

Expectations of the adjoints $\bar{x}_i = \partial y / \partial x_i$ can also be interpreted directly as sensitivities of parameters of the output uncertainties with respect to parameters of the input uncertainties. In the case of a single output y and uncorrelated inputs x_i, and writing $\mu(z)$ and $\sigma(z)$ respectively for the mean and standard deviation of the distribution modelling the uncertainty corresponding to the program variable z, we have for example that

$$E(\bar{x}_i) = \frac{\partial \mu(y)}{\partial \mu(x_i)}, \quad E(\zeta_i \bar{x}_i) = \frac{\partial \mu(y)}{\partial \sigma(x_i)},$$

$$\frac{E(y\bar{x}_i) - E(y)E(\bar{x}_i)}{\sigma(y)} = \frac{\partial \sigma(y)}{\partial \mu(x_i)}, \quad \frac{E(y\zeta_i\bar{x}_i) - E(y)E(\zeta_i\bar{x}_i)}{\sigma(y)} = \frac{\partial \sigma(y)}{\partial \sigma(x_i)}, \dots .$$

5 Implementation with Automatic Differentiation

All the expressions which we have considered, including those for unbiased estimators, can be evaluated automatically by an appropriately enhanced AD tool. It is straightforward to add intrinsics to an AD tool to specify the moments of the ζ_i. Independent variables can be defined to be of `type Normal, Student(n), Uniform etc`. A new operator `E` is defined which evaluates the expectation of a program variable. Further intrinsic functions can be defined in terms of `E`, such as

```
Cov (x,y) = E((x-E(x))*(y-E(y))
K(x) = E((x-E(x))**4)
Bias (x) = term0(x) - E(x)
```
etc.

Reverse AD can be used instead of forward to calculate the multivariate Taylor series coefficients. This makes the accumulation process more efficient, but requires slightly more care in the definitions of the intrinsics: the terms of each series must be assembled into a variant data type, prior to being combined into the expression to which the intrinsic is applied.

Similar remarks apply to the use of interpolation schemes such as those of Bischof et al.[7] [58] or Neidinger [410]. In the case of symmetric distributions the majority of cross-terms do not contribute to expectations, and so need not be computed: in particular, as remarked earlier, a fourth order estimate is automatically accurate to fifth order in this case.

5.1 Convergence and Singularities

The order of Taylor series actually required for a given level of accuracy depends on the non-linearity of f. As an extreme example, if

$$y = \exp x^2 \quad \text{with} \quad x = \zeta \in N(0,1) \ ,$$

then the expectation of y is unbounded. Of course, using a truncated distribution for x will give a correct coverage interval for y even in this case: for this example we could restrict ζ to the range $[-5, 5]$, with appropriate changes to the distribution moments. However it is useful that the AD-based model can signal automatically any potentially catastrophic non-linearity of f.

Singularities in f corresponding to plausible values of the input variables can also be dealt with by truncation, but similar difficulties can arise from imaginary poles of f. For example, the smooth function

$$y = \frac{1}{1+x^2} \quad \text{with } x = \zeta \in N(0,1)$$

has finite expectation, but the poles at i and $-i$ mean that the Taylor series in ζ for y has only unit radius of convergence. In such cases the simplistic approach of taking expectations term by term and summing produces a divergent series, even though the expectation of the infinite sum may be finite.

This underlines the remark made by Louis Rall during his 2004 talk in Chicago, that AD is not really a local operation. We describe the function f as *almost-linear*, relative to a given input uncertainty pdf and a desired probability of coverage for the outputs, when all poles of f are sufficiently distant from the mean, and the radius of convergence about the mean is sufficiently large, as to allow the use of a single truncated distribution upon each of the ζ_i.

[7] Andreas Griewank (anonymous communication) points out that this would allow a fifth order estimator of n output variable values y to be obtained at a cost of order $n^2/2$ univariate fourth-order Taylor expansions, which is about one third of the cost of computing the full order four tensor.

However in the example of $1/(1+x^2)$, truncation sufficient to ensure convergence would rule out altogether the possibility of obtaining a 95% coverage interval. Harley Flanders (personal communication) proposes the use of a *partitioning* approach to provide a general solution for cases like this, and we describe one such approach below.

5.2 The Partition Approach to Expectations

The partition approach to evaluating expectations works as follows. Partition the distribution for the ζ_i into boxes $B_j = \prod_i I_{i,j}$ in such a way that only finite boxes have non-negligible probability. By refining if necessary, ensure that distance from the centre $(c_{i,j})_i$ of each finite box to any pole of f is large compared to the length of the correponding radius of the box. For each non-negligible box B_j, recalculate the Taylor coefficients for y in terms of powers of $\xi_i = \zeta_i - c_{i,j}$.

Now, using the independence of the ξ_i, and our prior knowledge of the values of the integrals of the ξ_i^k with respect to the pdf for each element of the partition, apply the expectation operator to each box separately and then sum the results. This gives the correct result even for functions which are not almost-linear. Provided standard partitions are used, integrals for the ξ_i^k over $I_{i,j}$ can be pre-calculated and stored in tabular form.

The use of interval methods for optimization requires a similar form of partitioning to that advocated here. More accuracy can be obtained by using more Taylor terms or by allowing the partition to be refined.

6 Validation of Uncertainty Models

Numerical values of bias terms can be used to determine the validity of the GUM assumptions or to validate the non-linear model being used to construct the coverage intervals for the outputs. This can be done either as an alternative or as a supplement to Monte Carlo simulation [138, 246]. In the case where a non-linear multistage model is being used, higher order terms of the first stage outputs can be used directly to initialize the second stage inputs.

The generalized output values from an AD-based tool of this type can be used to construct many different coverage intervals, depending not only on the level of probability required but also on the strength of the assumptions made by the uncertainty model.

6.1 Single Output

For a single output variable y, let $I_s(y)$ be the interval defined by

$$I_s(y) = [E[y] - s \cdot \sigma(y), E[y] + s \cdot \sigma(y)] \,, \qquad \text{where } \sigma(y) = \sqrt{E[(y - E[y])^2]} \,.$$

Then $I_{4.48}(\mathrm{y})$ is a 95% coverage interval for y without any assumptions whatsoever on the distribution of y, by Chebyshev's inequality.

However if the kurtosis of y is known to be less than 4.0, an assumption which can be verified by using AD to compute the relevant expectation[8], then taking $s = 3.0$ gives a 95% coverage interval for y. This is because, setting $z = (y - E(y))/\sigma(y)$ and letting p denote the pdf of z, we have that

$$s^4 \cdot P(|z| > s) \leq \int z^4 p(z)\, dz \ .$$

Under the single assumption that the kurtosis is less than 4.0, which we can verify numerically by checking that $E(z^4) \leq 4.0$, setting $s = 3.0$ gives $P(y \notin I_{3.0}(y)) \leq 0.05$, and $s = 4.48$ gives a 99% coverage interval for y.

6.2 The Hyperbolic Cosine Transform

More ambitiously, we have

$$\cosh st \cdot P(|z| > s) \leq \int \cosh tz \cdot p(z)\, dz \ .$$

For z normal the value of the integral is $\exp \frac{1}{2}t^2$. Setting $t = s$ and validating, for a particular value of s and a 'nearly normal' z, the single verifiable[9] assumption that

$$E(\cosh sz) \leq 1.5 \exp \frac{1}{2}s^2$$

gives

$$P(|z| > s) \leq 3 \exp -\frac{1}{2}s^2 \ , \text{ etc.}$$

Estimates of this form are particularly useful for relatively high values of s, corresponding to a requirement for high levels of probability for which the use of Monte Carlo Methods is problematic. For example setting $s = 4.0$ with the hyperbolic cosine hypothesis gives a 99.9% coverage interval with no further assumptions about the distibution of z.

If more assumptions are made about the moments of y, then tighter coverage intervals can be constructed. If the uncertainly in y is known (or assumed) to be normal then s can be reduced to 1.96 for 95%, 2.58 for 99%, and 3.29 for 99.9%, and similarly for other distributions.

---

[8] Naive use of GUM simply assumes *inter alia* that the kurtosis of y is exactly 3.0 if the ζ_i are normal.

[9] From an implementation point of view, it is important to ensure that the Taylor series developed for cosh contains no odd terms, rather than being defined in terms of exp.

6.3 Multiple Outputs

Similar observations to the single output case hold for the case of $n > 1$ outputs. If

$$\mathrm{Cov}(y) = E((y - E(y))(y - E(y))^T) = RR^T$$

is the AD-calculated covariance for the outputs, then we can define

$$z = R^{-1}(y - E(y))$$

for the *calculated* values of R. Certainly (for example)

$$s^4 n^2 \cdot P(|z| > s\sqrt{n}) \leq \int z^4 p(z)\, dz = E((z^2)^2)\ ,$$

and we can calculate the value of $E((z^2)^2)/n^2$. Under the GUM assumptions, we have $E((z^2)^2)/n^2 \approx 1 + 2/n$, so we can attempt to verify the hypothesis that this value is less than 3.0 (say). In this case, we have

$$P(|z| > s\sqrt{n}) \leq 3.0/s^4\ ,\ \text{etc.}$$

7 Way Ahead

We have shown how to use AD to propagate Taylor series corresponding to uncertainty distributions through functions modelling measurements. A novel point is that the use of AD allows assumptions about the output moments to be validated[10] before a coverage interval at a given level of probability is built. This could be extended to Laurent series. Future work may also include the incorporation of rounding errors and errors arising from the inexact solution of intrinsic equations, using the methods of Iri [112,281,291]. A very desirable feature would be the incorporation of interval methods to provide rigorous bounds on the errors in the calculated expectation values. The systematic use of partitioning could also allow automatic validation of control flow changes (such as `if` statements) via partitions of coverage intervals on intermediate values of program variables.

[10] We recommend using AD to validate the assumption that the kurtosis is less than some given value, preferably an hypothesized value significantly larger than the calculated one. This is different from using the calculated value to construct the coverage interval. It is the former procedure which we are recommending here.

High-Order Representation of Poincaré Maps

Johannes Grote, Martin Berz, and Kyoko Makino

Department of Physics and Astronomy, Michigan State University, East Lansing, MI, USA
{grotejoh,berz,makino}@msu.edu

Summary. A method to obtain polynomial approximations of Poincaré maps directly from a polynomial approximation of the flow or dynamical system for certain types of flows and Poincaré sections is presented. Examples for the method and its computational implementation are given.

Key words: Differential algebra, Poincaré map, dynamical systems.

1 Introduction

Poincaré maps are a standard tool in general dynamical systems theory to study qualitative properties of a dynamical system, e.g. the flow generated by an ordinary differential equation, most prominently the asymptotic stability of periodic or almost periodic orbits. A Poincaré map essentially describes how points on a plane S (the Poincaré section) which is transversed by such an orbit $\mathcal{O}$ (the reference orbit) and which are sufficiently close to $\mathcal{O}$ get mapped back onto S by the flow. The two key benefits in this approach are that long-term behavior of the the flow close to $\mathcal{O}$ can be analyzed through the derivative of the Poincaré map at the intersection point of S and $\mathcal{O}$, which is available after just one revolution of $\mathcal{O}$, and that the dimensionality of the problem has been reduced by one, since the Poincaré map is defined on S and neglects the "trivial" direction of the flow perpendicular to the surface.

In the numerical treatment of these problems one is faced with the question of which numerical representations of a flow are particularly favorable in the sense that they easily allow the computation of corresponding Poincaré maps for a given reference orbit and Poincaré section. In this paper we will show that high-order polynomial approximations of the flow, which have been obtained either by automatic differentiation of an ODE solver with respect to initial conditions or using differential algebraic (DA) tools as in [38,44], allow a direct deduction of polynomial approximations of Poincaré maps of a certain type. We focus on the case where the flow under consideration has been generated

by an ODE. The proposed algorithm is a part of an extended method for the computation of rigorous interval enclosures of the polynomial approximation of the Poincaré map discussed here.

2 Overview of DA Tools

The DA tools necessary to appreciate the method are described in detail in [38]. However, we wish to review briefly the two most important applications of DA-methods used for the problem discussed here: the DA-integration method employed to obtain high-order polynomial approximations $\varphi(x_0, t)$ of the flow and the functional inversion tools necessary in later steps of the algorithm.

2.1 DA-Integration of ODEs

First we tackle the problem of obtaining a polynomial approximation of the dependence on initial conditions of the solution of the initial value problem

$$\dot{x}(t) = f(x(t), t), \quad x(0) = X_0 + x_0 \ , \tag{1}$$

where $f : \mathbb{R}^\nu \supset U^{\text{open}} \longrightarrow \mathbb{R}^\nu$ is given as a composition of intrinsic functions which have been defined in DA-arithmetic. This also entails that f exhibits sufficient smoothness to guarantee existence and uniqueness of solutions for all initial conditions. The vector $X_0 \in \mathbb{R}^\nu$ is constant, and the midpoint of the domain box $D = [-d_1, d_1] \times \ldots \times [-d_\nu, d_\nu]$ for the small relative initial conditions $x_0 \in D$. Typical box widths d_i are of the order 10^{-2} to 10^{-8}. The polynomial approximation $\varphi(x_0, t)$ of the flow of (1) we desire is an expansion in terms of the independent time coordinate t and the initial conditions x_0 relative to X_0. The representation of this approximation is a DA-vector storing the expansion coefficients up to a prespecified order n in a structured fashion.

The standard procedure of a Picard iteration yields a polynomial approximation of the solution of (1) after repeated application of a Picard operator on the initial conditions. This iteration in general increases the order of the expansion by at least one in every step. Since a DA-vector can store coefficients up to order n, we expect that the iteration converges after finitely many steps in the DA case.

Accordingly, the Picard operator in the DA-computation is defined by

$$\mathcal{C}(\cdot) := (X_0 + x_0) + \partial_{\nu+1}^{-1} f(\cdot) \ ,$$

where f is computed in DA-arithmetic, and $\partial_{\nu+1}^{-1}$ is the *antiderivation operator*, essentially the integration with respect to the ν+1st variable t. With a suitable definition of a contraction in the DA case, $\mathcal{C}$ is a contracting operator, and fixed-point theorems exist which guarantee that repeated applications of $\mathcal{C}$ on the initial condition DA-vector representation $x(0) = X_0 + x_0$ converge to the DA-vector solution $\varphi(x_0, t)$ of (1) in finitely many steps. After this iteration has converged, the time step is substituted for the time variable which yields the final solutions only in terms of the initial conditions x_0.

2.2 Functional Inversion Using DA-Arithmetic

Next we review the functional inversion method used to obtain the inverse $\mathcal{M}^{-1}$ of a function $\mathcal{M}$, or rather a DA-vector which stores the coefficients of $\mathcal{M}^{-1}$ up to the desired order. Assume we are given a smooth map $\mathcal{M} : \mathbb{R}^\nu \longrightarrow \mathbb{R}^\nu$ s.t. $\mathcal{M}(0) = 0$, and its linearization M is invertible at the origin. This assures the existence of a smooth inverse $\mathcal{M}^{-1}$ in a neighborhood of the origin. If we write $\mathcal{M} = M + \mathcal{N}$, where $\mathcal{N}$ is the nonlinear part, and insert this into the fundamental condition $\mathcal{M} \circ \mathcal{M}^{-1} = \mathcal{I}$, we obtain the relation

$$\mathcal{M}^{-1} = M^{-1} \circ (\mathcal{I} - \mathcal{N} \circ \mathcal{M}^{-1})$$

and see that the desired inverse $\mathcal{M}^{-1}$ is a fixed point of the operator $\mathcal{C}(\cdot) := M^{-1} \circ (\mathcal{I} - \mathcal{N} \circ \cdot)$. Since $\mathcal{C}$ is a contraction, the existence of the fixed point $\mathcal{M}^{-1}$ of $\mathcal{C}$ is verified, and $\mathcal{M}^{-1}$ can be obtained through repeated iteration of $\mathcal{C}$, beginning with the identity $\mathcal{I}$. Also in this case the iteration converges to $\mathcal{M}^{-1}$ in finitely many steps, which is intuitively clear: If at one iteration step $\mathcal{M}^{-1}$ is determined up to order m, then $\mathcal{C}(\mathcal{M}^{-1})$ is determined at least up to order $m + 1$, since $\mathcal{N}$ is purely nonlinear.

3 Description of the Method

3.1 Preliminary Remarks

We begin our discussion with the assumption that (1) exhibits a periodic solution $\varphi(X_0, t)$ which starts on a suitable Poincaré section and returns after a period T, which has been determined, e.g. by a high-order Runge-Kutta integration. As described in the previous section, there exist DA-arithmetic integration methods which allow us to transport the domain box $X_0 + D$, where $D = [-d_1, d_1] \times ... \times [-d_\nu, d_\nu]$, through one cycle of the period. In the last time step we keep the full expansion of the final solution $\varphi(x_0, t)$ in terms of the variables x_0 and the time t. The problem of constructing the Poincaré map has thus been reduced to the construction of a map which projects the set $\{\varphi(x_0, T) : x_0 \in D\}$ to the surface S.

We want to consider as large a class of surfaces as possible as Poincaré sections. A suitable assumption is that the Poincaré section $S \subset \mathbb{R}^\nu$ is given in terms of a function $\sigma : \mathbb{R}^\nu \longrightarrow \mathbb{R}$ as $S := \{x \in \mathbb{R}^\nu : \sigma(x) = 0\}$. Since the function σ also needs to be expressed in terms of elementary functions available in the computer environment for DA arithmetic, it is necessarily smooth, and hence so is the surface S. This contains most surfaces of practical interest, in particular the most common case where S is an affine plane $S := \{x \in \mathbb{R}^\nu : x_1 = c\}$ for some $c \in \mathbb{R}$; then $\sigma(x) = x_1 - c$.

3.2 Construction of the Crossing Time

The goal of the next step is to derive an expression for the crossing time $t_c(x_0)$ at which the trajectory starting at the initial condition $x_0 \in D$ crosses the

surface S. From an analytic standpoint the existence of such a time $t_c(x_0)$ is only guaranteed locally at X_0, but since usually D is small and σ and the vector field f are regular, the crossing time can often be defined for all $x_0 \in D$. Once we have obtained a DA-vector representing $t_c(x_0)$, then $\mathcal{P}(x_0)$ can be found easily by inserting the crossing time into the flow

$$\mathcal{P}(x_0) := \varphi(x_0, t_c(x_0)) \;, \tag{2}$$

where the right hand side is evaluated in DA-arithmetic. We proceed by constructing an artificial function $\psi(x_0, t)$ by

$$\begin{aligned} \psi_k(x_0, t) &:= x_k \,\forall\, k \in \{1, ..., \nu\} \\ \psi_{\nu+1}(x_0, t) &:= \sigma(\varphi(x_0, t)) \;. \end{aligned}$$

The value $t_c(x_0)$, depending on the variables x_0, is determined by the condition

$$\sigma(\varphi(x_0, t_c(x_0))) = 0 \;, \tag{3}$$

and ψ contains both the constraint (3) and the independent variables x_0. Because of (3), $t_c(x_0)$ satisfies

$$\psi(x_0, t_c(x_0)) = (x_0, 0) \;.$$

If ψ is invertible at $(x_0, t_c(x_0))$ we can evaluate

$$\psi^{-1}(x_0, 0) = \psi^{-1}(\psi(x_0, t_c(x_0))) = (x_0, t_c(x_0))^T$$

and immediately extract the DA-vector representation of $t_c(x_0)$ in terms of x_0 in the last component. However, here the invertibility of ψ at the point $(x_0, t_c(x_0))$ is guaranteed by the transversality of the flow at S. This leads to the definition of

$$t_c(x_0) := \psi_{\nu+1}^{-1}(x_0, 0) \;,$$

which allows us to obtain the final Poincaré map by construction (2).

3.3 Summary of the Algorithm

To conclude the presentation of the method, we summarize the algorithm:

1. Determine the period T approximately for the periodic orbit.
2. Choose a suitable Poincaré section S.
3. Obtain a DA-vector representation of the solution $\varphi(x_0, t)$ for one period T. Preserve the full expansion in x_0 and t in the last step.
4. Set up and invert the auxiliary function ψ using DA functional inversion to obtain a DA-vector representation of ψ^{-1}.
5. Resolve $t_c(x_0) := \psi^{-1}(x_0, 0)$.
6. Obtain $\mathcal{P}(x_0) := \varphi(x_0, t_c(x_0))$.

4 Examples

The method described above has been implemented in the COSY Infinity [43] programming language, which supports the DA-vector data type and its operations. The code lists for the example calculations are available upon request from the authors. The COSY output lists the Taylor expansion coefficients of $t_c(x_0)$ and $\mathcal{P}(x_0)$ sorted by order, with the last five columns showing the respective powers of the expansion variables $x_{0,1}$ through $x_{0,4}$ and t.

4.1 The Planar Kepler Problem

As a first example we study the planar Kepler problem,

$$\begin{aligned}\dot{x}_1 &= x_2 \\ \dot{x}_2 &= -\frac{x_1}{(x_1^2+x_3^2)^{3/2}} \\ \dot{x}_3 &= x_4 \\ \dot{x}_4 &= -\frac{x_3}{(x_1^2+x_3^2)^{3/2}}\ ,\end{aligned}$$

where we choose the initial conditions $x(0) = X_0 + x_0$. Here $X_0 = (0,-1,1,0)^T$ is the midpoint of the domain D for x_0 and the starting point for the reference orbit, and $D = [-10^{-4}, 10^{-4}]^4$. The reference orbit is periodic with a period $T = 2\pi$. The Poincaré section on which we project is $S := \{x \in \mathbb{R}^4 : x_1 = 0\}$.

The Kepler problem serves as a good test case, since not only the reference orbit, but all orbits originating in $X_0 + D$ are periodic. This means every trajectory crosses S at the same point where it originated after one revolution. Thus the ith component $\mathcal{P}_i(x_0)$ of the Poincaré map is the identity with respect to the expansion variable $x_{0,i}$. In the following, we show the results of the expansion coefficients of the crossing time $t_c(x_0)$ and the final components of $\mathcal{P}(x_0)$ after an 18th order computation. The result for the crossing time $t_c(x_0)$ is

```
  I  COEFFICIENT            ORDER EXPONENTS
  1  0.2731858587386304E-13   0   0 0  0 0  0
  2  0.3905776925772165E-03   1   1 0  0 0  0
  3  -.7362216057939549E-02   1   0 1  0 0  0
  4  0.7362216057939555E-02   1   0 0  1 0  0
  5  -.1651789475691988E-16   1   0 0  0 1  0
...
197  0.1730482119203762E-18   7   1 1  3 2  0
198  0.1875627603432954E-18   7   0 2  3 2  0
199  -.1075817722340857E-18   7   1 2  1 3  0
200  0.1018021833717433E-18   7   1 1  2 3  0
201  0.1149402904334884E-18   7   0 2  2 3  0,
```

where this is the expansion around the period $T = 2\pi$. We see that $t_c(x_0)$ has no constant part up to roughly machine precision and scales almost linearly with the variables $x_{0,1}$ to $x_{0,4}$, the second order terms are already significantly smaller.

Next we display the final result for the first component $\mathcal{P}_1(x_0)$. If the computed Poincaré map projects to the surface S, then its constant part must vanish. Indeed this is what we see up to leftover terms of negligible magnitude:

```
I  COEFFICIENT            ORDER EXPONENTS
1  -.2488159282559507E-19   2   0 2  0 0  0
2  -.2482865326639168E-19   2   0 0  2 0  0.
```

Finally we give the result for $\mathcal{P}_2(x_0)$. In this case we restricted the Poincaré map $\mathcal{P}$ to S by setting $x_{0,1} = 0$. We see that $\mathcal{P}_2(x_0)$ preserves the first order identity in the $x_{0,2}$ variable up to a scaling factor of 10^{-4}, the domain half width. This rescaling supports validated computation.

```
I  COEFFICIENT            ORDER EXPONENTS
1  -.9999999999999988       0   0 0  0 0  0
2  0.1000000000000023E-03   1   0 1  0 0  0
3  -.1695071177393755E-17   1   0 0  1 0  0
4  -.1850766989750646E-18   1   0 0  0 1  0
5  0.3388131789017201E-19   2   0 2  0 0  0
...
34 0.2649298927416580E-19   7   0 2  5 0  0
35 -.2439099199128762E-19   7   0 5  1 1  0
36 0.5124187266775200E-19   7   0 4  2 1  0
37 0.5711061765338940E-19   7   0 3  3 1  0
38 0.2743591506567708E-19   7   0 2  4 1  0.
```

The results for $\mathcal{P}_3(x_0)$ and $\mathcal{P}_4(x_0)$ are similar to $\mathcal{P}_2(x_0)$.

4.2 A Muon Cooling Ring

In accelerator physics, a muon cooling ring is a simple representation of a device made up of solenoids, RF cavities, and hydrogen absorbers that is designed to 'cool' a muon particle beam, i.e. reduce the volume of phase space the beam occupies; for details see [5, 349, 422]. Its equations of motion are

$$\begin{aligned}
\dot{x}_1 &= x_3 \\
\dot{x}_2 &= x_4 \\
\dot{x}_3 &= x_4 - \frac{\alpha}{\sqrt{x_3^2 + x_4^2}}\, x_3 + \frac{\alpha}{\sqrt{x_1^2 + x_2^2}}\, x_2 \\
\dot{x}_4 &= -x_3 - \frac{\alpha}{\sqrt{x_3^2 + x_4^2}}\, x_4 - \frac{\alpha}{\sqrt{x_1^2 + x_2^2}}\, x_1 \ ,
\end{aligned}$$

where $\alpha \in [0, 1]$ is the cooling parameter ($\alpha = 1$ being the fastest cooling), and we consider the initial values $x(0) = X_0 + x_0$ with $X_0 = (0, 1, 1, 0)^T$ and $D = [-10^{-4}, 10^{-4}]^4$. The centerpoint X_0 lies on a periodic orbit of the form $\varphi(0, t) =$

$(\cos(t), -\sin(t), -\sin(t), -\cos(t))$ with a period of $T = 2\pi$. However, no other orbit originating in the box $X_0 + (D\backslash\{0\})$ is periodic, but instead is slowly pulled towards the invariant solution $\varphi(0,t)$ with an asymptotic phase. This should be visible from the eigenvalues of the Poincaré map for the section $S := \{x \in \mathbb{R}^4 : x_1 = 0\}$. Again, we show the results for the crossing time $t_c(x_0)$ and the components of $\mathcal{P}(x_0)$ after an 18th order computation with a choice of $\alpha = 0.1$. For the crossing time $t_c(x_0)$ we obtain:

```
  I  COEFFICIENT             ORDER EXPONENTS
  1  0.2041481078081152E-13   0   0 0  0 0  0
  2  -.4881805626857354E-02   1   1 0  0 0  0
  3  0.1420453958184906E-03   1   0 1  0 0  0
  4  -.1420453958184956E-03   1   0 0  1 0  0
  5  0.1804749589528265E-02   1   0 0  0 1  0
...
 66  -.1728272108226884E-14   4   0 0  2 2  0
 67  -.1380573416990237E-15   4   1 0  0 3  0
 68  0.5285647002734696E-17   4   0 1  0 3  0
 69  0.4817909040734367E-14   4   0 0  1 3  0
 70  0.6371702406569035E-16   4   0 0  0 4  0.
```

This is the expansion around the period $T = 2\pi$. Inserting this into the flow $\varphi(x_0, t)$ and restricting $\varphi(x_0, t)$ to S yields that for $\mathcal{P}_1(x_0)$ all expansion coefficients are zero. For the component $\mathcal{P}_2(x_0)$ we get:

```
  I  COEFFICIENT             ORDER EXPONENTS
  1   1.000000000000000       0   0 0  0 0  0
  2  0.7300927710720673E-04   1   0 1  0 0  0
  3  0.2699072289279350E-04   1   0 0  1 0  0
  4  -.5747288684637408E-06   1   0 0  0 1  0
  5  -.1174080052084289E-08   2   0 2  0 0  0
...
 31  -.3092279715866550E-16   4   0 1  1 2  0
 32  0.5188804928611400E-16   4   0 0  2 2  0
 33  0.7267117814015809E-18   4   0 1  0 3  0
 34  -.3574718550317267E-17   4   0 0  1 3  0
 35  -.4144960345719535E-17   4   0 0  0 4  0;
```

for $\mathcal{P}_3(x_0)$:

```
  1   1.000000000000000       0   0 0  0 0  0
  2  -.3619742360532049E-19   1   0 1  0 0  0
  3  0.1000000000000003E-03   1   0 0  1 0  0
  4  -.1167493942379190E-08   2   0 2  0 0  0
  5  0.2334987884782753E-08   2   0 1  1 0  0
...
 30  -.9698631674909994E-17   4   0 1  1 2  0
 31  0.3092775905473012E-16   4   0 0  2 2  0
 32  0.7881782248585311E-18   4   0 1  0 3  0
 33  -.3750583012239952E-17   4   0 0  1 3  0
 34  -.4166566008579194E-17   4   0 0  0 4  0;
```

and for $\mathcal{P}_4(x_0)$:

```
  I  COEFFICIENT             ORDER EXPONENTS
  1  0.2775557561562886E-16   0   0 0  0 0  0
  2  0.5747288684637220E-06   1   0 1  0 0  0
  3  -.5747288684637250E-06   1   0 0  1 0  0
  4  0.7306674999405320E-04   1   0 0  0 1  0
  5  0.4038817974889429E-12   2   0 2  0 0  0
...
 31  0.3038943603851730E-18   4   0 1  1 2  0
 32  -.5017859446188993E-17   4   0 0  2 2  0
 33  0.2175112902784127E-18   4   0 1  0 3  0
 34  -.2064794267030274E-16   4   0 0  1 3  0
 35  0.1176604677592068E-17   4   0 0  0 4  0.
```

If we compute the eigenvalues of the linear part $P(x_{0,2}, x_{0,3}, x_{0,4})$ of $\mathcal{P}(x_0)$, when viewed as a function of $x_{0,2}$, $x_{0,3}$, and $x_{0,4}$, we get $\lambda_1 = 1$ and $\lambda_{2,3} \approx 0.73038 \pm i(0.00574)$. λ_1 is connected to the identity in the linear part of $\mathcal{P}_3(x_0)$ with respect to $x_{0,3}$, and the magnitude of less than 1 for λ_2 and λ_3 is a consequence of the cooling action in x_2- and x_4-directions, the desired effect of the muon cooler.

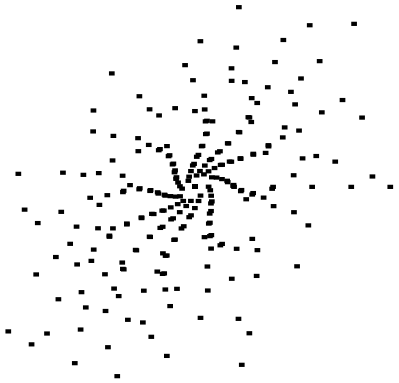


Fig. 1. Tracking of six particles for the first 50 turns in the muon cooling ring.

We use the Poincaré map for a detailed study of the dynamics by using it to iterate an ensemble of initial conditions through repeated orbits around the ring. This is an approach frequently followed in beam physics [38, 157], since it replaces time consuming integration of ODEs for one revolution by mere application of a polynomial. The results are shown in Fig. 1, showing the behavior in the transverse $x_{0,2}$-$x_{0,4}$-plane of an ensemble of six particles launched on the $x_{0,2}$-axis at the points $n \cdot 4$ cm for $n = 1, \ldots, 6$. The tracking picture is obtained after repeated application of the transverse components of the Poincaré map for 50 turns, and clearly exhibits the desired cooling effect near the attracting center.

Computation of Matrix Permanent with Automatic Differentiation*

Koichi Kubota

Department of Information and System Engineering, Chuo University,
Tokyo, Japan
kubota@ise.chuo-u.ac.jp

Summary. With a well-known formulation of matrix permanent by a multivariate polynomial, algorithms for the computation of the matrix permanent are considered in terms of automatic differentiation, where a succinct program with a C++ template for the higher order derivatives is described. A special set of *commutative quadratic nilpotent elements* is introduced, and it is shown that the permanent can be computed efficiently as a variation of implementation of higher order automatic differentiation. Given several ways for transforming the multivariate polynomial into univariate polynomials, six algorithms that compute the value of the permanent are described with their computational complexities. One of the complexities is $\mathrm{O}(n2^n)$, the same as that of the most popular Ryser's algorithm.

Key words: Matrix permanent, Taylor series, higher-order derivatives, commutative quadratic nilpotent elements, Padre2

1 Introduction

The permanent of an n-dimensional square matrix $A = (a_{ij})$ is defined as

$$\mathrm{per}(A) \equiv \sum_{\sigma} \prod_{k=1}^{n} a_{k\sigma(k)} = \sum_{\sigma} a_{1\sigma(1)} a_{2\sigma(2)} \cdots a_{n\sigma(n)} \ , \tag{1}$$

where σ runs over all the permutations of $\{1, \cdots, n\}$ [314, 369]. The definition of the permanent is quite similar to that of the determinant, but there are no sign changes. The value of the permanent is related to combinatorial features of a system represented by a matrix. Since the computation of the permanent was shown to be #P-complete [515], the direction of research on the permanent was moved to the computation of approximate values [4, 171, 189, 286, 340].

The most popular and efficient algorithm for computing the exact value of the permanent is Ryser's algorithm, whose computational complexity is $\mathrm{O}(n2^n)$ [314,

* This work is supported by Grant-in-Aid for Scientific Research of JSPS [16560054], Chuo University Overseas Research Program, Chuo University Personal Research Grant, and Chuo University Grant for Special Research.

369]. Another algorithm with multivariate polynomials is also popular. It is a simple way to compute the exact value of the permanent with symbolic manipulation systems.

Starting from the well-known definition of the permanent as a coefficient of a multivariate polynomial, we derive several univariate polynomials and give methods for computing of the coefficients of the polynomials. First, giving a succinct program with a C++ template, we introduce a straightforward algorithm for the permanent with automatic differentiation (AD) that can be used for differentiating multivariate polynomials. Second, providing several ways to transform the multivariate polynomial into univariate polynomials, we consider numerical computations of the polynomials where the computation of the permanent can be represented as that of a Taylor series. A uniqueness condition is required for the transformation. Third, to represent naturally the sparsity of the higher order derivatives needed by the computation of the permanent, we introduce "commutative quadratic nilpotent elements" for the evaluation of the coefficient of the multivariate polynomials. The elements can be implemented by operator overloading in a manner similar to the implementation of automatic differentiation. Finally, we describe the computational complexities of several algorithms for computing the permanent. One of them is $\mathrm{O}(n2^n)$, the same as that of the Ryser's algorithm.

2 Formulation

After brief overview of the formulation of the computation of the permanent, we propose transformations of the multivariate polynomial into univariate polynomials.

2.1 Multivariate Polynomial

According to the definition of the permanent (1), it is well known that the permanent can be represented as the coefficient of the term $x_1x_2\cdots x_n$ in the (expanded) multivariate polynomial f defined by

$$f(x_1, x_2, \cdots, x_n) \equiv \prod_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_j = \prod_{i=1}^{n}(a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n)\ . \tag{2}$$

Thus,

$$\mathrm{per}(A) = \frac{\partial^n}{\partial x_1 \partial x_2 \cdots \partial x_n} f(x_1, x_2, \cdots, x_n) \tag{3}$$

gives a method for computing the value of the permanent, and it can be computed naturally and easily with AD.

A well known algorithm due to Ryser's computes the same coefficient of $f(x_1, \cdots, x_n)$:

$$\mathrm{per}(A) = (-1)^n \sum_{s\in 2^{\{1,2,\ldots,n\}}} (-1)^{|s|} \prod_{i=1}^{n}\sum_{j\in s} a_{ij}\ , \tag{4}$$

where s runs over all the subsets of the integer set $\{1, \cdots, n\}$. This form (4) evaluates f at 2^n points represented by $\{0, 1\}^n$, and its computational complexity is $\mathrm{O}(n2^n)$ with a Gray-code technique. That is the best known algorithm for computing the exact value of the permanent.

2.2 Transformation to Univariate Polynomial

Here, considering n non-negative integers α_1, α_2, $\cdots$, α_n, we substitute x^{α_i} for x_i and transform $f(x_1, \cdots, x_n)$ in (2) into $f(x)$ defined by

$$f(x) \equiv f(x^{\alpha_1}, x^{\alpha_2}, \cdots, x^{\alpha_n}) = \prod_{i=1}^{n} (a_{i1}x^{\alpha_1} + a_{i2}x^{\alpha_2} + \cdots + a_{in}x^{\alpha_n}) \ . \tag{5}$$

The following condition is very important for choosing the integers α_i $(i = 1, \cdots, n)$ to compute the permanent as the coefficient of $f(x)$.

Condition 2.1 Uniqueness condition on $\{\alpha_1, \cdots, \alpha_n\}$: *Choose one of the values n times from $\{\alpha_1, \cdots, \alpha_n\}$ and denote the k-th chosen value by β_k. Note that β_k may be equal to β_ℓ $(\ell \neq k)$. Denote the sum of α_i by α_* $(= \sum_{i=1}^{n} \alpha_i)$.*

If and only if $\sum_{k=1}^{n} \beta_k = \alpha_*$, the sequence β_1, $\cdots$, β_n is a permutation of α_1, $\cdots$, α_n. In other words, when $\sum_{k=1}^{n} \beta_k = \alpha_*$, there are no indices k and $\ell(\neq k)$ such that $\beta_k = \beta_\ell$. Obviously, α_k's should be distinct from each other. Thus, without loss of generality, we can assume $\alpha_1 < \alpha_2 < \cdots < \alpha_n$.

When the above uniqueness condition is satisfied on $\{\alpha_1, \cdots, \alpha_n\}$, the coefficient of x^{α_*} of $f(x)$ in (5) gives the value of per(A). For example, when $\{\alpha_1, \alpha_2, \alpha_3\} = \{1, 2, 4\}$, α_* is 7, and 7 is different from $1+1+1$ $(=3)$, $1+1+2$ $(=4)$, $1+1+4$ $(=6)$, $1+2+2$ $(=5)$, $1+4+4$ $(=9)$, $2+2+2$ $(=6)$, $2+2+4$ $(=8)$, $2+4+4$ $(=10)$ and $4+4+4$ $(=12)$. Thus, $\{\alpha_1, \alpha_2, \alpha_3\} = \{1, 2, 4\}$ satisfies the uniqueness condition. For another set $\{\alpha_1, \alpha_2, \alpha_3\} = \{1, 2, 3\}$, $\alpha_* = 6$, and it is equal to $2+2+2$ so the set $\{1, 2, 3\}$ does not satisfy the uniqueness condition.

We show some sets that satisfy this uniqueness condition.

Lemma 1. *The set $\{\alpha_i : \alpha_i = 2^{i-1}\}_{i=1}^{n}$ satisfies the uniqueness condition, so the value of the permanent is given by the coefficient of x^{2^n-1} of the polynomial*

$$f(x) \equiv \prod_{i=1}^{n} (a_{i1}x^{2^0} + a_{i2}x^{2^1} + \cdots + a_{in}x^{2^{n-1}}) \ . \tag{6}$$

Proof. For $\alpha_i = 2^{i-1}$, $\alpha_* = \sum_{i=1}^{n} \alpha_i = 2^n - 1$. Denote the sequence $[\beta_1, \cdots, \beta_n]$ in Condition 2.1 by B and assume that $\sum_{k=1}^{n} \beta_k = 2^n - 1$. Since α_* is odd, at least one $\alpha_1(=1)$ is contained in B, i.e., there is an index ℓ_1 such that $\beta_{\ell_1} = \alpha_1(=1)$. Consider $\alpha_* - \alpha_1 = 2^n - 2 = 2 * (2^{n-1} - 1)$. We must represent $\alpha_* - \alpha_1$ by the sum of $n-1$ values repeatedly chosen from α_1, $\cdots$, α_n. When we can choose a set whose sum equals $\alpha_* - \alpha_1$, there must be a subset whose sum is equal to 2, i.e., there is one index ℓ_2 such that $\beta_{\ell_2} = \alpha_2(=2)$, or there are two indices ℓ_2 and ℓ_3 such that $\beta_{\ell_2} = \alpha_1(=1)$ and $\beta_{\ell_3} = \alpha_1(=1)$. Because, if there is no subset whose sum not equal to 2, the value $2^n - 2$ cannot be represented with a multiple of 4. Note that β_{ℓ_1} should not be used in these subsets. Therefore, the set B should contain "three α_1's" or "one α_1 and one α_2."

By a similar observation, there must be the n subsets whose sums are 1, 2, 4, $\cdots$, 2^{n-1}, respectively. We can use only n numbers, so that we should assign one number to construct each subset, i.e., $\{1\}$ for 1, $\{2\}$ for 2, $\cdots$, $\{2^{n-1}\}$ for 2^{n-1}. That is, there are no β_k's whose sum is equal to α_* except such that the $\beta_1, \cdots, \beta_n$ is a permutation of $\alpha_1, \cdots, \alpha_n$.

Corollary 1. *The set* $\{\alpha_i\}_{i=1}^n$ *defined by* $\alpha_i = 2^{i-1} - 1$ $(i = 1, \cdots, n)$ *satisfies the uniqueness condition 2.1 and* $\sum_{i=1}^n \alpha_i = \alpha_* = 2^n - n - 1$.

Corollary 2. *For integer* p (> 1)*, the set* $\{\alpha_i\}_{i=1}^n$ *defined by* $\alpha_i = p^{i-1}$ *satisfies the uniqueness condition 2.1.*

3 Methods

In this section, after a brief explanation of automatic differentiation (AD), we introduce *commutative quadratic nilpotent elements*, and we outline an example implementation of commutative quadratic nilpotent elements.

3.1 Automatic Differentiation

AD [42, 136, 227] is a well known method to compute derivatives of functions whose values are computed by programs. Here, we discuss some topics of AD related to this work. Given a program that computes a value of a function, AD is a method for generating a program that computes the values of the derivatives of the function. There are mainly two styles for its implementation, one is a precompiler, and the other is a library program that works through (a kind of) operator overloading.

There are also several methods to compute higher order derivatives with many variables. One of the simplest ways is repeated applications of a precompiler to the derived programs. Another simple way is the use of nested definitions of classes with the C++ template mechanism. An example of a succinct program for the permanent is given in Fig. 1. In this program we define a class "`ad<double>`" for differentiation, then declare another class "`ad<ad<double> >`"[2] for the second order differentiation, and so on. This small program computes the permanent of an $n \times n$ $(n = 5)$ matrix. When the size n, is large it may consume much memory. However, it is a simple way to compute the permanent as well as the higher order derivatives of a multivariate function. For the computation of the permanent, only definitions of the addition and the multiplication are described. However, definitions of other operators, elementary functions and utility interfaces are easily given in this manner. We could generate computational graphs for the higher order derivatives for the reverse mode AD. For a univariate polynomial, the computation of the Taylor series is the method for computing the higher order derivatives.

3.2 Commutative Quadratic Nilpotent Element

The target value as the permanent is only the coefficient of the monomial with the term $x_1 x_2 \cdots x_n$ in the multivariate polynomial $f(x_1, \cdots, x_n)$ defined by (2). In the intermediate computation of the expansion of f, there is no need to compute any monomials containing quadratic powers of x_i's, i.e., terms ${x_1}^{i_1} {x_2}^{i_2} \cdots {x_n}^{i_n}$ with $i_k \geq 2$ at least one k can be deleted. Thus, we can consider an algebra where every monomial with a quadratic power of x_i in its term is replaced with zero.

[2] The space is important, i.e., "`ad<ad<double>>`" does not work.

```
/* Permanent with higher order AD; by kubota@ise.chuo-u.ac.jp */
#include <iostream>
template <class T>
struct ad {
  T v, dv;
  ad(const T v0=T(),const T dv0=T()):v(v0),dv(dv0){}
};
template <class T>
ad<T> operator+(const ad<T> x, const ad<T> y) {
  return ad<T>(x.v+y.v, x.dv+y.dv);}
template <class T>
ad<T> operator*(const ad<T> x, const ad<T> y) {
  return ad<T>(x.v*y.v,x.dv*y.v+x.v*y.dv);}
template <class T>
ad<T> operator*(double x, const ad<T> y){
  return ad<T>(x*y.v,x*y.dv);}
typedef ad<double> ad1;
typedef ad<ad1> ad2;  /* typedef ad<ad<double> > ad2;*/
typedef ad<ad2> ad3;
typedef ad<ad3> ad4;
typedef ad<ad4> ad5;
double m[5][5]={
 {1,2,3,4,5},{2,3,4,5,6},{3,4,5,6,7},{4,5,6,7,8},{5,6,7,8,9}};
int main() {
  int n=5;
  ad5 x[n],f;
  x[0].dv.v.v.v.v=1;
  x[1].v.dv.v.v.v=1;
  x[2].v.v.dv.v.v=1;
  x[3].v.v.v.dv.v=1;
  x[4].v.v.v.v.dv=1;
  f.v.v.v.v.v=1;
  for(int i=0;i<n;i++) {
    ad5 s;
    for(int j=0;j<n;j++) s=s+m[i][j]*x[j];
    f=f*s;
  }
  std::cout<<"permanent="<<f.dv.dv.dv.dv.dv<<std::endl;
}
```

Fig. 1. Simple AD program for computing the permanent.

That is, introducing a set of n elements ξ_1, ξ_2, $\cdots$, ξ_n such that they are (i) commutative $\xi_i\xi_j = \xi_j\xi_i$ $(i = 1, \cdots, n, j = 1, \cdots, n)$ and (ii) quadratic nilpotent $\xi_i^2 = 0$ $(i = 1, \cdots, n)$, we consider a computation of polynomials with these ξ_i's.

With this algebra (or elements), we can eliminate the useless monomials in the expansion of the polynomial f in (2), saving computational time and space. For example, with an implementation of this algebra, we can represent the expanded term of $(a_1\xi_1 + a_2\xi_2 + \cdots + a_n\xi_n)^k$ by the $\binom{n}{k}$ non-zero monomials. The number of non-zero monomials is $\binom{n+k-1}{k}$ with the conventional symbolic manipulators. When $n = 3$, with three elements $\{\xi_1, \xi_2, \xi_3\}$, we have

$$
\begin{aligned}
&(a\xi_1 + b\xi_2 + c\xi_3)^2 \\
&\quad = a^2{\xi_1}^2 + b^2{\xi_2}^2 + c^2{\xi_3}^2 + 2(ab\xi_1\xi_2 + ac\xi_1\xi_3 + bc\xi_2\xi_3) \qquad (7)\\
&\quad = 2(ab\xi_1\xi_2 + ac\xi_1\xi_3 + bc\xi_2\xi_3)\ , \qquad (8)
\end{aligned}
$$

where the numbers of monomials in (7) and (8) are $\binom{3+2-1}{2} = 6$ and $\binom{3}{2} = 3$, respectively.

To implement this algebra, we can use a method similar to that of AD with operator overloading. A monomial is represented as a pair of the coefficient and a term, and an expression (or a sum) of monomials is represented as a set of the pairs (details and example codes are described in [319]). An example implementation is outlined below.

(1) Define a class `monomial` for monomials of the commutative quadratic nilpotent element. A monomial $c \cdot {\xi_1}^{b_1}{\xi_2}^{b_2} \cdots {\xi_{n-1}}^{b_{n-1}}{\xi_n}^{b_n}$ $(b_k \in \{0, 1\})$ is represented by a pair of a coefficient and a bit-vector that indicates the exponents of the term, i.e. $(c, [b_n b_{n-1} \cdots b_2 b_1])$. For example, $(1.5 \cdot \xi_1\xi_3)$ and $(2.0 \cdot \xi_2\xi_3)$ of 3 commutative nilpotent elements are represented by $(1.5, [101])$ and $(2.0, [110])$, respectively.

(2) Define a class `cqne` for expressions of the commutative quadratic nilpotent elements. For example, a set of one monomial $1.5 \cdot \xi_1\xi_3$ can be regarded as an expression $E_1 = \{(1.5, [101])\}$. Another set of one monomial $2.0 \cdot \xi_2\xi_3$ is also an expression $E_2 = \{(2.0, [110])\}$. Thus an expression $1.5 \cdot \xi_1\xi_3 + 2.0 \cdot \xi_2\xi_3$ is represented by a set union of them, i.e. $\{(1.5, [101]), (2.0, [110])\}$.

(3) Define addition and multiplication for `monomial`. Addition is defined only for two monomials with the same term. The result of the addition is the pair of the sum of the coefficients and the same bit-vector of the summands (augends). The multiplication of two monomials is equal to a monomial represented by the pair of the product of the coefficients and the logical OR'ed bit-vector of the multiplicands. The result is 0 if there is a ξ_i that appears in both terms of the multiplicands.

(4) Define addition and multiplication for `cqne`. Since instances of `cqne` represent expressions (polynomials), their addition and multiplication are defined naturally. The addition $E_1 + E_2$ is a set union of the sets as the representations of E_1 and E_2. If there are two monomials with the same terms, they are reduced to one monomial by the addition of instances of `monomial`. The multiplication $E_1 * E_2$ is a set of monomials constructed by a sum of component-wise multiplications of E_1 and E_2.

With this implementation, for two instances E_1 and E_2 of class `cqne`, the computational complexities of $E_1 * E_2$ and $E_1 + E_2$ are as follows. Denoting the number of elements of the set of E_1 by n_1, that of E_2 by n_2, a multiplication $E_1 * E_2$ requires $n_1 \cdot n_2$ multiplications of instances (monomials) of class `monomial` and $n_1 \cdot n_2$ insertion operations of an item (a monomial) into a set (an expression). An addition $E_1 + E_2$ requires at most $\min(n_1, n_2)$ additions of monomials and n_1 (n_2) search/insertion operations of an item into a set of size n_2 (n_1).

4 Algorithms

In this section, five algorithms for the permanent of an n-dimensional square matrix $A = (a_{ij}) \subset \mathbf{R}^{n \times n}$ and one algorithm for integer matrices are introduced.

Algorithm 1 (Differentiation of multivariate polynomial) *This is a trivial algorithm with a simple application of automatic differentiation.*

(1) Construct a program P_0 which computes the value of f in (2).
P_0 has n variables $x_1, \cdots, x_n$ as inputs.
(2) Specifying x_1 as the independent variable, derive a program P_1 from P_0 that computes $\partial f / \partial x_1(x_1, \cdots, x_n)$ with AD.

(3) Specifying x_2 as the independent variable, derive another program P_2 from P_1 that computes $\partial^2 f/\partial x_1 \partial x_2(x_1, \cdots, x_n)$ with AD.
(4) Similarly, specifying x_3, x_4, $\cdots$, x_n, derive programs P_3, P_4, $\cdots$, P_n, respectively, with AD. The last derived program P_n computes the value of the permanent as the derivatives in (3).

Remark 1. The derivations of the programs P_1, P_2, $\cdots$, P_n can be performed by repeated applications of a precompiler (e.g., Padre2 [316–318]). Of course, a similar program can be derived with a C++ class library (e.g., a program described in Fig. 1) defined for the multivariate higher order derivatives.

If the source program P_0 computes $f(x_1, \cdots, x_n)$ in (2) directly, the execution of P_0 requires $n^2 + n - 1$ multiplications and $n^2 - n$ additions. With AD, "one multiplication" and "one addition" in P_0 correspond to "two multiplications and one addition" and "one addition" in P_1, respectively. Thus, the number of multiplications in the execution of P_n is $\mathrm{O}(n^2 2^n)$.

Algorithm 2 (Differentiation of univariate polynomial) *With the Taylor series, the coefficients of the expansion of f in (6) can be computed directly. Here we assume that the values of α_i's are given as Lemma 1. Algorithms corresponding to other values (given by Corollary 1, Corollary 2, etc.) may be described similarly.*

(1) Construct a program P_0 that computes the value of f in (6).
(2) Replacing each variable v in P_0 with an appropriate data structure for Taylor series with degree $2^n - 1$ ($v = \sum_{k=0}^{2^n-1} c_k x^k$), derive a program P_1.
(3) Executing the derived program P_1, the value of the permanent is directly computed as the coefficient of x^{2^n-1}.

Remark 2. Since the degree of the Taylor series can be truncated at x^{2^n-1}, the size of the additional memory space is 2^n times of that for the original variables in the source program P_0. We should implement the multiplication of Taylor series by means of FFT so that the computational complexity of the execution of one multiplication is $\mathrm{O}(n2^n)$. The total computational complexity is $\mathrm{O}(n^2 2^n)$ since there are $(n-1)$ multiplications of 2^nth order Taylor series.

Algorithm 3 (Commutative quadratic nilpotent elements) *The n commutative quadratic nilpotent elements ξ_1, $\cdots$, ξ_n can be used as real arguments in the evaluation of $f(x_1, \cdots, x_n)$ (Sect. 3.2). The elements ξ_1, $\cdots$, ξ_n are implemented as instances of a class (named* `cqne` *in Sect. 3.2) in which overloaded multiplication and addition operators are defined.*

(1) Construct a program P_0 that computes the value of f in (2).
(2) Using the commutative quadratic nilpotent elements ξ_1, $\cdots$, ξ_n, derive another program P_1 that evaluates the value of $f(\xi_1, \cdots, \xi_n)$. That is, the data type `double` *that appears in P_0 should be replaced with the class* `cqne` *for commutative quadratic nilpotent elements in P_1 (in Sect. 3.2).*
(3) Executing P_1, the coefficient of the computed monomial as the result is equal to the value of the permanent.

Remark 3. The computational complexity for computing $s_i = a_{i1}\xi_1 + a_{i2}\xi_2 + \cdots + a_{in}\xi_n$ $(i = 1, \cdots, n)$ is $\mathrm{O}(n^2)$, and that for computing $[t_0 = 1$ and $t_k = t_{k-1} * s_k$ $(k = 1, \cdots, n)]$ is $\mathrm{O}(n(2^n - 2) \cdot n \log 2)$, where $n \log 2$ represents the coefficient corresponding to the insertion/search operations for a set of which size is at most

2^n. Therefore, the total computational complexity is $\mathrm{O}(n^2 2^n)$. Since the maximal number of the monomials in the intermediate expressions is $\binom{n}{\lfloor n/2 \rfloor}$, the complexity of set operations for one multiplication is $\mathrm{O}\left(n\binom{n}{\lfloor n/2 \rfloor} \log\left(n\binom{n}{\lfloor n/2 \rfloor}\right)\right)$ with a balanced tree structure. (This algorithm can be regarded as an implementation of the similar algorithm that computes only the monomials without quadratic factor in their terms [314].) The number of multiplications and additions are $n^2 + n - 1$ and $n^2 - n$, respectively. When we should take account of the bit operation, the complexity becomes $\mathrm{O}(n^3 2^n)$.

Algorithm 4 (Computation of residues) *For Lemma 1, $f(x)$ in (6) is a polynomial with degree $n2^{n-1}$, and the value of the permanent is given by the coefficient of x^{2^n-1}. Thus*

$$\frac{1}{2\pi i}\oint \frac{f(z)}{z^{2^n}}\,\mathrm{d}z = \frac{1}{2\pi}\int_0^{2\pi} f\left(e^{i\theta}\right)\cdot e^{-i(2^n-1)\theta}\,\mathrm{d}\theta = \frac{1}{2\pi}\int_0^{2\pi} g(\theta)\,\mathrm{d}\theta \tag{9}$$

gives the permanent, where $g(\theta) = f(e^{i\theta})e^{-i(2^n-1)\theta}$.

(1) Construct a program P_0 that, given a value of θ as input, computes the value of the function $g(\theta)$ defined by (9).
(2) Perform numerical integration of $g(\theta)$ on $[0, 2\pi]$ with the N-point trapezoidal rule, where N is given in the following Lemma 2.
(3) We have

$$\mathrm{per}(A) = \frac{1}{2\pi}\sum_{k=0}^{N-1} g\left(\frac{2\pi}{N}k\right)\cdot\frac{2\pi}{N} = \frac{1}{N}\sum_{k=0}^{N-1} g\left(\frac{2\pi}{N}k\right)\ . \tag{10}$$

Lemma 2 (Appropriate N). *Although the degree of the original polynomial is $n2^{n-1}$, the right-most equation in (10) with $N = 2^n$ gives the exact value of the permanent. That is, $N = 2^n$ does not cause the alias on the coefficient of x^{2^n-1} for DFT (Discrete Fourier Transformation).*

Proof. There are no monomials represented by $x^{2^n-1+k2^n}$ for non-zero k in $f(x)$ in (6) because of the similar argument in the Lemma 1 and its proof of the degree of $x^{2^n-1+k2^n}$. Thus $N = 2^n$ is large enough for the summation (10) to give the coefficient of x^{2^n-1}.

Remark 4. Algorithm 4 requires $\mathrm{O}(n^2 2^n)$ arithmetic operations, since $\mathrm{O}(n^2)$ is required for computing $g(\theta_j)$ for given θ_j, and there are $N = 2^n$ points $\theta_1, \cdots, \theta_{2^n}$ at which $g(\theta)$ is evaluated.

Algorithm 5 (Mixed algorithm) *This is a mixed algorithm using commutative quadratic nilpotent elements and FFT.*

After computation of three products each of which computes a product of $n/3$ factors with the commutative quadratic nilpotent elements, the convolutions of three products are computed with FFT of degree 2^n for avoiding aliases (Lemma 2). For simplicity, we assume that n is a multiple of 3.

(1) Divide (2) into three parts for $k = 1, 2, 3$: $s_k = \prod_{(k-1)n/3+1}^{kn/3} \sum_{j=1}^{n} a_{ij}x_j$. ($f(x_1, \cdots, x_n)$ is computed by $f(x_1, \cdots, x_n) = s_1 \cdot s_2 \cdot s_3$).

(2) Compute all the coefficients of monomials in s_1, s_2, s_3 with commutative quadratic nilpotent elements. The maximal number of the monomials is $\binom{n}{n/3}$.
(3) To construct a polynomial $s_1(x)$ by replacing ξ_i of the term of each monomial in s_1 with $x^{2^{i-1}}$ $(i = 1, \cdots, n)$, prepare an array consisting of 2^n components in which $s_1(x)$ is represented. Similarly, construct $s_2(x)$ and $s_3(x)$ in arrays from s_2 and s_3, respectively.
(4) With FFT of degree 2^n, derive the 2^n-dimensional vector $\hat{s}_1$ from $s_1(x)$ as well as $\hat{s}_2$ and $\hat{s}_3$ from $s_2(x)$ and $s_3(x)$, respectively.
(5) Construct $\hat{f}$ by the computing component-wise product of $\hat{s}_1$, $\hat{s}_2$ and $\hat{s}_3$. Then, derive all the coefficients of $f(x)$ from $\hat{f}$ by the inverse FFT.
(6) The coefficient of x^{2^n-1} of $f(x)$ is the value of the permanent.

Remark 5. This algorithm requires $\sum_{k=1}^{n/3} n \cdot \binom{n}{k} \log(n \cdot \binom{n}{k})$ arithmetic operations for computing each of s_1, s_2 and s_3, where $\log(n \cdot \binom{n}{k})$ represents the computational complexity of each insertion/search operation for a set with the size $n \cdot \binom{n}{k}$. Construction of $s_i(x)$ from s_i is $\mathrm{O}(2^n)$, and its FFT is $\mathrm{O}(n2^n)$ $(i = 1, 2, 3)$. Since the inequality $\sum_{k=1}^{n/3} n \cdot \binom{n}{k} \log(n \cdot \binom{n}{k}) \leq \frac{n}{3} n \binom{n}{n/3} \log\left(n \binom{n}{n/3}\right) \leq n2^n$ holds for large n, the total complexity is $\mathrm{O}(n2^n)$.

The following **Algorithm** 6 computes the value of the permanent of a non-negative integer square matrix $A \subset {\mathbf{Z}_+}^{n \times n}$.

Algorithm 6 (Higher precision computation) *When A is a non-negative integer matrix, e.g., (0,1)-matrix, the value of the permanent is also a non-negative integer. Thus, all the coefficients of $f(x)$ of (5) are also non-negative integers. Let M be an upper bound of the coefficients of $f(x)$. We have*

$$\mathrm{per}(A) \equiv \left\lfloor f(\frac{1}{M}) M^{\alpha_*} \right\rfloor \bmod M\ , \quad \textit{or} \quad \mathrm{per}(A) \equiv \lfloor f(M)/M^{\alpha_*} \rfloor \bmod M\ ,$$

where $\alpha_* = \sum_{i=1}^{n} \alpha_i$. Each equation requires only one evaluation of f, i.e., $f(1/M)$ or $f(M)$. The length of the precision of the arithmetic operations should be $2^n \log M$.

Remark 6. Algorithm 6 gives the value of the permanent with one evaluation of the polynomial $f(x)$. The number of arithmetic operations in the evaluation is $\mathrm{O}(n^2)$, but the required precision (bits) of each operation is $\mathrm{O}(2^n \log M)$ so that the total complexity is $\mathrm{O}(n^2 2^n \log M \log(2^n \log M) \log\log(2^n \log M))$.

5 Discussions and Comments

5.1 Higher Order Differentiation of Multiple Variables

AD for multiple variables to compute the higher derivatives of $f(x_1, \cdots, x_n)$ in (2) can be implemented with precompilers and/or with operator overloading features in C++. With symbolic manipulators, it is quite easy to derive the permanent. For taking account of the sparsity of the derivatives, use of the commutative quadratic nilpotent elements (Sect. 3.2) is better than the original differentiation of multivariate polynomial.

5.2 Use of Taylor Series

The transformation (Sect. 2.2) into univariate polynomials from the multivariate polynomial is simple, so that the use of Taylor series is also an easy way to compute the value of the permanent. Changing the data structure of the coefficient of Taylor series, for example, using a Boolean data type, can be used for the decision problem whether the value of the permanent is zero for (0,1)-matrices.

5.3 Transformation into Univariate Polynomial

We show several sets of α_i's for replacing x_i with x^{α_i}, to transform the multivariate polynomial (2) into univariate polynomials. There are other sets satisfying the uniqueness condition (**Condition** 2.1). For some small n, sets of α_i's that satisfy the condition are shown in Table 1.

Table 1. Examples of the set of α_i's satisfying the uniqueness condition

n	
3	0 1 3
	0 2 3
4	0 1 4 6
	0 2 5 6
5	0 1 5 11 13
	0 2 8 12 13

n	
6	0 2 10 21 22 26
	0 4 5 16 24 26
7	0 1 5 21 43 45 53
	0 8 10 32 48 52 53

Lemma 3. *For any n, if there exists a function $w(n)$ such that there is a set of α_i's ($\alpha_1 < \alpha_2 < \cdots < \alpha_n < w(n)$) satisfying the uniqueness condition 2.1, the value of the permanent can be computed with $O(n^3 w(n))$ arithmetic operations.*

Proof. The exact value of the permanent can be computed with numerical integration with the N-point trapezoidal rule. The N should be larger than the degree of the polynomial $f(x)$ in (5). $\mathrm{O}(n^2)$ operations for each evaluation of $f(x)$ and $N = n \cdot w(n)$ is big enough to integrate, thus the total computational complexity is $\mathrm{O}(n^3 w(n))$.

5.4 Computation of Residue

Since the degree of the polynomial is very large, the integrand should be integrated along with $z = e^{i\theta}$ in (9). The integration of a cyclic function on a whole cycle should be computed with the trapezoidal rule at equidistant points. Although $g(\theta)$ consists of components of very high frequencies, other numerical integration schemes may be considered for integration of $g(\theta)$ to reduce the number of the points at which g is evaluated. In this case, the resulting value is an approximation of the permanent with some truncation errors. More investigations may be needed in this direction.

6 Conclusion

We show algorithms for computing the exact value of the permanent with the technique of automatic differentiation and with commutative quadratic nilpotent elements. Mixing commutative quadratic nilpotent elements and FFT, we show another algorithm for the exact permanent whose computational complexity is $\mathrm{O}(n2^n)$, equal to that of Ryser's algorithm.

Computing Sparse Jacobian Matrices Optimally*

Shahadat Hossain[1] and Trond Steihaug[2]

[1] Department of Mathematics and Computer Science, University of Lethbridge, Canada
shahadat.hossain@uleth.ca
[2] Department of Informatics, University of Bergen, Norway
trond.steihaug@ii.uib.no).

Summary. A restoration procedure based on a priori knowledge of sparsity patterns of the compressed Jacobian matrix rows is proposed. We show that if the rows of the compressed Jacobian matrix contain certain sparsity patterns the unknown entries can essentially be restored with cost at most proportional to substitution while the number of matrix-vector products to be calculated still remains optimal. We also show that the conditioning of the reduced linear system can be improved by employing a combination of direct and indirect methods of computation. Numerical test results are presented to demonstrate the effectiveness of our proposal.

Key words: Sparse Jacobian matrices, indirect methods, partition, Pascal seeding.

1 Introduction

To determine sparse Jacobian matrices efficiently it is necessary to exploit information such as sparsity and other special structure such as certain regularity patterns of the sparsity structure. Given "seed" matrix $S \in R^{n\times p}$ and the Jacobian matrix $J \in R^{m\times n}$, the "compressed" Jacobian can be obtained as the product JS using, for example, the forward mode of automatic aifferentiation (AD). When the sparsity information is available a priori the nonzero entries of matrix J can be restored (i.e. recovered) by solving for them in the linear system of equations

$$JS = B \ ,$$

where B is the compressed Jacobian matrix obtained via AD forward mode. Unless stated otherwise the product JS is assumed to be computed as p matrix-vector products using AD forward mode from which the unknown elements of J are solved. The Curtis, Powell, and Reid (CPR) [141] compressions allow the nonzero entries

* This research is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Norwegian Research Council (NFR).

to be "read off" the compressed Jacobian directly. However, it has been shown that finding minimal CPR determination [128] and optimal direct determination [272] of Jacobian matrices are NP-hard. On the other hand, with certain classes of seed matrices the nonzero entries of J can be obtained indirectly via substitution or elimination [225]. While indirect methods, in general, are more efficient than direct methods in that they typically require fewer matrix-vector products, numerical precision of the computed quantities may suffer due to the accumulation of round-off errors. In the elimination proposal of Newsam and Ramsdell [412] two classes of seed matrices have been considered: the Vandermonde and the Chebyshev-Vandermonde. Although the Vandermonde seeding allows for efficient solution of the reduced linear system for the nonzero entries, the numerical conditioning is a major concern except for when the number of nonzero entries in each row is very small. Geitner, Utke, and Griewank's proposal [201] uses graph coloring to improve the numerical conditioning of the computed entries using the Newsam and Ramsdell approach. Hossain and Steihaug [271] suggest elimination schemes that employ new classes of seed matrices which have been further analyzed and tested by Griewank and Verma [233]. We follow the terminology in [233] and use the term Pascal seeding for the one suggested in [271]. With the new classes of seed matrices the numerical accuracy of the computed quantities has been found to be very favorable compared with the Vandermonde systems. Moreover, the resulting linear systems can be solved for the unknowns quite efficiently by using the special structure of the Pascal seed matrix. In the present work, we propose extensions to the indirect methods that improve the numerical conditioning further while economizing computational effort in the restoration phase.

The notational conventions used in this paper are as follows. If an uppercase letter is used to denote a matrix (e.g., A), the (i, j) entry is denoted by $A(i, j)$ or by the corresponding lowercase letter a_{ij}. We use colon notation [222] to specify a submatrix of a matrix. For $A \in R^{m\times n}$, $A(i,:)$ and $A(:, j)$ denote the ith row and the jth column, respectively. For a vector of column indices v, $A(:, v)$ denotes the submatrix comprised of columns whose indices are contained in v. For a vector of row indices u, $A(u,:)$ denotes the submatrix comprised of rows whose indices are contained in u. A vector is specified using only one dimension. For example, the ith element of $v \in R^n$ is written $v(i)$. The transpose operator is denoted by $(\cdot)^T$. A blank or "0" represents a zero entry, and any other symbol in a matrix denotes a nonzero entry.

The remainder of this paper is organized as follows. In Sect. 2 we describe the sparse Jacobian matrix computation problem via matrix compression. We provide a brief overview of matrix compression techniques that have appeared in the literature. Section 3 presents the Schur complement approach for the restoration of nonzero entries from the compressed Jacobian matrix. For each Jacobian row an extended linear system is obtained, and the nonzero entries are computed from the Schur complement. The special structure of the extended system economizes the computational cost in recovering the nonzero entries. In Sect. 4 we describe how to combine different compression and restoration techniques adaptively to recover the nonzero entries efficiently and accurately. Section 5 contains the discussion of exploiting "local" sparsity patterns of Jacobian rows. For certain zero-nonzero patterns we show that the linear system to be solved yields substitution while using minimal number of matrix-vector products. Section 6 presents results from numerical experiments. The paper concludes in Sect. 7 with some final remarks.

2 Optimal Matrix Compression and Restoration

Both AD and Finite Differencing (FD) allow estimates of the nonzero entries of a Jacobian matrix to be obtained as matrix-vector products. For a sparse matrix with known sparsity pattern or if the sparsity can be determined easily [230] substantial savings in computational cost (i.e. the number of matrix-vector products) can be achieved.

A group of columns in which no two columns have nonzero elements in the same row position is known as *structurally orthogonal*. If columns j and k are structurally orthogonal, then for each row index i at most one of $A(i,j)$ and $A(i,k)$ can be nonzero. In general $\sum_j A(i,j) = A(i,k)$ for some k (k will depend on i), where the sum is taken over the column indices of a group of structurally orthogonal columns. An estimate of the nonzero elements in the group can be obtained in

$$\left.\frac{\partial f(x+ts)}{\partial t}\right|_{t=0} = f'(x)s \approx As = \frac{1}{\epsilon}[f(x+\epsilon s) - f(x)] \equiv b \tag{1}$$

with a forward difference (one extra function evaluation), where b is the finite difference approximation, and $s = \sum_j e_j$ where e_j denotes the jth coordinate vector. With forward AD, the unknown elements in the group are obtained as the product $b = f'(x)s$, accurate up to the round-off error resulting from the finite machine precision. The sparse Jacobian matrix determination problem can be stated as follows.

> Obtain vectors $s_1, \cdots, s_p$ such that the matrix-vector products
>
> $$b_i \equiv As_i,\ i = 1, \cdots, p \text{ or } B \equiv AS$$
>
> determine the $m \times n$ matrix A uniquely. Matrix S is popularly called the *seed matrix*.

Denote by ρ_i the number of nonzero elements in row i of A and let $v \in R^{\rho_i}$ contain the column indices of (unknown) nonzero elements in row i of A. Let $S \in R^{n \times p}$ be any (seed) matrix. Compute

$$B = AS\ .$$

Let

$$A(i,v) = \big(\ \alpha(1)\ \cdots\ \alpha(\rho_i)\ \big) = \alpha^T, \alpha \in R^{\rho_i},$$
$$B(i,:) = \big(\ \beta(1)\ \cdots\ \beta(p)\ \big) = \beta^T, \beta \in R^p\ ,\ \text{and}$$
$$\hat{S_i}^T = S(v,:)\ .$$

Then the unknown elements satisfy the overdetermined ($\rho_i \leq p$) linear system

$$\hat{S}_i \alpha = \beta\ . \tag{2}$$

Without loss of generality, assume that $\rho_i = p$. Then if $\hat{S}$

- is a permutation matrix, we have *direct determination*,
- can be permuted to a triangular matrix, we have *determination by substitution*, and
- is a general nonsingular matrix, we have *determination by elimination*.

For sparse Jacobian matrix determination we are interested in the seed matrices for which the reduced linear system is numerically well-conditioned and "easy" to solve. The CPR method uses a greedy technique to partition columns into structurally orthogonal groups. The resulting seed matrix yields direct determination of nonzero entries implying that the nonzero entries are simply identified in the compressed matrix B. In the following we will assume that the matrix A is a $m \times p$ CPR-compressed matrix and the number of columns in the seed matrix is q. It is important to notice that in a CPR-compression the number of nonzero elements in any row is preserved. Let ρ_{max} denote the maximum number of nonzero elements in any row of A. Then to determine the nonzero entries uniquely at least ρ_{max} products are needed in any method based on matrix-vector product calculation. The column merging technique proposed in [270] defines seed matrices for which the reduced linear systems are triangular while requiring fewer matrix-vector products than CPR-based direct methods.

An important feature of elimination methods such as [271,412] is that the number of matrix-vector products needed to completely determine a sparse Jacobian matrix using AD forward mode is ρ_{max}. Recall that if

$$\mathcal{P}(\lambda) = \sum_{j=1}^{q} a_j \lambda^{j-1}$$

is a polynomial of degree $q-1$, where the coefficients a_j are to be determined by interpolation, then $\mathcal{P}(\lambda_j) = \mathcal{P}_j$ for $j = 1, \ldots, q$ defines a system of q linear equations from which the coefficient vector $a \in R^q$ is uniquely determined if the λ_i are distinct. Thus the set

$$\left\{ \lambda^{j-1} : \lambda \in R \text{ for } j = 1, \ldots, q \right\}$$

forms a basis for the function space spanned by polynomials. In matrix notation, the linear system is written

$$V^T a = \mathcal{P} \ ,$$

where V is a Vandermonde matrix, which can be solved in $O(q^2)$ floating point operations [222]. The difficulty here is that the numerical conditioning of the system deteriorates exponentially with the size of V. This difficulty with conditioning is ameliorated by choosing Chebyshev polynomials $\mathcal{T}_i$ to form an orthogonal basis for the polynomial function space. Such systems, also known as Chebyshev-Vandermonde systems, provide an alternative to Vandermonde systems with a computational cost of $O(q^2)$ floating point operations [458]. Vandermonde and Chebyshev-Vandermonde seed matrices have been considered in [201, 412]. To define the successive column merging seed matrix [271] also known as the Pascal seed matrix, let 0_j denote the zero vector in R^j. Then row i in S^T is (column i in the seed matrix $S \in R^{p \times q}$)

$$[0_{i-1} \ u \ 0_{p-d-i}] \ ,$$

where component j in $u \in R^{d+1}$ with $d \leq p - \rho_{max}$ is the binomial coefficient $\binom{d}{j-1}$. Compared with the Vandermonde-type seed matrices, the Pascal matrix is sparse, structured, and contains only integer entries [271]. Furthermore, the condition number of the reduced system grows only modestly with p, e.g., as p^d. In Sect. 3 we provide a closed form expression of $\| \cdot \|_1$ condition number estimate for Pascal seeding. If we accept a condition number, for example, of 10^8 and want to

use the fact that every row of A has 4 zeros, we can allow p to be as big as $p = 100$. Very few of the standard test problems have that many columns in the compressed Jacobian matrix. Also, since d can be considered as a (user controlled) parameter, the nonzero entries can be determined with higher precision with an increase in the computational cost due to the calculation of additional matrix-vector products.

Preprocessing the Jacobian matrix with the CPR seeding reduces the problem dimension from $m \times n$ to $m \times p$. The rate of growth of the condition number for the Pascal seeding of $O(p^d)$ indicates that numerical conditioning can be improved by an appropriate preprocessing step. This observation can be generalized as combining methods in which parts of the Jacobian matrix are compressed and restored adaptively. In an implementation, the procedure entails restoration of unknown entries from the product of the Jacobian matrix with a sequence of seed matrices. We elaborate this idea in Sect. 4.

The use of sparsity in the preprocessed (or pre-compressed) Jacobian matrix rows can be highly effective in improving the numerical conditioning of the reduced linear system while maintaining the optimality of the number of matrix-vector products. The substitution scheme that exploits the consecutive zeros property in the Jacobian rows [270] is generalized as a "recurring" sparsity pattern in Sect. 5.

3 Schur Complement Approach

Let v and z be vectors of column indices of nonzero and zero elements, respectively, of some row of A and let ρ denote the number of nonzero elements in that row. Suppose $S \in R^{p\times q}$ is a seed matrix and assume that the compressed Jacobian $B = AS$ has been computed. Define $Z \in R^{d\times p}, d = (p - q),\ q \geq \rho$

$$Z(i,:) = e^T_{z(i)},\ i = 1, \ldots, d\ .$$

Consider the matrix

$$W = \begin{pmatrix} S^T \\ Z \end{pmatrix}\ ,$$

and let

$$A(i,v) = \begin{pmatrix} \alpha(1) \cdots \alpha(\rho) \end{pmatrix} = \alpha^T, \alpha \in R^\rho,$$
$$B(i,:) = \begin{pmatrix} \beta(1) \cdots \beta(q) \end{pmatrix} = \beta^T, \beta \in R^q\ .$$

Then,

$$Wx = b \tag{3}$$

with $x = (x_1\ x_2)\,, \alpha = x(v)$ and $b = (\beta\ 0)$. Let the matrix W be partitioned into blocks so that (3) can be written as

$$\begin{pmatrix} S_1 & S_2 \\ Z_1 & Z_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \beta \\ 0 \end{pmatrix}\ , \tag{4}$$

where $S_1 = S(1:q, 1:q)^T$, and the vectors x and b are partitioned accordingly. Equation (4) can be solved for x as below.

1. Compute: $-Z_1 S_1^{-1} S_2 + Z_2$
2. Solve for x_2: $(-Z_1 S_1^{-1} S_2 + Z_2)x_2 = -Z_1 S_1^{-1}\beta$
3. Solve for x_1: $S_1 x_1 = \beta - S_2 x_2$

4. Obtain: $\alpha = x(v)$

The seed matrix S can be chosen to be column merging [270], Pascal, Vandermonde, or other suitable matrix, every square submatrix of which has full rank. We call the coefficient $-Z_1 S_1^{-1} S_2 + Z_2$ in Step 2 in the above algorithm the *Schur complement matrix* or simply the *Schur matrix*. Computing $S_1^{-1} S_2$ entails solving d linear systems with the same coefficient matrix S_1 and the same right hand side given by S_2 for each row of A. Therefore, S_1 needs to be factored only once, and the factors are used repeatedly. Further, Z_1 and Z_2 are $0-1$ matrices so that computing their product with other matrices is simple.

In the following, we assume that the matrix S is the Pascal seed matrix. It can be verified that the block S_1 is upper triangular, and the nonzero entries are the binomial coefficients:

$$S_1(i,j) = \begin{cases} \begin{pmatrix} d \\ j-1 \end{pmatrix} & \text{if } i+d \geq j \geq i \\ 0 & \text{otherwise} \end{cases}, 1 \leq i,j \leq q \ . \tag{5}$$

Then ${S_1}^{-1}$ is upper triangular and is given by

$${S_1}^{-1}(i,j) = \begin{cases} (-1)^{j-i} \begin{pmatrix} d+j-i-1 \\ d-1 \end{pmatrix} & \text{if } j \geq i \\ 0 & \text{otherwise} \end{cases}, 1 \leq i,j \leq q \ . \tag{6}$$

$$W = \begin{array}{c} \begin{matrix} \quad S_1 & & & & & S_2 & & \end{matrix} \\ \left(\begin{array}{ccccc|ccc} 1 & 3 & 3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 & 3 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 3 & 3 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 3 & 3 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right) \\ \begin{matrix} \quad Z_1 & & & & & Z_2 & & \end{matrix} \end{array}, \quad \begin{matrix} \text{Schur Matrix} \\ (-Z_1 S_1^{-1} S_2 + Z_2) \end{matrix} = \begin{pmatrix} -21 & -35 & -15 \\ 6 & 8 & 3 \\ -3 & -3 & -1 \end{pmatrix}$$

Fig. 1. Schur matrix with the Pascal seeding when $p = 8, q = 5$, and $z = (1 \ \ 4 \ \ 5)$.

Figure 1 shows the Schur matrix W with $p = 8$ and $q = 5$. Vector z indicates that columns 1, 4, and 5 contain zero in some row of the Jacobian matrix. The remaining $q = 5$ elements are nonzero and are to be determined. We note that $\rho \leq q \leq p$ is an important user-supplied parameter. With $q = \rho$ the number of matrix-vector products needed is minimal but with a corresponding increase in the computational cost for restoring the nonzero elements, while $q = p$ results in direct determination of the nonzero entries.

Using (6), the inverse of S_1 is computed as

$$S_1^{-1} = \begin{pmatrix} 1 & -3 & 6 & -10 & 15 \\ 0 & 1 & -3 & 6 & -10 \\ 0 & 0 & 1 & -3 & 6 \\ 0 & 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} .$$

We have no rounding errors in computing $-Z_1 S_1^{-1} S_2 + Z_2$ since the matrix W contains only integer entries. An estimate of the numerical conditioning of S_1 in L_1 norm can be given as

$$\kappa_1 = \|S_1\|_1 \|{S_1}^{-1}\|_1 = \sum_{i=0}^{d} \binom{d}{i} \sum_{i=0}^{q-1} \binom{d+i-1}{d-1} = 2^d \binom{q+d-1}{d} = O(q^d) .$$

Conditioning of the step 3 is the worst possible and is $O(p^{2d})$. However, this modest increase in conditioning does not seem to cause significant numerical difficulties in our computational experiments.

4 Combined Determination

Pre-compressing the Jacobian matrix $J \in R^{m \times n}$ with the CPR seeding improves the numerical conditioning of the reduced linear systems. The pre-compressed matrix $A = JS^{(cpr)}$ is further compressed by, e.g., the Pascal seeding to give $B = AS^{(Pascal)}$ from which the nonzero entries are solved. In general we can describe the compression process as a sequence of sub-compressions $S^{(1)}, S^{(2)}, \ldots, S^{(l)}$ such that

$$S = S^{(1)} S^{(2)} \cdots S^{(l)}, \quad JS = B ,$$

$S^{(i)}$ is constructed using sparsity information in $A^{(i)}$

$$A^{(i+1)} = A^{(i)} S^{(i)}, \quad A^{(1)} = J ,$$

and $A^{(i)}$ is restored from

$$A^{(i)} S^{(i)} = A^{(i+1)}, \quad A^{(i+1)} = B .$$

In this way appropriate seeding can be chosen based on the sparsity and other structural information such as regularity of sparsity pattern at different stages of the compression process.

Often the available sparsity can be better used in a piecemeal fashion, e.g., by blocks of columns based on row sparsity pattern. The CPR, being a greedy algorithm, tries to put as many columns as possible into the current group of structurally orthogonal columns. Therefore, it is anticipated that the earlier column groups will contain most of the nonzero entries. The distribution of zero entries in the Jacobian row affects the conditioning of the system in step 2 of the Schur complement approach. If the zero entries in a row occur in the last d locations then the block Z_1 are

zero, and the block Z_2 becomes a permuted identity matrix. Then x_2 is zero and x_1 is determined in step 3 without having to perform steps 1 and 2. If most of the zeros are in the second half, say of length p', the first half (i.e., the columns in the first half) can be determined directly, and the conditioning now grows as $O((p')^d)$ rather than $O(p^d)$ using Pascal seeding, for example. To illustrate, let $A = [A_1\ A_2] \in R^{m\times p}$ be a Jacobian matrix such that A_1 is best restored by method 1 with the seed matrix $\tilde{S_1} \in R^{p'\times q'}$ and that A_2 is best restored by method 2 with the seed matrix $\tilde{S_2} \in R^{p''\times q''}$ where $p = p' + p''$ and $q = q' + q''$. Define

$$S_1 = \begin{pmatrix} \tilde{S_1} & 0 \\ 0 & I \end{pmatrix} , \text{ where } I \text{ denotes the identity matrix of order } p'' , \text{ and}$$

$$S_2 = \begin{pmatrix} I & 0 \\ 0 & \tilde{S_2} \end{pmatrix} , \text{ where } I \text{ denotes the identity matrix of order } q' .$$

Then $S = S_1S_2$ constitutes the seed matrix for the methods 1 and 2 combined.

5 Using Recurring Sparsity Structure in Rows

It has been shown in [270] that if each row of $A \in R^{m\times p}$ has a "pattern" of at least d consecutive zero entries with $d \le p - \rho_{max}$, then a seed matrix $S \in R^{p\times(p-d)}$ can be defined for which column i has the form

$$[0_{i-1}\ e\ 0_{p-d-i}] \ , \tag{7}$$

where $e \in R^{d+1}$ is a vector of all 1's. Then $\hat{S_i}, i = 1, 2, \ldots, m$ corresponding to row i of A is nonsingular and substitutable. More specifically, $\hat{S_i}$ consists of rows of S with indices corresponding to column indices of the nonzero entries of row i of A. Since row i of A contains a sequence of d consecutive zeros, the rows of S picked up for $\hat{S_i}$ must be d rows apart leading to a split in $\hat{S_i}$ into a lower triangular and an upper triangular part.

If each row of A contains a pattern of $p - \rho_{max}$ consecutive zeros, the resulting substitution scheme is optimal in the number of matrix-vector products needed to determine A. In general, the zeros in each row may not be consecutive. Finding a column permutation so that zeros in each row appear consecutively is also NP-hard [271]. Extending the idea of consecutive zeros pattern we can allow few nonzero entries to be interspersed with zeros. To illustrate, consider matrix A with $p = 8$, $\rho_{max} = 6$, and $d = 2$. Suppose for some row of A the two zeros are not consecutive, but in each row there exists a pattern of three elements in which two of them are identically zero (not necessarily consecutive). One such row is shown below.

$$[\times\ \times\ \underbrace{0\ \times\ 0}_{\uparrow}\ \times\ \times\ \times\ ] .$$
$$\text{pattern}$$

Then A can be determined by substitution with $q = 6$ matrix-vector products using the seed matrix shown in (7) with $d = 2$. Thus, if for each row of A there exists a sequence of $d + 1$ elements of which d elements are identically zero, the column-merging seed matrix of (7) can be used to recover the nonzero entries with $p - d$

matrix-vector products. In other words, d zero elements in a row can have one nonzero element interspersed.

Let d' be the number of nonzero entries in the pattern. The corresponding seed matrices will be chosen so that the splitting of the reduced seed matrices could involve solving a $d' \times d'$ linear system and the remaining unknown entries in the row can be restored via substitution.

Possible recurring sparsity patterns of length five with two nonzero ($d' = 2$) elements that can be restored with $e = [1101]$ result in the eight patterns

$$\begin{array}{l} \times\ \times\ 0\ 0\ 0,\ \times\ 0\ \times\ 0\ 0,\ \times\ 0\ 0\ 0\ \times,\ 0\ \times\ \times\ 0\ 0, \\ 0\ \times\ 0\ \times\ 0,\ 0\ \times\ 0\ 0\ \times,\ 0\ 0\ \times\ \times\ 0,\ 0\ 0\ 0\ \times\ \times\ . \end{array}$$

A pattern of length $d+2$ with two nonzero elements separated by one zero element can be restored with $e = [1 \ldots 101 \ldots 101 \ldots 1]$ (for $d \geq 4$ and $d+2$ divisible by 3). Figure 2 depicts the reduced seed matrix for a pattern of length five with two nonzero entries interspersed with three zeros. The 2×2 triangular system in the middle must be solved to start substituting for the remaining unknown entries in the Jacobian row.

Use of a priori known "regular patterns" in sparse Jacobian and Hessian matrices has been considered in [218]. Computational molecules (or stencil) for Laplacian approximated by finite differences in the solution of certain type of partial differential equation problems give rise to sparse Jacobian and Hessian matrices. The techniques presented in [218] can optimally determine matrices with certain regular sparsity patterns such as banded structure. In our proposal the zero-nonzero patterns of interest in the rows of the pre-compressed Jacobian matrix need not have any specific structure.

Returning to the recurring sparsity pattern of length five with two nonzero elements, the patterns

$$\times\ 0\ 0\ \times\ 0 \text{ and } 0\ 0\ \times\ 0\ \times$$

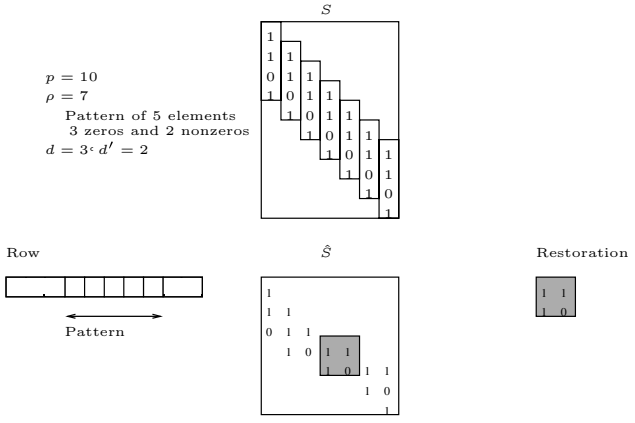


Fig. 2. Restoration of pattern $0\ \times\ 0\ \times\ 0$

cannot be restored for this particular seed matrix. On the other hand, with the Pascal seeding, all patterns are restorable.

6 Numerical Experiments

In this section we provide numerical test results of applying the restoration techniques proposed in the paper. The test programs are implemented in MATLAB and run on a SunBlade and an Intel PC running Solaris and Windows XP, respectively. We report experiments on conditioning of the linear system for the restoration of nonzero elements where the test problems are drawn from the Harwell-Boeing test matrix collection [154]. The six test problems are pattern matrices where any optimal structurally orthogonal column partition contains more than ρ_{max} groups. Table 1 reports the important matrix statistics as well as the cardinality of the optimal structurally orthogonal column partition obtained via exact graph coloring [272]. The number of column groups in the optimal partition is more than ρ_{max} for all problems. Figure 3 depicts the L_2 condition number (in sorted order) of the Pascal (reduced) seed matrices and the Pascal Schur matrices for the Harwell-Boeing test problems. In both the approaches the growth of the condition number is modest; only a small percentage of the rows experience poor conditioning. In general, the conditioning of the Schur matrices are poorer than that of the Pascal reduced seed matrices.

Table 1. Structurally orthogonal column partition of Harwell-Boeing test problems.

Name	Number of columns (n)	Number of rows (m)	ρ_{max}	Optimal partition
ash219	85	219	2	4
abb313	176	313	6	10
ash331	104	331	2	6
will199	199	199	6	7
ash608	188	608	2	6
ash958	292	958	2	6

7 Concluding Remarks

We have presented effective ways to exploit sparsity to determine sparse Jacobian matrices. The Schur complement approach is highly efficient in restoring the nonzero entries. On practical problems the numerical conditioning does not cause a significant problem. The combined determination technique proposed here can be applied with the pattern restoration to reduce the numerical conditioning further. More importantly, the linear system we have to solve for the nonzero entries in pattern restoration is at most $\rho_{max} \times \rho_{max}$, which is much smaller than p – the number of columns in the compressed Jacobian. Most of the computations in the Schur complement approach can be performed in integer arithmetic, minimizing the potential round-off error on a finite precision computer.

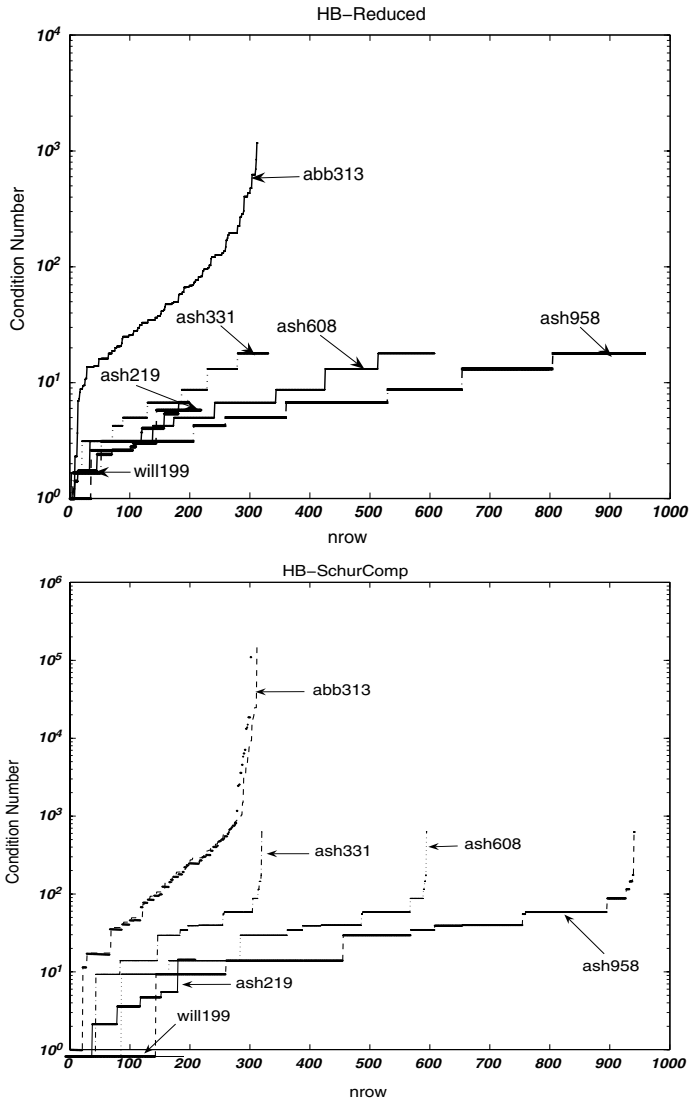


Fig. 3. Condition number of the Harwell Boeing matrices. (a) Pascal (reduced) seed matrix. (b) Schur matrix.

The pattern restoration substitution technique reduces the condition number based on the identification of particular sparsity pattern of a row to be determined. A related question is concerned with the identification of a pattern automatically and prescribing a seed matrix for the restoration of the pattern. We note that the columns of the CPR-compressed Jacobian matrix can be permuted to shift the zero entries toward the right end of the matrix.

Application of AD-based Quasi-Newton Methods to Stiff ODEs

Sebastian Schlenkrich[1], Andrea Walther[1], and Andreas Griewank[2]

[1] Technische Universität Dresden, Institute for Scientific Computing, Dresden, Germany
{schlenk,awalther}@math.tu-dresden.de
[2] Humboldt Universität Berlin, Department of Mathematics, Berlin, Germany
griewank@mathematik.hu-berlin.de

Summary. Systems of stiff ordinary differential equations (ODEs) can be integrated properly only by implicit methods. For that purpose, one usually has to solve a system of nonlinear equations at each time step. This system of equations may be solved by variants of Newton's method. The main computing effort lies in forming and factoring the Jacobian or a suitable approximation to it. We examine a new approach of constructing an appropriate quasi-Newton approximation for solving stiff ODEs. The method makes explicit use of tangent and adjoint information that can be obtained using the forward and the reverse modes of algorithmic differentiation (AD). We elaborate the conditions for invariance with respect to linear transformations of the state space and thus similarity transformations of the Jacobian. We present one new updating variant that yields such an invariant method. Numerical results for Runge-Kutta methods and linear multi-step methods are discussed.

Key words: Quasi-Newton, stiff ODE, adjoint-based update, scaling invariance, ADOL-C

1 Introduction

For many time-dependent simulations, the underlying system can be modelled as the solution of an initial value problem (IVP)

$$\dot{x}(t) = f(x(t)) \,, \quad t \in (0,T) \,, \quad x(0) = \eta \in \mathbb{R}^n \,, \tag{1}$$

on a time interval $[0,T]$, where $x(t)$ denotes the state variable. To compute a numerical approximation of the solution x, we perform a discretization $\{t_0, \ldots, t_N\}$ of the time interval $[0,T]$ using the step size $h_k = t_k - t_{k-1}$. Applying a numerical integration method yields the discrete solution vectors

$$x_k = x_{k-1} + h_k \Phi(x_k, x_{k-1}, \ldots, x_{k-l}, h_k) \quad k = 1, \ldots, N, \qquad x_0 = \eta$$

at the times t_k. Here, $\Phi = \Phi(x_k, \ldots)$ represents the step function of the integration method. If Φ does not depend on the so far unknown x_k, the integration method is called explicit and is well suited for a wide range of ordinary differential equations (ODEs). However, as soon as the underlying problem is described by stiff ODEs, for example due to very different time scales (see [249]), implicit methods must be used to allow a reasonable time step. Then, the step function Φ also depends on the unknown value x_k. Therefore, the new state x_k must be obtained by solving the n-dimensional system

$$F_k(x_k) := x_k - x_{k-1} - h_k \Phi(x_k, x_{k-1}, \ldots, x_{k-l}, h_k) = 0 \in \mathbb{R}^n \ . \tag{2}$$

Since f is usually nonlinear, (2) is often also nonlinear, although there are classes of linearly implicit ODE solvers. To compute the next state x_k, one may apply an iterative Krylov method. As an alternative, one may solve a system of mostly nonlinear equations in each integration step using an iteration of the form

$$x^{(i+1)} = x^{(i)} - A_i^{-1} F_k(x^{(i)}) \ , \tag{3}$$

where the sequence $\{x^{(i)}\}_{i \in \mathbb{N}}$ should converge to the solution $x^* = x_k$ of (2) for given values of $x_{k-1}, \ldots, x_{k-l}$. For that purpose, Newton's method can be applied by setting $A_i = F'(x_k)$ if the complete Jacobian is available at a reasonable cost. Since factoring the Jacobian at each time step is usually quite expensive, we present here a new adjoint-based quasi-Newton method that provides a factorized approximation of the Jacobian. First studies in this direction were made by Brown et al. [74] for Broyden's method. However, the use of quasi-Newton methods is not widespread, for two reasons. First, the quasi-Newton methods proposed up to now were not scaling invariant. Hence, a simple scaling of the variables may strongly effect the convergence behaviour. Second, so far it is not possible to perform adaptive time stepping with a cost that is quadratic in the dimension of the problem, because the factorization that is updated cannot be adapted to the new time step cheaply. Hence, one has to perform a new QR-factorization as soon as the time step changes. Therefore, the most well-known packages for the integration of stiff ODEs, DASPK [73] and CVODE [127], only provide several variants of Newton's method as direct methods or Krylov methods as indirect variants but no low-rank updating approach.

The quasi-Newton updates that we propose use tangent and adjoint information obtained using algorithmic differentiation (AD). One of them has the property of scaling invariance to overcome the first problem of quasi-Newton updates in the context of stiff ODEs. Future work will be dedicated to a new factorization procedure that allows also a comparatively cheap change of the time step size. These two ingredients would form a powerful combination that should allow a more extensive use of quasi-Newton methods for the integration of stiff ODEs.

The paper has the following structure. The new quasi-Newton updates are presented in Sect. 2, and we elaborate the conditions for invariance with respect to similarity transformations of the Jacobian. The resulting update formulas are implemented using C/C++ as the programming language and the AD-tool ADOL-C for providing the required derivatives. Implementation details are described in Sect. 3. The numerical results obtained for two Runge-Kutta methods and two BDF methods are discussed in Sect. 4. Finally, a summary and an outlook of future work are given in Sect. 5.

2 Quasi-Newton Approximations

Applying Newton's method to the system (2), the complete Jacobian of F is required for each time step. Additionally, one has to factorize the Jacobian to solve the linear system. For large dimensions n, the derivative information $F'(x_i)$ can be computed within machine accuracy using AD, but the computational complexity may grow linearly in n, for example if the Jacobian is dense. Together with the cubic effort required for the factorization, this cost is often not acceptable. Alternatively we can use information on F from previous iterations and update an approximation of the Jacobian. For this purpose, one may apply rank-1 updates. Then the approximation A_{i+1} of the Jacobian at $x^{(i+1)}$ is given by

$$A_{i+1} = A_i + uv^T \ ,$$

with two vectors u and $v \in \mathbb{R}^n$ to be determined. For almost all previously proposed quasi-Newton methods, the two vectors are chosen such that the direct tangent condition

$$A_{i+1}s_i = F'_k(x^{(i+1)})s_i \tag{4}$$

or the secant condition

$$A_{i+1}s_i = F_k(x^{(i+1)}) - F_k(x^{(i)}) \equiv y_i \tag{5}$$

is fulfilled with $s_i = x^{(i+1)} - x^{(i)}$. Since the forward mode of AD provides the information $F'_k(x^{(i+1)})s_i$ at a very moderate cost, we will use the exact direct tangent condition (4) throughout this paper. The secant condition (5) is used for example in Broyden's method given by

$$A_{i+1} = A_i + \frac{(y_i - A_i s_i)s_i^T}{s_i^T s_i} \ .$$

Applying the reverse mode of AD, one can evaluate the product $z_i^T F'_k(x^{(i+1)})$ for a vector z_i also at a moderate cost. In the context of solving nonlinear equations using quasi-Newton methods, this property yields the adjoint tangent condition

$$z_i^T A_{i+1} = z_i^T F'_k(x^{(i+1)}) \ . \tag{6}$$

Provided $z_i^T A_i s_i \neq z_i^T F'_k(x^{(i+1)})s_i$ the two tangent conditions (4) and (5) are consistent, and there is exactly one rank-1 update of A_i satisfying them, namely

$$A_{i+1} = A_i + \frac{(F'_k(x^{(i+1)})s_i - A_i s_i)(z_i^T F'_k(x^{(i+1)}) - z_i^T A_i)}{(z_i^T F'_k(x^{(i+1)}) - z_i^T A_i)s_i} \ . \tag{7}$$

This formula is referred to as *Two-sided Rank-1 (TR1)* update and has been exploited in the context of nonlinear optimization [235]. Whereas we choose naturally $s_i = x^{(i+1)} - x^{(i)}$ the question remains how we select the adjoint directions z_i. However, for integrating stiff ODEs, there is no obvious choice for the weight vector z_i appearing in the adjoint tangent condition. This situation differs significantly from the nonlinear optimization context, where the adjoint weight vectors can be defined as corrections of the Lagrange multipliers in a natural way. Since this is not possible for the pure integration of stiff ODEs, we will discuss two alternatives for choosing

the adjoint weight vector z_i. Nevertheless, this setting will be completely changed if the ODEs form equality constraints in an optimal control setting. Then, once more, the adjoint weight vector can be defined on the base of Lagrange multipliers.

Least-squares approach

To motivate the definition of z_i, we employ the linear model

$$M(x) := F_k(x^{(i+1)}) + A_{i+1}(x - x_{i+1})$$

of F in x^{i+1}. Then, the first approach refers to a minimization problem corresponding to (2). With $J(x) := \|F_k(x)\|_2^2$ it is given by

$$J(x^*) = \min_x J(x) \qquad \Longleftrightarrow \qquad F_k(x^*) = 0 \; .$$

We suppose that the gradient of $J(x^{(i+1)})$ provides a decent direction. Then the gradient of the minimization problem $\tilde{J}(x) := \|M(x)\|_2^2 \to \min$ of the model M should be the same in $x^{(i+1)}$. This yields the condition

$$\nabla J(x^{(i+1)}) = \nabla \tilde{J}(x^{(i+1)}) \qquad \Longleftrightarrow \qquad z_i = F_k(x^{(i+1)}),$$

and we call the resulting formula for A_{i+1} *Least-squares update*.

Scaling invariance

A favored property of the iterative method (3) is independence with respect to linear transformations in the state space of the ODE. Suppose additionally to (1) one has a transformed IVP

$$\dot{\tilde{x}}(t) = \tilde{f}(\tilde{x}(t)) \; , \quad t \in (0,T) \; , \qquad \tilde{x}(0) = \tilde{\eta} \in \mathbb{R}^n \; . \tag{8}$$

This is related to the original problem by a linear transformation of the state space with a regular $T \in \mathbb{R}^{n \times n}$ such that

$$\tilde{f}(\tilde{x}(t)) = T f(T^{-1}\tilde{x}(t)) \; , \quad t \in (0,T) \; , \quad \text{and} \quad \tilde{\eta} = T\eta \; . \tag{9}$$

Then for the solutions of the original problem (1), and the transformed problem (8) $\tilde{x}(t) = Tx(t)$ is valid for $t \in (0,T)$.

For the implicit Euler's method, transformation (9) yields with $\tilde{x} = Tx$

$$\begin{aligned} \tilde{F}_k(\tilde{x}) &= \tilde{x} - \tilde{x}_{k-1} - h\tilde{f}(\tilde{x}) = Tx - Tx_{k-1} - hTf(x) \\ &= TF_k(T^{-1}\tilde{x}) \; . \end{aligned}$$

This holds in a similar way for Runge-Kutta and BDF methods. According to such a transformation of F_k to $\tilde{F}_k$ the iteration (3) should yield $\tilde{x}^{(i)} = Tx^{(i)}$ for all iterations i when a linear transformation $\tilde{x}^{(0)} = Tx^{(0)}$ is applied to the state. Analysing the TR1 update, one obtains the following result with respect to scaling invariance:

Theorem 1 (Conditions for Scaling Invariance).
Suppose $T \in \mathbb{R}^{n \times n}$ is a regular matrix and $F : \mathbb{R}^n \to \mathbb{R}^n$ a given vector function. For $x \in \mathbb{R}^n$, define $\tilde{x} = Tx$ and $\tilde{F} : \mathbb{R}^n \to \mathbb{R}^n$ with $\tilde{F}(\tilde{x}) = TF(T^{-1}\tilde{x})$. Then for the rank-1 updates given by

$$A_{i+1} = A_i + \frac{(F'(x^{(i+1)}) - A_i)s_i z_i^T (F'(x^{(i+1)}) - A_i)}{z_i^T (F'(x^{(i+1)}) - A_i)s_i} \quad \textit{and} \tag{10}$$

$$\tilde{A}_{i+1} = \tilde{A}_i + \frac{(\tilde{F}'(\tilde{x}^{(i+1)}) - \tilde{A}_i)\tilde{s}_i \tilde{z}_i^T (\tilde{F}'(\tilde{x}^{(i+1)}) - \tilde{A}_i)}{\tilde{z}_i^T (\tilde{F}'(\tilde{x}^{(i+1)}) - \tilde{A}_i)\tilde{s}_i} \,, \tag{11}$$

one has that $\tilde{A}_i = TA_iT^{-1}$ *holds for all* i *if*

$$\tilde{x}^{(0)} = Tx^{(0)}, \qquad \tilde{A}_0 = TA_0T^{-1}, \qquad \tilde{s}_i = Ts_i, \quad \tilde{z}_i = T^{-T}z_i \tag{12}$$

is valid for all i*.*

Proof. We prove the assertion by induction. For $i = 0$, we have

$$\tilde{x}^{(1)} = \tilde{x}^{(0)} - \tilde{A}_0^{-1}\tilde{F}(\tilde{x}^{(0)}) = T\left(x^{(0)} - A_0^{-1}F(x^{(0)})\right) = Tx^{(1)} \,.$$

It follows immediately that $\tilde{F}'(\tilde{x}^{(1)}) = TF'(x^{(1)})T^{-1}$, and we obtain

$$\begin{aligned}
\tilde{A}_1 &= TA_0T^{-1} + \\
&\quad \frac{(TF'(x^{(1)})T^{-1} - TA_0T^{-1})Ts_0 z_0^T T^{-1}(TF'(x^{(1)})T^{-1} - TA_0T^{-1})}{z_0^T T^{-1}(TF'(x^{(1)})T^{-1} - TA_0T^{-1})Ts_0} \\
&= TA_0T^{-1} + T\frac{(F'(x^{(1)}) - A_0)s_0 z_0^T (F'(x^{(1)}) - A_0)}{z_0^T (F'(x^{(1)}) - A_0)s_0}T^{-1} \\
&= T\left(A_0 + \frac{(F'(x^{(1)}) - A_0)s_0 z_0^T (F'(x^{(1)}) - A_0)}{z_0^T (F'(x^{(1)}) - A_0)s_0}\right)T^{-1} \\
&= TA_1T^{-1} \,,
\end{aligned}$$

and the assertion is shown for $i = 0$. The induction step $i \mapsto i + 1$ can be proven in the same way by replacing the subscripts 0 and 1 with the subscripts i and $i + 1$, respectively, in the last two equations.

For the direction $s_i = x^{(i+1)} - x^{(i)}$, condition $\tilde{s}_i = Ts_i$ is naturally fulfilled. Unfortunately, this is not true for the weight vector $z_i = F_k(x^{(i+1)})$ used in the Least-squares update. Hence, it yields only the same invariance properties with respect to scaling in the range as the Bad Broyden update.

We must also consider that the denominator in (7) might vanish before the iteration converges. In this situation, several strategies are conceivable. One approach perturbs the vectors s_i and z_i. In the implementation, we choose to perform no update and reuse the current approximation. Alternatively, we could choose the adjoint direction as $z_i = (F'(x^{(i+1)}) - A_i)s_i$, making the denominator greater than zero as long as the iteration did not yet converge and the approximation is not exact. However, this update is not invariant with respect to linear transformations in the state space. Therefore, we also present an alternative definition of z_i to maintain full transformation invariance in the domain and range.

Adjoint approach

Quite often one has a problem-dependent functional $\phi(x)$ in addition to the initial value problem to be solved, e.g. the output of one product, the concentration of

all ingredients for a chemical reaction or the total loss of energy. Then one can use an adjoint vector to quantify the influence of discretization errors or errors in the solution of (2) on the problem-dependent functional $\phi(x)$. For that purpose, we define the adjoint vector $\lambda \in \mathbb{R}^n$ as solution of the adjoint system

$$G_{x_k}(\lambda) := F_k'(x_k)^T \lambda - \nabla\phi(x_k) = 0 \,. \tag{13}$$

Consequently λ can be interpreted as the sensitivity of $\phi(x_k)$ with respect to changes in the equation $F_k(x_k) = 0$. Solving (13) by the quasi-Newton iteration

$$\lambda^{(i+1)} = \lambda^{(i)} - A_i^{-T} \left(F'(x^{(i+1)})^T \lambda^{(i)} - \phi(x^{(i+1)}) \right) \tag{14}$$

yields the direct tangent condition $A_{i+1}^T \sigma_i = F'(x^{(i+1)})^T \sigma_i$ with $\sigma_i = \lambda^{(i+1)} - \lambda^{(i)}$ for the system $G_{x_k}(\lambda) = 0$. This is equivalent to an adjoint tangent condition with $z_i = \sigma_i$ for the system $F_k(x) = 0$. Hence, two quasi-Newton iterations are performed simultaneously: The first one solves (2), and the second one solves (13). However, due to the definition of both nonlinear systems of equations, the system matrix is exactly the same and therefore can be reused. The second approach is called *Adjoint update*. Because the functional ϕ relates to the problem, it depends on the state, too. Therefore, a transformation of x forces a consistent transformation of ϕ which ensures the transformation invariance. To prove this assertion, we first show the following theorem:

Theorem 2 (Scaling Invariance of the Adjoint Information).
Suppose $T \in \mathbb{R}^{n\times n}$ is a regular matrix, and $F : \mathbb{R}^n \to \mathbb{R}^n$ is a vector function. For $x \in \mathbb{R}^n$, define $\tilde{x} = Tx$ and $\tilde{F} : \mathbb{R}^n \to \mathbb{R}^n$ with $F(\tilde{x}) = TF(T^{-1}\tilde{x})$. Furthermore, assume that $\phi, \tilde{\phi} : \mathbb{R}^n \to \mathbb{R}$ are given with $\tilde{\phi}(\tilde{x}) = \phi(x)$. Let $\lambda, \tilde{\lambda} : \mathbb{R}^n \to L(\mathbb{R}^n, \mathbb{R})$ be the solutions of

$$\lambda(x)^T F'(x) = \nabla\phi(x)^T \quad \textit{and} \quad \tilde{\lambda}(\tilde{x})^T \tilde{F}'(\tilde{x}) = \tilde{\nabla}\tilde{\phi}(\tilde{x})^T \,.$$

Then

$$\tilde{\lambda}(\tilde{x})^T = \lambda(x)^T T^{-1} \,. \tag{15}$$

Proof. One has $\tilde{F}'(\tilde{x})^{-1} = TF'(x)^{-1}T^{-1}$. Furthermore, the equality $\tilde{\phi}(\tilde{x}) = \phi(T^{-1}\tilde{x})$ holds. It follows that

$$\tilde{\nabla}\tilde{\phi}(\tilde{x})^T = \frac{d}{d\tilde{x}}\tilde{\phi}(\tilde{x}) = \frac{d}{d\tilde{x}}\phi(\underbrace{T^{-1}\tilde{x}}_{x}) = \frac{d}{dx}\phi(x)T^{-1} = \nabla\phi(x)^T T^{-1} \,.$$

Therefore, we obtain

$$\tilde{\lambda}(\tilde{x})^T = \tilde{\nabla}\tilde{\phi}(\tilde{x})^T \tilde{F}'(\tilde{x})^{-1} = \nabla\phi(x)^T T^{-1} TF'(x)^{-1}T^{-1} = \lambda(x)^T T^{-1} \,,$$

and the assertion is proven.

The property (15) can be transfered directly to the quasi-Newton iteration to solve (13) if $\tilde{\lambda}^{(0)} = T^{-T}\lambda^{(0)}$. Hence, we obtain

$$\tilde{z}_i = \tilde{\lambda}^{(i+1)} - \tilde{\lambda}^{(i)} = T^{-T}(\lambda^{(i+1)} - \lambda^{(i)}) = T^{-T} z_i \,.$$

Therefore, the adjoint update is scaling invariant. Although it did not occour in the numerical tests, with this update, it is also possible that the denominator in (7) vanishes. Perturbing the directions s_i and z_i would destroy the transformational invariance. Therefore, performing no update is appropriate to maintain invariance of the iteration.

3 Implementation Details

The software to test and compare the proposed rank-1 updates is written in C/C++. It provides three different integration methods, the implicit Euler method, the 3-stage Radau IIA method, and BDF formulas [249]. Applying the implicit Euler method, the system to be solved is $F_k(x) = x - x_{k-1} - hf(x) = 0$ and has dimension n. The convergence order of this method is one. Using the 3-stage Runge-Kutta method Radau IIA, one must solve a nonlinear system of dimension $3n$. Hence, the complexity increases, but the method has order five. The BDF formulas correspond to linear multi-step methods, where the system of equations is given by

$$F_k(x) = \alpha_0 x - \sum_{j=1}^{l} \alpha_j x_{k-j} - h_k f(x) = 0 \in \mathbb{R}^n$$

with certain scalars α_j. These methods are of order 1.

For the solution of the nonlinear systems, we implemented Newton's method in the following way. To compute the complete Jacobian of the right hand side function $f(x(t))$, we employ the AD-tool ADOL-C [229] that provides exact first and higher order derivatives for C/C++ function evaluations using operator overloading. Subsequently, the Jacobian of the nonlinear system is easily computed from the Jacobian of the right-hand side using vector forward mode of AD. Finally, a QR-factorization is performed to compute the next Newton step.

Furthermore, we coded Broyden's method as well as the two new quasi-Newton approaches to approximate the Jacobian information during the solution of the nonlinear system. Once more, we maintain a QR-factorization of the corresponding updates in order to compute the next iteration step efficiently. As a starting point we compute the exact Jacobian for the initial value x_0. The derivative information required by the TR1 update, namely $F_k'(x^{(i+1)})s_i$ and $z_i^T F_k'(x^{(i+1)})$, are calculated using the scalar forward and reverse mode provided by ADOL-C.

For stabilizing the Newton as well as the quasi-Newton approach we perform a line search with quadratic and cubic interpolation, respectively, as described in [143]. Furthermore, we incorporated a simple adaptive time stepping according to the approach analyzed in [474].

4 Numerical Results

For the numerical tests, we take two initial value problems from the *Testset for Initial Value Problem Solvers*, University of Bary, Italy [361]. The first one is the *Pollution Problem*, a stiff system of 20 non-linear ODEs. It describes a chemical reaction as part of the air pollution model developed at The Dutch National Institute of Public Health and Environmental Protection (RIVM), and consists of 25 reaction and 20 reacting compounds. The second test is the *Medical Akzo Nobel Problem* consisting originally of two partial differential equations. Semi-discretization of this system yields 400 stiff ODEs. The Akzo Nobel research laboratories formulated this problem in their study of the penetration of radio-labeled antibodies into a tissue that has been infected by a tumor. In both problems, the right-hand side of the ODE is nonlinear.

Pollution Problem

To suit our software, we reformulate the Pollution Problem as an autonomous ODE system with 21 component functions. For comparison, the tests were performed using Newton's method with AD-based Jacobians as described in Sect. 3, the Broyden update, which is a secant method using only information of F, and the two presented variants of the TR1 update. Since the Pollution Problem describes a chemical reaction, we chose the problem-dependent functional $\phi(x)$ to be the concentration of CO_2. For this problem we use adaptive time stepping, where the step size criteria are the same for all numerical tests.

Table 1. Euler method with adaptive time stepping

	$h_0 = 10^{-2}$				$h_0 = 10^{-3}$			
	Newt	LS	Adj	Broy	Newt	LS	Adj	Broy
time steps	412	412	412	—	4077	4077	4077	4077
iterations	713	716	918	—	4512	4504	4521	11319
CPU-time (s)	0.33	0.15	0.20	—	2.07	0.98	1.05	1.69

The numerical results achieved with the Euler method are given in Table 1. The integration was performed for the time interval $[0, 60]$, i.e. $T = 60$. The integration fails using a larger time step h_0 as initialization if Broyden's method is applied. All other approaches yield the results reported as solutions at the test suite website [361]. The number of time steps is the same for all methods where the integration over the whole time interval was possible. However, the numbers of iterations for solving the nonlinear systems differ remarkably. These numbers are again almost the same for Newton's approach and the Least-squares update, i.e. the TR1 update with $z_i = F_k(x^{(i+1)})$, but due to the computation of the complete Jacobian and its factorization required for Newton's method, the corresponding run time is naturally significantly larger. The iteration count for the adjoint update, i.e. the TR1 update with z_i based on the problem dependent function, is higher which is reflected in the run times. Since one has to perform two reverse mode differentiations the factor between the run times is larger than the factor between the iteration counts. The iteration count for Broyden's method is even higher, but since no derivative calculations are performed, the run time is less than the run time for the Newton's method.

Table 2. Radau IIA with adaptive time stepping

	$h_0 = 10^{-2}$				$h_0 = 10^{-3}$			
	Newt	LS	Adj	Broy	Newt	LS	Adj	Broy
time steps	412	412	412	412	565	565	565	—
iterations	708	723	1081	2661	1037	1041	1462	—
CPU-time (s)	5.66	0.85	1.30	2.42	8.34	1.24	1.80	—

The numerical results achieved with Radau IIA and with two BDF formulas, i.e. $l = 3$ and $l = 6$, are given in Tables 2 and 3, respectively. Once more, the integration was performed for the time interval $[0, 60]$ to verify the results. The integration fails using a smaller time step h_0 as initialization (Radau IIA) or a higher order method (BDF) if Broyden's method is applied. The numbers for the methods where

Table 3. BDF-formula with adaptive time stepping

	$l = 3$				$l = 6$			
	Newt	LS	Adj	Broy	Newt	LS	Adj	Broy
time steps	412	412	412	412	412	412	412	—
iterations	693	702	929	2144	690	695	933	—
CPU-time (s)	0.37	0.21	0.27	0.40	0.37	0.19	0.28	—

the integration converges confirm the behaviour of the solution methods for the nonlinear system of equations already observed for the Euler method.

Medical Akzo Nobel Problem

We reformulate this problem as an autonomous ODE system yielding a system of 401 ODEs. For this example, it was not possible to get results using Broyden's method despite intensive testing with respect to step sizes. As the functional ϕ in the adjoint update, we choose the product of the concentrations of the reacting components. Furthermore, we do not apply varying step sizes since our step size heuristic is not appropriate for this problem, and the sparsity of the Jacobian is not taken into account.

Table 4. Euler method with constant time steps.

	$h = 10^{-1}$			$h = 10^{-2}$		
	Newt	LS	Adj	Newt	LS	Adj
time steps	200	200	200	2000	2000	2000
iterations	570	1290	2012	3912	4891	7121
CPU-time (s)	1209.62	70.54	124.01	8370.17	254.39	425.75

The numerical results achieved with the Euler method are given in Table 4. All approaches yield the results reported as solutions at the test suite website [361]. However, it was necessary to provide the exact Jacobian at $t = 5$ if $h = 10^{-1}$ applying the TR1 updates since the right hand side jumps exactly at that place and therefore is not continuous. The iteration increases when using the inexact derivative information provided by the Least-squares and Adjoint update. However, due to the lower cost to perform one iteration, the factor of the run times is 17 for the Least-squares and 10 for the Adjoint update with $h = 10^{-1}$, and 33 for the Least-squares and 20 for the Adjoint update with $h = 10^{-2}$.

This observation is also confirmed by examining the run times needed for the calculation of one Jacobian and its factorization compared to one rank-1 update of an approximation. The computation of the Jacobian needed 0.051 seconds, while its factorization lasts 2.1 seconds. In contrast to this, computing the new factorized approximation in the Least-squares update only needs 0.054 seconds. This shows that the main computing effort lies in the factorization of the Jacobian to solve the linear system.

The numerical results achieved with Radau IIA and with two BDF formulas, i.e. $l = 3$ and $l = 6$, are given in Tables 5 and 6, respectively. The behaviour already observed for the Euler method is confirmed: The iteration count increases due to

the approximation of the Jacobian, but the overall run time is drastically reduced due to the much lower computation effort required by one quasi-Newton iteration in comparison to the calculation and factorization of the complete Jacobian.

Table 5. Radau IIA with constant time steps

	$h = 10^{-1}$		
	Newt	LS	Adj
time steps	200	200	200
iterations	591	1705	2842
CPU-time (s)	32175.02	988.96	1756.97

Table 6. BDF-formula with constant time steps $h = 10^{-2}$

	$l = 3$			$l = 6$		
	Newt	LS	Adj	Newt	LS	Adj
time steps	2000	2000	2000	2000	2000	2000
iterations	3880	4771	6684	3798	4667	6730
CPU-time (s)	8228.23	266.97	418.12	7932.48	266.80	425.66

5 Conclusions and Outlook

The use of quasi-Newton methods for the solution of nonlinear systems arising during the integration of stiff ODEs is not widespread. We present two new variants of the Two-sided Rank-1 update (TR1). These AD-based quasi-Newton methods fulfill the exact direct tangent condition as well as the exact adjoint tangent condition. Here, the selective choice of tangents and adjoints facilitates invariance and therefore norm independence of the state space. The proposed update formulas were tested using a well-known IVP test suite. For the examples considered during this project, the achieved numerical results are very promising. Usually the Least-squares and the Adjoint updates perform significantly better than Broyden's method. However, there are several open questions. First, detailed convergence analysis of the TR1-update for the solution of nonlinear equations is needed. This theoretical examination may also motivate alternative choices of s_i and z_i. Additionally, the maintaining or adjustment of a suitable factorization in the case of varying time step sizes has to be studied for a successful integration of the quasi-Newton methods for the integration of stiff ODEs. Here the task is to find a factorization that allows a change in the step size without performing a complete factorization again.

Reduction of Storage Requirement by Checkpointing for Time-Dependent Optimal Control Problems in ODEs

Julia Sternberg[1] and Andreas Griewank[2]

[1] Technische Universität Dresden, Institut für Wissenschaftliches Rechnen, Germany
jstern@math.tu-dresden.de
[2] Humboldt-Universität zu Berlin, Institut für Mathematik, Germany
griewank@mathematik.hu-berlin.de

Summary. We consider a time-dependent optimal control problem, where the state evolution is described by an ODE. There is a variety of methods for the treatment of such problems. We prefer to view them as boundary value problems and apply to them the Riccati approach for non-linear BVPs with separated boundary conditions.

There are many relationships between multiple shooting techniques, the Riccati approach and the Pantoja method, which describes a computationally efficient stage-wise construction of the Newton direction for the discrete-time optimal control problem. We present an efficient implementation of this approach. Furthermore, the well-known checkpointing approach is extended to a "nested checkpointing" for multiple transversals. Some heuristics are introduced for an efficient construction of nested reversal schedules. We discuss their benefits and compare their results to the optimal schedules computed by exhaustive search techniques.

Key words: Optimal control, Newton's method, Riccati approach, nested checkpointing, surface hardening of steel

1 Introduction

Consider the following unconstrained primal control problem

$$\min_{\mathbf{u}} \phi(\mathbf{x}(T)) \ , \tag{1}$$

where the system is described by

$$\dot{\mathbf{x}} = f(\mathbf{x}(t), \mathbf{u}(t), t), \quad \mathbf{x}(0) = \mathbf{x}_0 \ . \tag{2}$$

Here, $\mathbf{x} : [0,T] \to \mathbf{R}^n$, $\mathbf{u} : [0,T] \to \mathbf{R}^m$, $f : \mathbf{R}^n \times \mathbf{R}^m \times [0,T] \to \mathbf{R}$, and $\phi : \mathbf{R}^n \to \mathbf{R}$. The task is to find the function $\mathbf{u}(t)$ that minimizes (1). To characterize an optimal

control function $\mathbf{u}(t)$ for the minimization problem (1) and (2) we consider the following adjoint state equation

$$\dot{\bar{\mathbf{x}}} = -H_{\mathbf{x}}^T = -f_{\mathbf{x}}^T\, \bar{\mathbf{x}}, \qquad \bar{\mathbf{x}}(T) = \phi_{\mathbf{x}}^T(\mathbf{x}(T)) \;, \tag{3}$$

where the **Hamiltonian** function H is given by

$$H(\mathbf{x}(t), \mathbf{u}(t), \bar{\mathbf{x}}(t), t) = \bar{\mathbf{x}}^T(t)\, f(\mathbf{x}(t), \mathbf{u}(t), t) \;.$$

Here $\dot{\bar{\mathbf{x}}}$ represents the total time derivative of $\bar{\mathbf{x}}$ rather than a directional derivative as is customary in parts of the AD literature. At each point along the solution path the Hamiltonian function must be minimal with respect to the control value $\mathbf{u}(t)$. Therefore, for the optimal control problem (1)–(2), we have the **First Order Necessary Optimality Condition**

$$\left(\frac{\partial H}{\partial \mathbf{u}}\right)^T = 0, \qquad 0 \le t \le T \;. \tag{4}$$

Many numerical methods for solving optimal control problems have been proposed and used in various applications. Relationships among them are often not clear due to the lack of a generally accepted terminology. One popular concept is to juxtapose approaches that first discretize and then optimize with those that first optimize and then discretize. Methods of the first type are sometimes called **direct** (see e.g. [85]) as they treat the discretized control problem immediately as a finite dimensional nonlinear program, which can be handed over to increasingly sophisticated and robust NLP codes. In the alternative approach one first derives optimality conditions in a suitable function space setting and then discretizes the resulting boundary value problem with algebraic side constraints. Often such **indirect** methods (see e.g. [358]) yield highly accurate results, but they have some disadvantages as well. Sometimes it is not possible to construct the boundary value problem explicitly, as it requires that we can express the control function $\mathbf{u}$ in terms of $\mathbf{x}$ and $\bar{\mathbf{x}}$ from the relation (4). The second disadvantage is that often we have to find a very good initial guess including good estimates for the adjoint variables to achieve convergence to the solution. Alternatively one can solve the problem as a DAE with (4) representing a possibly discontinuous algebraic constraint.

There is a range of intermediate strategies. For example, one may discretize first the controls and later the states. Christianson [114] makes a different distinction between direct and indirect methods, depending on whether the adjoint variables are integrated only backward or also forward. For stability reasons we consider here only the first option and show how the memory requirement can be kept within reasonable bounds.

In general, the BVP (2)–(3) is non-linear with separated boundary condition (BC). We use a quasilinearization scheme to solve it iteratively. First, we linearize (2), (3), and (4). Then we solve the resulting linear BVP using the Riccati approach.

Section 2 of this paper introduces the quasilinearization scheme. Nested checkpointing techniques and their properties are discussed in Sect. 3. Section 4 gives a numerical example, and in Sect. 5 we present some conclusions.

2 Quasilinearization Techniques

In this section we introduce quasilinearization techniques, which can be applied for a stable solution of the optimal control problem (1)–(2).

2.1 Quasilinearization Scheme

We linearize (2), (3), and (4) about a reference solution $\mathbf{x}(t)$, $\mathbf{u}(t)$, $\bar{\mathbf{x}}(t)$ and obtain the following equations for variations $\delta\mathbf{x}(t)$, $\delta\bar{\mathbf{x}}(t)$, and $\delta\mathbf{u}(t)$:

$$\delta\dot{\mathbf{x}} - f_{\mathbf{x}}\delta\mathbf{x} - f_{\mathbf{u}}\delta\mathbf{u} = 0 \,, \tag{5}$$

$$\delta\dot{\bar{\mathbf{x}}} + H^T_{\mathbf{xx}}\delta\mathbf{x} + H^T_{\mathbf{xu}}\delta\mathbf{u} + H^T_{\mathbf{x\bar{x}}}\delta\bar{\mathbf{x}} = 0 \,, \tag{6}$$

$$H^T_{\mathbf{u}} + H^T_{\mathbf{ux}}\,\delta\mathbf{x} + H^T_{\mathbf{uu}}\,\delta\mathbf{u} + H^T_{\mathbf{u\bar{x}}}\,\delta\bar{\mathbf{x}} = 0 \,, \tag{7}$$

with the linearized initial and terminal conditions

$$\begin{pmatrix} I & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} \delta\mathbf{x}(0) \\ \delta\bar{\mathbf{x}}(0) \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ -\phi_{\mathbf{xx}}(T) & I \end{pmatrix}\begin{pmatrix} \delta\mathbf{x}(T) \\ \delta\bar{\mathbf{x}}(T) \end{pmatrix} = -\begin{pmatrix} \mathbf{x}(0) - \mathbf{x}_0 \\ \bar{\mathbf{x}}(T) - \phi^T_{\mathbf{x}}(T) \end{pmatrix} \,. \tag{8}$$

After expressing $\delta\mathbf{u}$ in terms of $\delta\mathbf{x}$ and $\delta\bar{\mathbf{x}}$ from the relation (7) we obtain

$$\delta\mathbf{u} = -H^{-1}_{\mathbf{uu}}\,\left(H^T_{\mathbf{u}} + H^T_{\mathbf{ux}}\,\delta\mathbf{x} + H^T_{\mathbf{u\bar{x}}}\,\delta\bar{\mathbf{x}}\right) \,.$$

Substituting of this expression into (5)–(6) yields the following linear BVP:

$$\begin{pmatrix} \delta\dot{\mathbf{x}} \\ \delta\dot{\bar{\mathbf{x}}} \end{pmatrix} = S(t)\begin{pmatrix} \delta\mathbf{x} \\ \delta\bar{\mathbf{x}} \end{pmatrix} + q(t) \,, \tag{9}$$

where

$$S(t) = \begin{pmatrix} S^{11} & S^{12} \\ S^{21} & S^{22} \end{pmatrix} = \begin{pmatrix} f_{\mathbf{x}} - f_{\mathbf{u}}\,H^{-1}_{\mathbf{uu}}\,H_{\mathbf{xu}} & | & -f_{\mathbf{u}}\,H^{-1}_{\mathbf{uu}}\,f^T_{\mathbf{u}} \\ -H_{\mathbf{xx}} + H_{\mathbf{ux}}\,H^{-1}_{\mathbf{uu}}\,H_{\mathbf{xu}} & | & H_{\mathbf{ux}}\,H^{-1}_{\mathbf{uu}}\,f^T_{\mathbf{u}} - f^T_{\mathbf{x}} \end{pmatrix}$$

is the system matrix, and

$$q(t) = \begin{pmatrix} q^1 \\ q^2 \end{pmatrix} = \begin{pmatrix} -f_{\mathbf{u}}\,H^{-1}_{\mathbf{uu}}\,H^T_{\mathbf{u}} \\ H_{\mathbf{ux}}\,H^{-1}_{\mathbf{uu}}\,H^T_{\mathbf{u}} \end{pmatrix}$$

is the non-homogeneous part. The BC for this problem are given by the relation (8). Rather than solving this linear BVP using collocation or another 'global' discretization scheme we prefer the Riccati approach which computes the solution in a sequence of forward and backward sweeps through the time interval $[0, T]$. To achieve a suitable decoupling of the solution components we consider a linear transformation of the form

$$\begin{pmatrix} \delta\mathbf{x}(t) \\ \delta\bar{\mathbf{x}}(t) \end{pmatrix} = \begin{pmatrix} I & 0 \\ K(t) & I \end{pmatrix}\begin{pmatrix} \delta\mathbf{x}(t) \\ a(t) \end{pmatrix} \,.$$

To determine a suitable $K(t)$ and the corresponding $a(t)$, we substitute $\begin{pmatrix} \delta\mathbf{x} \\ \delta\bar{\mathbf{x}} \end{pmatrix}$ in terms of $\begin{pmatrix} \delta\mathbf{x} \\ a \end{pmatrix}$ in (9), which yields

$$\frac{d}{dt}\begin{pmatrix} I & 0 \\ K & I \end{pmatrix}\begin{pmatrix} \delta\mathbf{x} \\ a \end{pmatrix} = \begin{pmatrix} S^{11} & S^{12} \\ S^{21} & S^{22} \end{pmatrix}\begin{pmatrix} I & 0 \\ K & I \end{pmatrix}\begin{pmatrix} \delta\mathbf{x} \\ a \end{pmatrix} + \begin{pmatrix} q^1 \\ q^2 \end{pmatrix} .$$

Now if we choose $K(t)$ as the solution of the Riccati equation

$$\dot{K}(t) = S^{21} - K(t)S^{11} + S^{22}K(t) - K(t)S^{12}K(t) ,$$

then the new variables $(\delta\mathbf{x}, a)$ satisfy the block system

$$\begin{pmatrix} \delta\dot{\mathbf{x}} \\ \dot{a} \end{pmatrix} = \left(\begin{array}{cc} S^{11} + S^{12}K & \mid S^{12} \\ 0 & \mid S^{22} - KS^{12} \end{array}\right)\begin{pmatrix} \delta\mathbf{x} \\ a \end{pmatrix} + \begin{pmatrix} q^1 \\ q^2 - Kq^1 \end{pmatrix} ,$$

with the separated BC

$$\begin{pmatrix} I & 0 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} \delta\mathbf{x}(0) \\ a(0) \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ -\phi_{\mathbf{xx}}(T) + K(T) & I \end{pmatrix}\begin{pmatrix} \delta\mathbf{x}(T) \\ a(T) \end{pmatrix} = -\begin{pmatrix} \mathbf{x}(0) - \mathbf{x}_0 \\ \bar{\mathbf{x}}(T) - \phi_{\mathbf{x}}^T(T) \end{pmatrix} .$$

This approach leads to Algorithm 1.

Algorithm 1 Quasilinearization scheme for solving optimal control problems using the Riccati method

Choose initial control trajectory $\mathbf{u}^0(t)$, $t \in [0,T]$, $k = 0$.

Do:

Original initialization:

$\mathbf{x}^k(0) = \mathbf{x}_0$.

Original sweep: $t : 0 \to T$

Integrate forward $\dot{\mathbf{x}}^k = f(\mathbf{x}^k(t), \mathbf{u}^k(t), t)$.

Adjoint initialization:

Set $\bar{\mathbf{x}}^k(T) = \phi_{\mathbf{x}}^T(T)$.

Set $K(T) = \phi_{\mathbf{xx}}(T)$ *and* $a(T) = 0$.

Adjoint sweep: $t : T \to 0$

Integrate backward $\dot{\bar{\mathbf{x}}}^k = -f_{\mathbf{x}}^T(\mathbf{x}^k(t), \mathbf{u}^k(t), t)\ \bar{\mathbf{x}}^k$.

Integrate backward $\dot{K}(t) = S^{21} - K(t)S^{11} + S^{22}K(t) - K(t)S^{12}K(t)$.

Integrate backward $\dot{a}(t) = \left(-K(t)S^{12} + S^{22}\right)a(t) - K(t)q^1 + q^2$.

Final initialization:

Set $\delta\mathbf{x}^k(0) = 0$.

Final sweep: $t : 0 \to T$

Integrate forward $\delta\dot{\mathbf{x}}^k = (S^{11} + S^{12}K)\delta\mathbf{x}^k + S^{12}a + q^1$.

Evaluate $\delta\mathbf{u}^k = -H_{\mathbf{uu}}^{-1}\left(H_{\mathbf{u}}^T + H_{\mathbf{ux}}^T\,\delta\mathbf{x}^k + H_{\mathbf{u\bar{x}}}^T(K\delta\mathbf{x}^k + a)\right)$.

$\mathbf{u}^{k+1}(t) = \mathbf{u}^k(t) + \delta\mathbf{u}^k(t)$.

$\mathbf{k} = \mathbf{k} + \mathbf{1}$.

While: $\|\delta\mathbf{u}^k(t)\|_2 \geq TOL$ *and* $k < MAX\_ITER$.

Discretizing the scheme in time, one obtains Pantoja's method [113,427], which represents a computationally efficient stage-wise construction of the Newton direction for the discrete-time optimal control problem. Moreover, this scheme can also be viewed as Newton's method applied to the solution of the non-linear BVP (2), (3), and (4) using a particular LU-matrix factorization. The relation between Algorithm 1 and Pantoja's method is discussed in detail in [488].

2.2 Information Flow by the Quasilinearization Scheme

From Algorithm 1 we can see that each iteration of the quasilinearization scheme consists of three sweeps through the time window $[0, T]$, which are referred to as **original**, **adjoint**, and **final** sweep. Figure 1 shows the dimensions of the data objects flowing between these three sweeps. The horizontal arrows represent informational flow between the three sweeps that are represented by slanted lines. Two cameras pointing at the original and adjoint sweeps represent the information which has to be stored if the current composite state is saved as a checkpoint. Here $B(t)$ is a $n \times m$ matrix path that must be communicated from the adjoint to the final sweep.

In any case the adjoint sweep requires much more computational effort than the original and the final sweeps because it involves matrix computations and factorizations. The final sweep proceeds forward in time and propagates vectors of dimension $(n + m)$. Computations on the final sweep proceed as soon as required information from the previous sweeps is available. The final sweep can be combined with the original sweep of a subsequent Newton step.

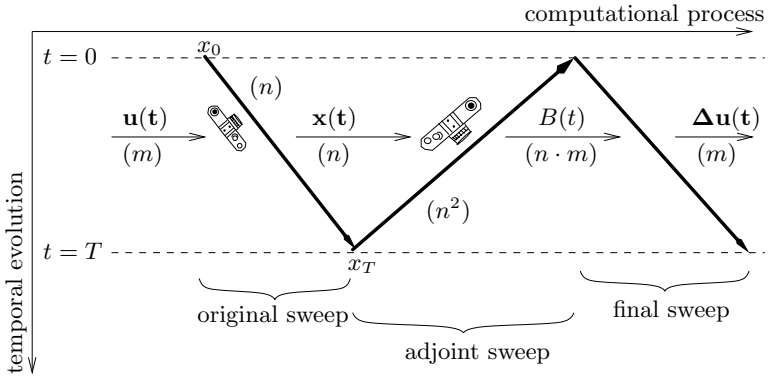


Fig. 1. Information flow for Riccati/Pantoja computation of Newton step.

The simplest strategy is to implement Algorithm 1 with straightforwardly storing all intermediate states of each sweep on a sequential data file and to restore them when they are needed. The memory requirement for the basic algorithm, where all intermediate values are stored, is of order $\mathcal{O}(l\, n^2)$, where l gives the number of time steps between 0 and T. However, this approach can be realized only when there is a sufficiently large amount of memory available. If this is not the case then we can apply checkpointing techniques.

As developed in [224, 225, 489] checkpointing means that not all intermediate states are saved but only a small subset of them is stored as checkpoints. In previous work we have treated cases where checkpoints are stored only for a reversal consisting of a single forward and an adjoint, or reverse sweep. But because of the triple sweep within each Newton iteration (see Algorithm 1 and Fig. 1) we are faced here with a new kind of checkpointing task. Since now checkpoints from various sweeps must be kept simultaneously, we refer to this situation as **nested checkpointing**.

Since the information to be stored on the original sweep differs from that needed on the adjoint sweep, we have two classes of checkpoints. Hence, we call the checkpoints **thin** on the original sweep and **fat** on the adjoint sweep. Thin checkpoints save a state space of dimension n, and fat checkpoints save a state space whose size has order n^2. While the length of steps may vary arbitrarily with respect to the physical time increment they represent, we assume throughout that the total number l of time steps is a priori known. When this is not the case, an upper bound on l may be used, which results in some loss of efficiency. Fully adaptive nested reversal schedules are under development.

3 Nested Reversal Schedules

In the present section we introduce a formal concept for nested checkpointing. Some heuristics are introduced for an efficient construction of nested reversal schedules. We discuss their benefits and compare their results to the optimal schedules computed by exhaustive search techniques.

3.1 Formalism

Let us consider a multiple sweep evolution $\mathcal{E}_{3(l)}$ containing three sweeps. Each sweep consists of l consecutive time steps. An example of such an evolution is shown in Fig. 2. Time steps are shown as horizontal arrows. Their directions denote the information flow. Nodes denote different intermediate states. Each sweep is characterized

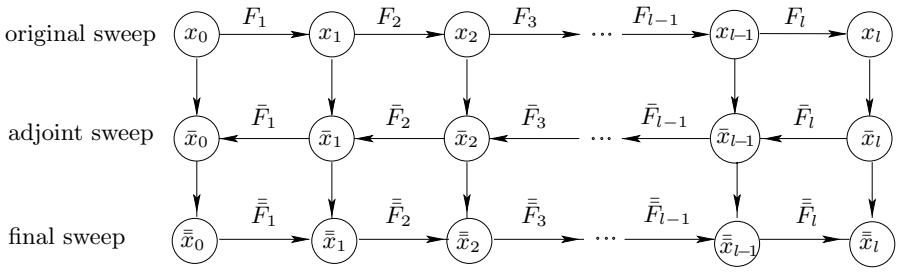


Fig. 2. Multiple sweep evolution $\mathcal{E}_{3(l)}$.

by a specified direction, i.e. direction of horizontal arrows within a single sweep. A direction shows the corresponding information flow between neighboring intermediate states of a single sweep. The information flow within a single sweep has a constant direction. An additional information flow exists between nodes, which are intermediate states of different successive sweeps, shown in Fig. 2 by vertical lines.

We denote intermediate states of the original, adjoint, and final sweeps as x_i, $\bar{x}_i$, and $\bar{\bar{x}}_i$, $0 \le i \le l$, respectively. In the same manner we identify intermediate steps or time steps of various sweeps as F_i, $\bar{F}_i$, and $\bar{\bar{F}}_i$, $1 \le i \le l$. Then we have

$$x_i = F_i(x_{i-1}) \,, \quad \bar{x}_{i-1} = \bar{F}_i(x_{i-1}, \bar{x}_i) \,, \quad \bar{\bar{x}}_i = \bar{\bar{F}}_i(\bar{x}_i, \bar{\bar{x}}_{i-1}) \,, \quad 1 \le i \le l \,.$$

We assume that dimensions of intermediate states within a single sweep are constant. Therefore, we denote them as

$$d \equiv \dim\{x_i\}, \quad \bar{d} \equiv \dim\{\bar{x}_i\}, \quad \bar{\bar{d}} \equiv \dim\{\bar{\bar{x}}_i\}, \quad 0 \le i \le l .$$

Moreover, we introduce evaluation costs, i.e., the computational effort for intermediate steps of different sweeps. We assume that within each single sweep we have uniform step costs, i.e., there exist three constants t, $\bar{t}$, and $\bar{\bar{t}}$ such that

$$t \equiv \mathrm{TIME}(F_i), \quad \bar{t} \equiv \mathrm{TIME}(\bar{F}_i), \quad \bar{\bar{t}} \equiv \mathrm{TIME}(\bar{\bar{F}}_i), \quad 1 \le i \le l .$$

Further, we assume that

$$\bar{d} >> d \quad \text{and} \quad \bar{t} >> t . \tag{10}$$

Thus, the dimension $\bar{d}$ of an intermediate state of an adjoint sweep is much higher than the dimension d of an intermediate state on the original sweep. Correspondingly the evaluation of time steps during an adjoint sweep is much more expensive than the evaluation of time steps during an original sweep. The assumptions (10) agree with the scenario presented in the Algorithm 1.

3.2 Definition of Nested Reversal Schedules and Its Characteristics

The goal is to implement an evolution $\mathcal{E}_{3(l)}$ using nested checkpointing. The question is how to place different checkpoints to implement the evolution $\mathcal{E}_{3(l)}$ most efficiently. We call each possible strategy a **nested reversal schedule** because checkpoints are set and released at two different levels. Thus, it is not required to store intermediate states of the final sweep as checkpoints since information computed during this sweep is required just for subsequent time steps within this sweep, but not for previous sweeps. If only a restricted amount of memory is available, it is convenient to measure its size in terms of the number of fat checkpoints that it can accommodate. Since fat checkpoints, i.e. checkpoints of dimension $\bar{d}$, have to be stored during the adjoint sweep, we can use available memory on the original sweep to store thin checkpoints, i.e. checkpoints of dimension d, to reduce the total number of evaluated original steps F_i. On the adjoint sweep we remove thin checkpoints sequentially and store fat checkpoints instead of them as soon as required memory is available, i.e. as soon as a sufficient number of thin checkpoints is removed. We denote by $S(\mathbf{d_{3(l)}}, C)$ any admissible nested reversal schedule that can be applied to a multiple sweep evolution $\mathcal{E}_{3(l)}$ with a dimension distribution $\mathbf{d_{3(l)}} = (d, \bar{d}, \bar{\bar{d}})$ and a given number C of fat checkpoints. More formally we use the following definition.

Definition 1 (Nested Reversal Schedule $S(\mathbf{d_{3(l)}}, C)$). *Consider an evolution $\mathcal{E}_{3(l)}$ traversing l time steps in three alternative sweeps. Let $C \in \mathbb{N}$ fat checkpoints be available each of which can accommodate one intermediate state vector of the dimension $\bar{d}$, i.e. one intermediate state $\bar{x}_i$, $0 \le i \le l$. Moreover, assume that checkpoints can be stored during original and adjoint sweeps, provided sufficient memory is available. Assume that c thin checkpoints can be stored in place a single fat one, i.e. $\bar{d} = c\, d$. Then a* **nested reversal schedule** *$S(\mathbf{d_{3(l)}}, C)$ initializes $j = 0$ and $\bar{j} = l$, and subsequently performs a sequence of following basic actions*

$A \equiv$ *Increment* j *by* 1
$\bar{A} \equiv$ *Decrement* $\bar{j}$ *by* 1 *if* $\bar{j} - j = 1$
$W_i \equiv$ *Copy state* j *to a thin checkpoint* $i \in \{0, 1, \ldots, (C-1)c\}$
$\bar{W}_i \equiv$ *Copy state* $\bar{j}$ *to a fat checkpoint* $i \in \{1, 2, \ldots, C\}$
$R_i \equiv$ *Reset state* j *to a thin checkpoint* $i \in \{0, 1, \ldots, (C-1)c\}$
$\bar{R}_i \equiv$ *Reset state* $\bar{j}$ *to a fat checkpoint* $i \in \{1, 2, \ldots, C\}$
$D \equiv$ *Decrement* l *by* 1 *if* $\bar{j} = 1$ *and* $\bar{j} - j = 1$

until l *has been reduced to 0.*

It has to be arranged that each nested reversal schedule begins with the action R_0, such that the original state x_0 is read from the thin checkpoint 0.

One example of a nested reversal schedule $S(\mathbf{d_{3(9)}}, 2)$ for an evolution $\mathcal{E}_{3(9)}$ with the corresponding dimension distributions $\mathbf{d_{3(9)}} = (1, 3, 1)$ is shown in Fig. 3. Two fat checkpoints are available, i.e. 2 intermediate states on the adjoint sweep can be kept in memory simultaneously. Each of these states is of dimension $\bar{d} = 3$. Moreover, 3 thin checkpoints of dimension $d = 1$ can be stored instead of a single fat one. Further, the original state x^0 is stored as an additional 0th thin checkpoint.

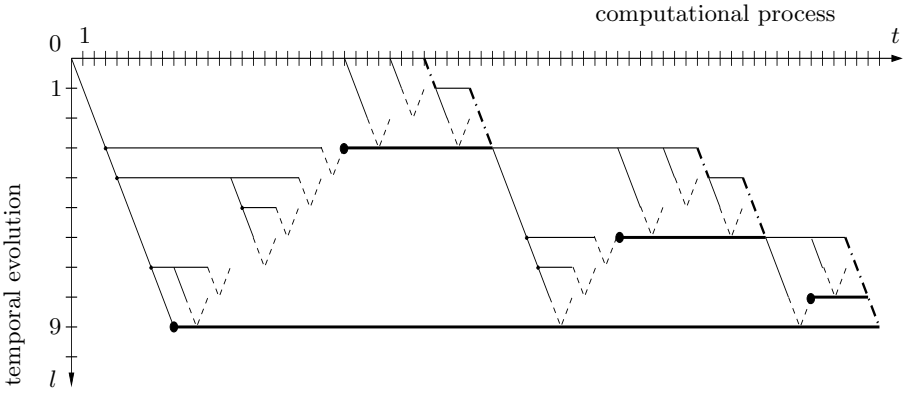


Fig. 3. Example of nested reversal schedule $S(\mathbf{d_{3(9)}}, 2)$.

In Fig. 3 physical steps are plotted along the vertical axis, and time required for the implementation of the evolution $\mathcal{E}_{3(9)}$ measured in number of executed steps is represented by the horizontal axis. Hence, the horizontal axis can be thought of as a computational axis. Each solid thin horizontal line including the horizontal axis itself represents a thin checkpoint, i.e. a checkpoint of dimension $d = 1$. Each solid thick horizontal line represents a fat checkpoint, i.e. a checkpoint of dimension $\bar{d} = 3$. Solid slanted thin lines represent original steps F_i, and adjoint steps $\bar{F}_i$ are visualized by dotted slanted lines. Final steps $\bar{\bar{F}}_i$ are drawn by slanted dashed-dotted thick lines.

One starts with the action R_0 restoring the original state x_0 from the 0th thin checkpoint. Three actions A are executed by performing three original steps F_1, F_2, and F_3 consecutively. The state x_3 is stored into the first thin checkpoint by the action W_1. Now again the action A is applied to perform one original step F_4, and

the state x_4 is stored into the second thin checkpoint by the action W_2. Further another three original steps are executed by the three actions A, and the state x_7 is stored in the third thin checkpoint. Then two further original steps are evaluated by the two actions A. Finally the state $\bar{x}_9$ is initialized and is stored in the first fat checkpoint by the action $\bar{W}_1$. The adjoint sweep is started by this action. Further, the state x_7 is restored from the third thin checkpoint by the action R_3, the state x_8 is reevaluated by the action A, and the states $\bar{x}_8$ and $\bar{x}_7$ are evaluated by the application of the action $\bar{A}$ twice. In this manner we come to the state $\bar{x}_3$, which is stored in the second fat checkpoint. On the way backward all thin checkpoints are removed, all fat checkpoints are occupied consecutively, and we have no more memory available to store fat or even thin checkpoints. Then one goes back to the adjoint state $\bar{x}_1$ by reevaluating required intermediate states. Consequently the first final step $\bar{\bar{F}}_1$ is performed. Further, one stores the current original state x_1 in the 0th thin checkpoint and continues in the same manner to execute all other final steps $\bar{\bar{F}}_2, ..., \bar{\bar{F}}_9$.

Using the nested reversal schedule $S(\mathbf{d_{3(9)}}, 2)$ from Fig. 3 one needs to perform 28 original steps F_i, 17 adjoint steps $\bar{F}$, and 9 final steps $\bar{\bar{F}}_i$.

Definition 2 (Repetition Numbers). *Consider a nested reversal schedule $S(\mathbf{d_{3(l)}}, C)$. The* **repetition numbers** *$r_i \equiv r(i)$, $\bar{r}_i \equiv \bar{r}(i)$, and $\bar{\bar{r}}_i \equiv \bar{\bar{r}}(i)$, defined as functions*

$$r, \bar{r}, \bar{\bar{r}} : [1, l] \to \mathbb{N} \ ,$$

count how often the ith original, the ith adjoint, and the ith final step are evaluated during the execution of the nested reversal schedule $S(\mathbf{d_{3(l)}}, C)$.

Provided a schedule is admissible in the sense that given $\mathbf{d_{3(l)}}$, C, and the initial l, it successfully reduces l to 0, its total runtime complexity can be computed from the additional problem parameters $\mathbf{t_{3(l)}} = (t, \bar{t}, \bar{\bar{t}})$. The temporal complexity of a nested reversal schedule $S(\mathbf{d_{3(l)}}, C)$, i.e. the run-time effort required to execute this nested reversal schedule can be computed as

$$\mathbf{T}(S(\mathbf{d_{3(l)}}, C), \mathbf{t_{3(l)}}) = t \sum_{i=1}^{l} r_i + \bar{t} \sum_{i=1}^{l} \bar{r}_i + \bar{\bar{t}} \sum_{i=1}^{l} \bar{\bar{r}}_i \ . \tag{11}$$

The optimal nested reversal schedule from the set of all admissible nested reversal schedules is required to minimize the evaluation cost, i.e., to achieve

$$\begin{aligned} &\mathbf{T}_{min}(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C) \ \equiv \\ &\qquad \min \left\{ \mathbf{T}(S(\mathbf{d_{3(l)}}, C), \mathbf{t_{3(l)}}),\ S(\mathbf{d_{3(l)}}, C) \text{ is admissible} \right\} \ . \end{aligned}$$

The set of optimal nested reversal schedules is denoted by $S_{min}(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C)$, so

$$\mathbf{T}(S_{min}(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C)) \equiv \mathbf{T}_{min}(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C) \ .$$

Now we face the task of constructing an appropriate optimal nested reversal schedule $S_{min}(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C)$. By brute force an optimal nested reversal schedule can be constructed using an exhaustive search algorithm (for more details see [488]). Using this approach one examines all possible distributions for thin and fat checkpoints and chooses the most efficient one. Clearly, such an exhaustive search is very expensive. In contrast to the situations for simple reversals involving only an original and an adjoint sweep, we have not been able to find a closed form characterization of optimal reversal schedules. Therefore, we have developed a heuristic for the construction of appropriate nested reversal schedules.

3.3 Heuristic

The intent of this heuristic is to restrict slightly the placements of thin checkpoints. Due the assumption (10), it is more convenient to reduce the freedom of movement of thin checkpoints, since even a considerable increment of the number of evaluated original steps does not cause a significant increase in the resulting evaluation cost wrt. the minimal evaluation cost $\mathbf{T}_{min}(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C)$ (accordingly (10) and (11)). Since one fat checkpoint can be stored as soon as the required memory is available, we store thin checkpoints such that after the removal of a sufficient number of thin checkpoints (c thin checkpoints), a corresponding fat checkpoint has to be stored at the same moment. Therefore, a nested reversal schedule can be decomposed into two nested subschedules and one simple reversal schedule as shown in Fig. 4.

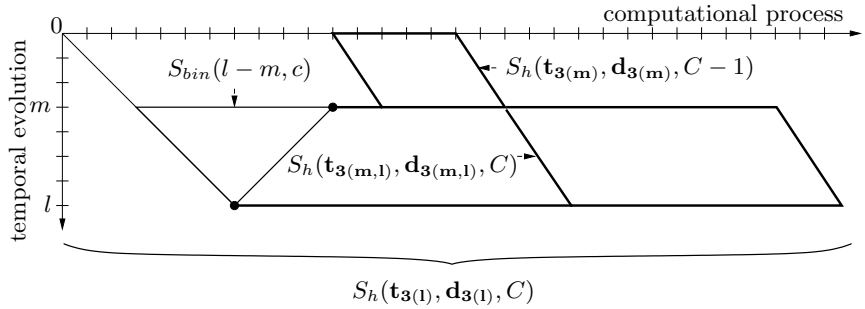


Fig. 4. Decomposition of a nested reversal schedule $S_h(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C)$.

Here, $S_h(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C)$ denotes a corresponding nested reversal schedule, constructed using the heuristic described above. This schedule is decomposed into two parts $S_h(\mathbf{t_{3(m)}}, \mathbf{d_{3(m)}}, C-1)$ and $S_h(\mathbf{t_{3(m,l)}}, \mathbf{d_{3(m,l)}}, C)$ as shown in Fig. 4 by storing the second fat checkpoint. An adjoint state stored in the second fat checkpoint corresponds to m. $S_{bin}(l-m, c)$ denotes the binomial reversal schedule with up to c checkpoints, applied for the reversal of $(l-m)$ original steps using minimal run-time and memory requirement (for details see e.g., [488]). Then, an appropriate nested reversal schedule $S_h(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C)$ can be constructed recursively by minimizing the evaluation cost

$$T_h(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C) = \min_m \{T_h(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C, m)\} = \min_m \{T_h(\mathbf{t_{3(m)}}, \mathbf{d_{3(m)}}, C-1) + T_h(\mathbf{t_{3(m,l)}}, \mathbf{d_{3(m,l)}}, C) + t_{add}(\mathbf{t_{3(m,l)}}, \mathbf{d_{3(m,l)}}, c)\}\,, \quad (12)$$

where $t_{add}(\mathbf{t_{3(m,l)}}, \mathbf{d_{3(m,l)}}, c)$ denotes an additional run-time effort required for placing $\bar{\mathbf{x}}_m$ in the second fat checkpoint. From (12) it is clear that the evaluation cost $T_h(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C)$ and consequently a corresponding nested reversal schedule $S_h(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C)$ can be evaluated using dynamic programming.

Instead of using dynamic programming we have developed a **Local-Descent Method**, which allows us to construct $S_h(\mathbf{t_{3(l)}}, \mathbf{d_{3(l)}}, C)$ using a linear run-time and memory requirement with respect to a number l of time steps and a number C of fat checkpoints (for details, see [488]).

4 Numerical Example

We consider a control problem that describes the laser surface hardening of steel (see [267]). The mode of operation of this process is depicted in Fig. 5. A laser

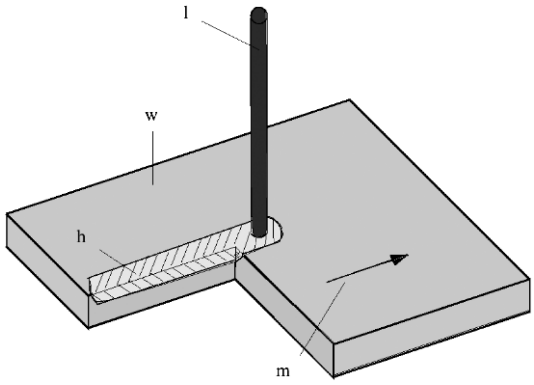


Fig. 5. Sketch of a laser hardening process.

beam moves along the surface of a workpiece, creating a heated zone around its trace. The heating process is accompanied by a phase transition, in which the high temperature phase in steel, called austenite, is produced. Since one usually tries to keep the moving velocity of the laser beam constant, the most important control parameter is the laser energy. Whenever the temperature in the heated zone exceeds the melting temperature of steel, the work-piece quality is destroyed. Therefore, the goal of surface hardening is to achieve a desired hardening zone, in our case described by a desired phase distribution a_d of austenite inside the workpiece Ω, but to avoid a melting of the surface. Hence we consider an optimal control problem with the cost functional $J(u)$ defined as

$$J(u) = \frac{\beta_1}{2} \int_\Omega (a(x,T) - a_d(x))^2 \, dx + \frac{\beta_2}{2} \int_0^T \int_\Omega [\theta - \theta_m]_+^2 \, dx \, dt + \frac{\beta_3}{2} \int_0^T u^2 \, dt \ , \quad (13)$$

where u is the laser energy and β_i $i = 1, 2, 3$ are positive constants. The second term in (13) penalizes temperatures above the melting temperature θ_m.

Let $\Omega := [0,5] \times [-1,0]$ with Lipschitz boundary $Q = \Omega \times (0,T)$, $\Sigma = \partial\Omega \times (0,T)$, $T = 5.25$. The system of state equations (14) consists of a semi-linear heat equation coupled with the initial-value problem for the phase transitions. a is the volume fraction of austenite, θ the temperature, τ a time constant and $[x]_+ = \max\{x, 0\}$ the positive part function. The equilibrium volume fraction a_{eq} is such that the austenite volume fraction increases during heating until it reaches some value $a < 1$. During cooling we have $a_t = 0$, and the value a is kept. The homogeneous Neumann conditions were assumed on the boundary. The term $-\rho L a_t$ describes the consumption of latent heat due to the phase transition. The term $u(t)\alpha(x,t)$ is the volumetric heat source due to laser radiation, where the laser energy $u(t)$ will serve as a control parameter. The density ρ, the heat capacity c_p, the heat conductivity k, and the latent heat L are assumed to be positive constants.

$$
\begin{aligned}
a_t &= \frac{1}{\tau(\theta)}[a_{eq}(\theta) - a]_+, && \text{in } Q\ , \\
a(0) &= 0, && \text{in } \Omega\ , \\
\rho c_\rho \theta_t - k\triangle\theta &= -\rho L a_t + u\alpha, && \text{in } Q\ , \\
\frac{\partial\theta}{\partial\nu} &= 0, && \text{on } \Sigma\ , \\
\theta(0) &= \theta_0, && \text{in } \Omega\ .
\end{aligned}
\tag{14}
$$

We study the following state and control constrained optimal control problem for the cost functional $J(u)$ as defined in (13):

$$
\min J(u),\ \text{s.t. } (\theta, a, u) \text{ solves (14) and } u \in U_{ad}, \tag{15}
$$

where $U_{ad} = \{u \in L^2(0,T) : \|u\|_{L^2(0,T)} \leq 2800\}$ is the closed, bounded, and convex set of admissible controls. The numerical implementation is obtained by a semi-implicit FE Galerkin scheme. The FE triangulation of Ω is done by a nonuniform mesh. The optimal control problem (15) is solved using the Quasilinearization scheme from Sect. 2 (see Algorithm 1). Nested reversal schedules are used for the reduction of memory requirements during the implementation of the Algorithm 1 applied to the optimal control problem (15).

5 Conclusion and Outlook

The iterative solution of optimal control problems in ODEs by various methods leads to a succession of triple sweeps through the discretized time interval. The second (adjoint) sweep relies on information from the first (original) sweep, and the third (final) sweep depends on both of them. This flow of information is depicted in Fig. 1. Typically the steps on the adjoint sweep involve more operations and require more storage than the other two. To avoid storing full traces of the original and adjoint sweeps we consider nested reversal schedules that require only the storage of selected original and adjoint intermediate states called thin and fat checkpoints. The schedules are designed to minimize the overall execution time given a certain total amount of storage for the checkpoints. While we have not found a closed form solution for this discrete optimization problem we have developed a cheap heuristic for constructing nested reversals that are quite close to optimality. Here we demonstrated that the dependence on l can be arranged polylogarithmically [224] by nested checkpoint strategies. Consequently, the operations count also grows as a second power of $\log_C l$, which needs not result in an increase of the actual run time due to memory effects.

We are currently applying the proposed scheduling schemes to laser hardening of steel [267] and other practical optimal control problems. As has been done in case of simple reversal schedules that involve only an original and an adjoint sweep our results should be extended to scenarios with nonuniform step costs and parallel computing systems.

Improving the Performance of the Vertex Elimination Algorithm for Derivative Calculation*

M. Tadjouddine[1], F. Bodman[2], J. D. Pryce[2], and S. A. Forth[1]

[1] Applied Mathematics & Operational Research, ESD, Cranfield University (RMCS Shrivenham), UK
{M.Tadjouddine,S.A.Forth}@cranfield.ac.uk
[2] Department of Information Systems, Cranfield University (RMCS Shrivenham), UK
j.d.pryce@cranfield.ac.uk

Summary. In previous work [TOMS, 2004, 30(3), 266–299], we used Markowitz-like heuristics to find elimination sequences that minimise the number of floating-point operations (flops) for vertex elimination Jacobian code. We also used the depth-first traversal algorithm to reorder the statements of the Jacobian code with the aim of reducing the number of memory accesses. In this work, we study the effects of reducing flops or memory accesses within the vertex elimination algorithm for Jacobian calculation. On RISC processors, we observed that for data residing in registers, the number of flops gives a good estimate of the execution time, while for out-of-register data, the execution time is dominated by the time for memory access operations. We also present a statement reordering scheme based on a greedy list scheduling algorithm using ranking functions. This statement reordering will enable us to trade off the exploitation of the instruction level parallelism of such processors with the reduction in memory accesses.

Key words: Vertex elimination, Jacobian accumulation, performance analysis, statement reordering, greedy list-scheduling algorithms, ELLIAD

1 Introduction

Many scientific applications require the first derivatives (at least) of a function $\mathbf{f} : \mathbf{x} \in \mathbb{R}^n \mapsto \mathbf{y} \in \mathbb{R}^m$ represented by computer code. This can be obtained using automatic differentiation (AD) [225,450]. We assume the function code has no loops or branches; alternatively, our work applies to *basic blocks* of more complicated code. From the program, we build the data dependence graph (DDG), or computational graph, of the function $\mathbf{f}$ as a Directed Acyclic Graph (dag) $G{=}(V, E)$, with vertex

* This work was partly supported by EPSRC under grant GR/R21882.

set V and edge set E. A vertex v_i represents a floating-point assignment of the original code; an edge $(v_j, v_i) \in E$ represents the data dependence relationship $v_j \prec v_i$ meaning v_j appears on the right hand side of the assignment that computes v_i. Logically, E and the relation $\prec$ are the same. Code may contain overwrites of variables; we assume these are removed by converting to Static Single Assignment form [142], so that a variable may be identified with the statement that computes it. We have $|V| = n + p + m = N$, where n, p, m are respectively the number of independent, intermediate and dependent vertices. We 'linearise' G by labelling its edges with local partial derivatives. Finally, we eliminate, in some order termed the elimination sequence, all intermediate vertices so that G is rendered bipartite. This process, the *vertex elimination approach*, can be found in [182, 225, 232, 394].

As shown in [182, 225], the linearised graph G can be viewed as an $N \times N$ sparse lower triangular matrix $\mathbf{C} = (c_{ij})$, and $\mathbf{C} - \mathbf{I}_N$ is called the *extended Jacobian*. The Jacobian $\mathbf{J}$ can be obtained by solving a sparse, triangular linear system with coefficient matrix $\mathbf{C} - \mathbf{I}_N$ using some form of Gaussian elimination. Since p, the number of intermediate vertices, tends to be large even in medium-sized applications, the performance of the vertex elimination algorithm can be degraded by fill-in. The floating point operations (flops) performed and the fill-in are determined by the elimination sequence. The question one would *ideally* like to answer is "Which elimination sequence gives the fastest code on a particular platform?" As a platform-independent approximation to this problem one may ask "Which elimination sequence minimises fill-in [respectively, flop-count]?" For a sparse symmetric positive definite system of linear equations, the fill-in problem is NP-complete [567], and we suspect that the same holds for our problem. Therefore, in practice a near-optimal sequence must be found by some heuristic algorithm. Our premiss is that such sequences allow us to generate faster Jacobian code.

Goedecker and Hoisie [216] report that performance of numerically intensive codes on many processors is a low percentage of nominal peak performance. There is a gap between CPU performance growth (around 55% per year) and memory performance growth (about 7% per year) [239]. To enhance performance, it would appear crucial to keep the memory traffic low. We study two aspects of the vertex elimination algorithm. First, we study how the number of floating point operations in the Jacobian code relates to its performance on various platforms. Second, we study how reordering the statements of the Jacobian code affects memory accesses and register usage. For these purposes, we generated Jacobian codes using Markowitz-like strategies and statement reordering and inspected the assembler from different processors and compilers. We studied how the execution time is affected by the number of flops and amount of memory traffic (loads and stores). We observed:

- A reordering of the Jacobian code's statements can improve its performance by a significant percentage when this reduces the memory traffic.
- For in-register data, the execution time is dominated by the number of floating point operations. A reduction of floating point operations gave further performance improvement.
- For out-of-register data, the execution time is dominated by the number of load and store operations. A reordering that reduced these memory access operations enhances Jacobian code performance.

Similar behaviour is found in performance analysis of other numerical codes, e.g., [216]. This paper presents the argument in the context of semantic augmentation of numerical codes as is carried out in AD of computer programs. We also

describe planned work to improve performance of Jacobian code, produced by vertex elimination, by reordering the statements using standard instruction scheduling algorithms.

2 Heuristics

Solving large linear systems by Gaussian elimination can be prohibitive due to the amount of fill-in. As said above, we use heuristic approximate solutions to the NP-complete problem of finding an elimination ordering to minimise fill-in. Over the past four decades several heuristics aimed at producing low-fill orderings have been investigated. These algorithms have the desired effects of reducing work as well. The most widely used are *nested dissection* [67, 203] and *minimum degree*. The latter, originating with the *Markowitz method* [353], is studied in [6].

Nested dissection, first proposed in [203], is a recursive algorithm which starts by finding a *balanced separator*, a set of vertices that when removed partition the graph into two or more components, each composed of vertices whose elimination does not create fill-in in any of the other components. Then the vertices of each component are ordered, followed by the vertices in the separator. Unlike nested dissection that examines the entire graph before reordering it, the minimum degree or Markowitz-like algorithms tend to perform local optimisations. At each elimination step, such a method selects a vertex with minimum cost or degree, eliminates it, and looks for the next vertex with the smallest cost in the new graph.

As described in [182, 501], we built the linearised computational graph in the following two ways:

1. Statement Level (SL) in which local derivatives are computed for each statement, no matter how complex its right-hand side.
2. Code List (CL) in which local derivatives are computed for each statement after the code has first been rewritten so that each statement performs a single unary or binary operation.

Then, we applied the following heuristics [182, 394] to the resulting graphs:

- Forward (F): where intermediate vertices are eliminated in forward order.
- Reverse (R): where intermediate vertices are eliminated in reverse order.
- Markowitz (M): at each elimination stage a vertex v_j of smallest Markowitz cost is eliminated. This cost is defined as the product of the number of predecessors of v_j times the number of its successors in the current, partly eliminated, graph.
- VLR (V): as Markowitz but using the VLR cost function defined by

$$\mathrm{VLR}(v_j) = \mathrm{mark}(v_j) - \mathrm{bias}(v_j),$$

  with $\mathrm{bias}(v_j)$ a fixed value for v_j, the product of the number of independent vertices and the number of dependent vertices to which v_j is connected.
- Any of the above with Pre-elimination (P): vertices with single successor are eliminated first. Then one of Forward, Reverse, Markowitz or VLR order is applied to those remaining.

We also used a Depth-First Traversal (DFT) algorithm [501] to reorder statements of the obtained Jacobian code, without altering dependencies between statements, in the hope of further performance improvement.

3 Performance Analysis

We consider two of the test problems reported in [182]: the Human Heart Dipole (HHD) from the Minpack 2 test suite [17] and the Roe flux calculation (ROE) [465]. These routines were differentiated using the AD tool ELIAD [182, 501] using the heuristics listed in Sect. 2. All the Jacobian codes were compiled on different platforms with maximum optimisation level and run for a number of times carefully calculated for each platform [182].

To assess the performance of the ELIAD-generated Jacobians, we studied the assembler from different platforms, counting the number of loads, stores and flops ('L', 'S' and 'Flops' in the tables) after compiler's optimisations.

Table 1 shows the results of our study from the SUN Ultra 10 processor with 440 MHz, 32 KB L1 cache, 2 MB L2 cache, and using the Workshop f90 6.0 Compiler. The observed time Obs-Time is the CPU time obtained by averaging a certain number of evaluations and runs, see [182] for details. Table 2 shows some run time predictions using a very simple model approximating the run time via the memory access count and the flops count. This approximate model estimates the following quantities: T_F, the time taken by the floating point operations

$$T_F = \frac{\text{Flops}}{\text{flops rate}} \times \text{cycle time} \times \text{latency}$$

Table 1. Performance data for the HHD and the Roe flux test cases on the Ultra10 platform, Obs-Time in μs.

	HHD			ROE		
Technique	Obs-Time	Flops	L+S	Obs-Time	Flops	L+S
SL-F	0.77	150	179	11.38	1732	2489
SL-R	0.79	148	188	7.26	1432	1600
CL-F	0.73	150	184	16.57	1843	3406
CL-R	0.80	148	201	6.98	1496	1655
SL-P-F	0.83	172	205	6.64	1580	1718
SL-P-R	0.73	172	182	6.11	1382	1626
CL-P-F	0.72	150	174	6.24	1580	1662
CL-P-R	0.71	148	182	5.81	1382	1609
SL-P-F-DFT	0.78	168	214	7.49	1584	1855
SL-P-R-DFT	0.66	164	180	5.73	1382	1466
CL-P-F-DFT	0.80	168	200	7.42	1587	1923
CL-P-R-DFT	0.66	164	167	5.84	1387	1305
SL-P-M	0.69	150	181	6.91	1524	1803
SL-P-M	0.83	168	214	5.71	1365	1507
CL-P-V	0.69	150	181	7.40	1524	1824
CL-P-V	0.83	168	214	6.17	1364	1503
SL-P-M-DFT	0.73	150	184	8.03	1529	1958
SL-P-V-DFT	0.80	168	200	6.19	1366	1375
CL-P-M-DFT	0.73	150	184	7.58	1532	1945
CL-P-V-DFT	0.80	168	200	5.59	1369	1362

and T_M, the time taken by memory access operations

$$T_M = \frac{(\mathrm{L} + \mathrm{S})}{\text{memory access rate}} \times \text{cycle time} \times \text{latency}.$$

The Ultra 10 processor can perform up to 2 flops per cycle (its flops rate is 2) with a latency of 3 cycles and 1 load or 1 store (its memory access rate is 1) with a latency of 2 cycles, and uses in-order execution [216] of instructions.

In Table 2, we represent the performance measures for a sample of methods shown in Table 1. The column 'Nom. flops' is the nominal flops count obtained from the source text. This table illustrates the following observations:

- A small reduction of flops count does not necessarily imply a reduction of the actual runtime Obs-time.
- T_M tends to be a better estimate of Obs-time than is T_F.
- The statement reordering improved performance when it reduced the number of memory accesses.

Table 2. A sample of methods applied to Roe flux on the Ultra 10 (timings in μs).

Technique	Nom. flops	Flops	L+S	T_F	T_M	Obs-time
SL-P-V-DFT	1462	1366	1375	4.65	6.26	6.19
CL-P-V-DFT	1578	1369	1362	4.68	6.20	5.59
CL-P-F	1742	1580	1662	5.40	7.56	6.24
CL-P-F-DFT	1742	1587	1923	5.40	8.74	7.42
SL-P-R	1505	1382	1626	4.71	7.40	6.11
SL-P-R-DFT	1505	1382	1466	4.71	6.66	5.73

These results have led us to believe that the runtime is more correlated with the memory accesses (loads and stores) than with the flops count. To further investigate this, we performed a linear regression analysis using the `regress` function of `MATLAB`'s statistics toolbox [356]. For both test cases in Table 1, we form the linear model:

$$T = aX + b + \epsilon\ ,$$

in which X represents the vector of flops or memory accesses (loads + stores), b a constant vector, ϵ a residual vector, and a a vector of parameters. Table 3 shows the 'explained variability' that is one of the statistics returned by `regress`, and the norm of the residual ϵ from the regression.

Table 3. MATLAB's `regress` results of the regression analysis of data of Table 1.

Model	variability	$\|\epsilon\|_2$
$T = a_1\mathrm{Flops} + b_1 + \epsilon_1$	0.87	8.7
$T = a_2(\mathrm{L+S}) + b_2 + \epsilon_2$	0.99	3.2

The flops explain about 87% of the variability in the observed time T, whereas the loads and stores explain about 99%. Furthermore, the (loads and stores) model has a smaller residual than the flops model. It is important to reduce the flops count

in numerical calculations, but it is even more crucial to minimise memory traffic. These experiments suggest consideration of code reordering techniques, data structures, and other optimisation techniques that reduce the amount of memory accesses if we aim to generate efficient derivative code even for medium-sized applications.

4 A Statement Reordering Scheme

In [182, 501], we used a Statement Reordering Algorithm (SRA) based on G', the DDG of the statements of the derivative code. By depth-first traversal, for each statement s, it tries to place the statements on which s depends close to s. It was hoped this would speed up the code by letting the compiler perform better register allocation since cache misses were shown not to be a problem in our test cases [501]. The benefits were inconsistent, probably because this does not account for the instruction level parallelism of modern cache-based machines and the latencies of certain instructions. In this work, we plan to encourage the compiler to exploit instruction level parallelism and use registers better by a SRA that gives priority to certain statements via a ranking function. The compiler's instruction scheduling and register allocation work on a dependency graph G'' whose vertices are machine code instructions. We have no knowledge of G'', so we work on the DDG G' on the premiss that our 'preliminary' optimisation will help the compiler generate optimised machine code. In the next sections, we shall use the instruction scheduling approach used for instance in [385] and the ranking function ideas of [251, 263, 334, 425] on a simple virtual processor.

4.1 The Processor Model

We consider the following simple model of a superscalar machine, similar to that of [35]. It has an unlimited number of floating-point (and other) 'registers'. It has one pipelined functional unit (FU) that can perform any scalar assignment-statement in our code, however complicated, in 2 clock cycles, including loading any number of operands from registers, computing, and storing the result in a register. An assignment-statement that does no processing (a simple copy) is assumed to take 1 cycle. A statement can be issued to the FU at each cycle; however data dependencies may 'stall' it: e.g. if the code `c=a*b; d=a+c` is begun at time $t = 0$, we cannot issue the second statement at $t = 1$ because `c` is not yet available.

We develop an algorithm that, within this model, tries to remove stalling by re-ordering code statements. Coarse-grained though the model is, we hope it imitates enough relevant behaviour of current superscalar architectures to produce re-orderings that give speedup in practice.

4.2 The Derivative Code and Its Evaluation

Since the original code was assumed branch- and loop-free, the same is true of the derivative code. It includes original statements v_i of f's code as well as statements to compute the local derivatives c_{ij} that are the nonzeros of the extended Jacobian $\mathbf{C}$ before elimination, but the bulk of it is *elimination statements*. As originally defined in [232], these take the basic forms $c_{ij} = c_{ik}c_{kj}$ or $c_{ij} = c_{ij} + c_{ik}c_{kj}$. Typically a c_{ij}

position is 'hit' (updated) more than once, needing non-trivial renaming of variables to put it into the needed single-assignment form.

Although not strictly necessary, we assume the vertex elimination process has been rewritten in 'one hit' form, e.g. by the inner product approach of [443]. That is, that each c_{ij} occurring in it is given its final value in a single statement of the form either $c_{ij} := c^0_{ij} + \sum_{k\in K} c_{ik}c_{kj}$ if updating an original elementary derivative, or $c_{ij} := \sum_{k\in K} c_{ik}c_{kj}$ if creating fill-in. Here K is a set of indices that depends on i, j and on the elimination order used, and c^0_{ij} stands for an expression that computes the elementary derivative, $\partial v_i/\partial v_j$. The result is that the derivative code is automatically in single-assignment form.

Its graph $G' = (V', \prec')$ — where V' is the set of statements and $\prec'$ the data dependence relationship — is a dag. A *schedule* π for G' assigns to each statement (denoted s in this section) a start-time in clock-cycle units, respecting data dependencies subject to the constraints of our processor model. It is a one-to-one (as the FU only does one statement at a time) mapping of V' to the integers $\{0, 1, 2, \cdots\}$. Write $t(s)$ for the execution time of s (1 or 2 in our model). Then to respect dependencies it is necessary and sufficient that

$$s_1 \prec s_2 \;\Rightarrow\; \pi(s_1) + t(s_1) \le \pi(s_2)\ . \tag{1}$$

for s_1 and s_2 in V'. The *completion time* of π is

$$T(\pi) = \max_{s\in V'}\{\pi(s) + t(s)\}\ .$$

Our aim is to find π to minimise the quantity $T(\pi)$ subject to (1). This optimisation problem is NP-hard [385, 387].

4.3 The DDG of the Derivative Code

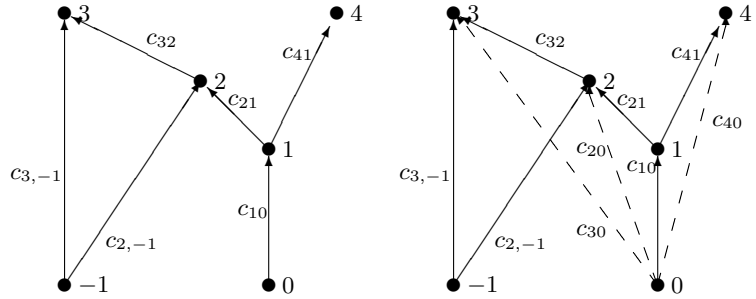
The classical way of constructing the derivative code's DDG is to parse the code, build an abstract representation, and deduce the dependences between all statements, see for instance [102, 387]. Since derivative code is generated from function code, its DDG can be constructed more easily by using data that is available during code generation. We omit details here.

$$\begin{aligned} v_{-1} &= x_1 \\ v_0 &= x_2 \\ v_1 &= \sin(v_0) \\ v_2 &= v_1 - v_{-1} \\ v_3 &= v_{-1}v_2 \\ v_4 &= \sqrt{v_1} \end{aligned}$$

Fig. 1. A code fragment.

Consider the code fragment of Fig. 1. Its computational graph is represented by the dag G on the left of Fig. 2, with, on the right, the extra edges produced on eliminating the intermediates v_1 and v_2 in that order.

The left of Fig. 3 shows derivative code from Fig. 1, with a somewhat arbitrary order of the statements respecting dependencies. Note one statement, #13, that combines computation of an elementary derivative with elimination. On the right is its DDG, which can be constructed directly from the elimination order and the original graph on the left of Fig. 2. (It could be made smaller by propagating the constant values $c_{2,-1}$ and c_{21}.) Edges from statement s are labelled by the value $(t(s) - 1)$, i.e., 0 or 1 representing the time delay imposed by s, in our processor model.



(a) Original computational graph (b) Augmented computational graph

Fig. 2. Graph augmentation process: eliminated vertices are kept, and fill-in is represented by dashed arrows.

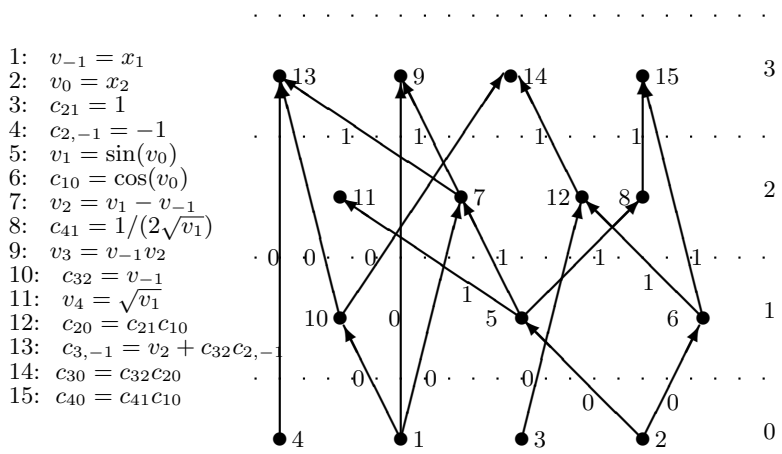


Fig. 3. The data dependence graph G' of the derivative code from the original dag G on the left of Fig. 2.

The depth-first traversal approach of [182, 501] gave the following schedule:

π_1 :

2	5		8	6		15	3	12	1	10	14	7	4	13	11	9	

π_1 contains 2 idle cycles and takes 18 cycles to complete including a cycle at the end: $T(\pi_1) = 18$. In Sect. 5, we produce a schedule that takes 16 cycles, the minimum possible. This new schedule combines the depth-first traversal property and the instruction level parallelism of pipelined processors via a ranking function.

5 A Greedy List Scheduling Algorithm

We use a greedy list scheduling algorithm as investigated in [35, 425]. We first preprocess the DDG G' to compute a ranking function that defines the relative priority of each vertex. Then a modification of the labelling algorithm of [35, 125] is used to iteratively schedule the vertices of G'.

Our ranking function uses the assumption that operations with more successors and which are located in a longer path should be given priority, being likely to execute with a minimum delay and to affect more operations in the rest of the schedule. We use the functions height(v) and depth(v) defined to be the length of the longest path from v to an input (minimal) vertex and to an output (maximal) vertex respectively. height(v) is defined by

1. for each input (minimal) vertex v, height$(v) = 0$;
2. for each other $v \in V'$, height$(v) = 1 + \max\{\text{height}(w), \text{ for all } w \prec v\}$.

depth(v) is defined in the obvious dual way. For a vertex $v \in V'$ we define the ranking function by

$$\text{rank}(v) = a * \text{depth}(v) + b * \text{succ}(v) \ , \tag{2}$$

where succ(v) is the number of successors of v and a and b are weights chosen on the basis of experiment. For $b = 0$, we recover the SRA using depth-first traversal as in [182, 501]. By combining the values depth(v) and succ(v), we aim to trade off between exploiting instruction level parallelism of modern processors and minimising register pressure. The preprocessing phase of our algorithm is as follows.

1. Compute the heights and depths of the vertices of G'.
2. Compute the ranks of the vertices as in (2).

The iterative phase of the algorithm schedules the vertices of G' in decreasing order of rank. It constructs the mapping π defined in Sect. 4.2 by combining the rank of a vertex and its *readiness* using the following rule:

Rule 1
A vertex v is ready *to be scheduled if it has no predecessor or if all its predecessors have already completed.*

This ensures predecessor data of the vertex v is available when v is scheduled. Ties between vertices are broken using the following rule:

Rule 2
Among vertices of the same rank choose those with the minimum height.
Among those of the same height, pick the first.

The core of the scheduling procedure is as follows:

1. Schedule first an input vertex v with the highest rank (break ties using Rule 2). That is, set time $\tau = 0$ and $\pi(v) = \tau$.
2. For $\tau > 0$, let v be the last vertex that was scheduled at times $< \tau$.
   a) Extract from the set of so far unscheduled vertices, the set A as follows:

$$\begin{aligned} &S(v) = \{w : w \succ v\} \\ &\text{If } \ S(v) \ \text{ is nonempty, set} \\ &B(v) = \{u : \text{height}(u) < \max\{\text{height}(w) \text{ for } w \in S(v)\}\}, \\ &\quad A = S(v) \cup B(v); \\ &\text{Otherwise} \\ &\quad A = \text{the set of remaining vertices.} \end{aligned}$$

   b) Extract from A the set of vertices R that are ready to be scheduled.

c) If R is empty, do nothing (a no-op at this cycle). Otherwise, choose from R a vertex v with maximum rank (break ties by Rule 2), and set $\pi(v) = \tau$.
d) Set $\tau = \tau + 1$.

3. Repeat step 2 until all vertices are scheduled.

We can easily check that this algorithm determines a schedule π that satisfies (1), thus preserving the data dependences between vertices of the graph.

Let us apply this algorithm to the DDG of Fig. 3 to get a schedule π_2. We first compute the height, depth and rank of each vertex using the coefficients $a = b = 1$:

vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
height	0	0	0	0	1	1	2	2	3	1	2	2	3	3	3
depth	2	3	2	1	2	2	1	1	0	1	0	1	0	0	0
succ	3	2	1	1	3	2	2	1	0	2	0	1	0	0	0
rank	5	5	3	2	5	4	3	2	0	3	0	2	0	0	0

To label the dag of Fig. 3, the algorithm starts with the input vertices and assigns $\pi_2(1) = 1$. Next it forms the set $A = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ of available statements and the set $R = \{2, 3, 4, 10\}$ of ready statements. Using the ranking list, it assigns $\pi_2(2) = 2$; etc. The result of this algorithm for the dag of Fig. 3 is the following optimal schedule without idle cycles:

π_2 :

1	2	5	6	10	3	4	7	12	8	11	13	9	14	15	

We observe that the completion time $T(\pi_2) = 16$, better than $T(\pi_1)$. The complexity of this labelling algorithm, which is similar to that of [35,125] for a dag with n vertices and e edges, was initially proved to be $\mathcal{O}(n^2)$ [35, 125] and can be implemented in $\mathcal{O}(n + e)$ as shown in [190].

6 Conclusions and Further Work

We have presented a detailed performance analysis of Jacobian calculations using the vertex elimination algorithm. We have shown that for even medium-sized numerical applications, the execution time is very much more correlated with the memory accesses than with the number of floating point operations. We pointed out that although the vertex elimination algorithm reduced the number of floating point operations, it should be coupled with instruction scheduling heuristics to enable exploitation of the superscalar nature of modern processors to maximise the performance of the derivative code. For that purpose, we described a statement reordering scheme based on a ranking function. We plan to implement it and test it using medium-sized problems on a range of superscalar processors. We may also look at ways of combining the two objectives of reducing flops and memory accesses in a single objective function.

Acknowledgements

The authors would like to thank Prof. J.K. Reid for enlightening discussions and one of the referees for a thorough reading of our paper and many useful comments and suggestions.

Flattening Basic Blocks*

Jean Utke

Mathematics and Computer Science Division, Argonne National Laboratory,
Argonne, IL, USA
utke@mcs.anl.gov

Summary. Preaccumulation of Jacobians of low-level code sections is beneficial in certain automatic differentiation application scenarios. Cross-country vertex, edge, or face elimination can produce highly efficient Jacobian preaccumulation code but requires the code section's computational graph to be known. Such graphs can easily be derived for straight-line codes with scalar variables. Practical codes in languages such as Fortran, C, and C++ with array indexing, pointers, and references introduce ambiguities. Constructing unambiguous computational graphs is necessary to realize the full potential of cross-country elimination. We give a practical solution and investigate theoretical options for graph construction.

Key words: Cross-country elimination, aliasing, DAG, computational graph, Jacobian preaccumulation, OpenAD

1 The Problem

The forward and reverse modes of automatic differentiation (AD) are the basic approaches that most AD tools implement. However, these modes are only two extremes of all elimination orderings possible on the computational graph that underlies a numerical code [42, 136, 225, 227]. Unlike forward and reverse modes, an elimination sequence with arbitrary order (cross-country elimination) requires the construction of the computational graph or an equivalent representation. A hybrid approach such as ADIFOR's [56] statement-level reverse mode combines forward and reverse modes, but true cross-country elimination has previously been implemented

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38, and by the National Science Foundation's Information Technology Research Program under Contract OCE-0205590, "Adjoint Compiler Technology & Standards" (ACTS).

and extensively used only in EliAD. This tool has been applied to practical problems [182, 500] showing benefits of Jacobian accumulation code by using elimination sequences not tied to only forward or reverse mode.

Considering elimination orderings only at the statement level allows us to exploit the *single expression use* property of right-hand-side expressions; that is, at this level we know the optimal solution for minimizing the operations count [396]. Extending the scope to basic blocks yields larger computational graphs. These graphs no longer have the single expression use property but contain more structural information than do the individual right-hand-side expressions. The obvious motivation is that elimination sequences on larger graphs have the potential for better solutions than do statement-level eliminations, which just become a special case.

EliAD has a number of restrictions on the input code it can handle. In particular, a general purpose AD tool targeting languages such as Fortran, C, or C++ has to contend with ambiguities in the computational graph introduced by accesses to array elements, pointers, formal subroutine arguments, and reference variables that cannot be statically resolved to single (virtual) addresses. This phenomenon is often referred to as *aliasing* and is ubiquitous. For dynamically created computational graphs, the aliasing problem turns into the question of how these graphs structurally depend on the code's inputs. Since repeatedly optimizing preaccumulations is commonly considered too expensive for dynamically created computational graphs, we aim at source transformation tools. A more recent approach requiring the construction of the computational graph involves scarcity-preserving eliminations [226, 234].

This paper does not focus on the circumstances under which cross-country or scarcity-preserving eliminations yield advantages or which particular code generation is the most efficient. Rather, we start by recognizing the need for practical construction of computational graphs in the presence of aliasing, and we introduce an approach to solving the ambiguity problem with the help of established compiler analyses. We view this as a prerequisite to the application of the aforementioned elimination techniques in general purpose AD tools targeting codes that exhibit the aliasing problem.

The construction and use of computational graphs particularly for basic blocks also play a role in optimizing compiler technology. There are similarities in the approach [3] for constructing the graphs, but ambiguities may be treated differently. Their typical use for tasks such as register allocation and object code generation indicates a relatively advanced level in the compilation process that is far removed from the high-level programming language used for the source-to-source transformation in our context. Consequently, we do not rely on a compiler environment to provide the computational graphs that allows us to suit the approach specifically for the purposes of AD.

The remainder of this section gives some context, notational framework and an introduction to the problem. In Sect. 2 we describe the compiler-based analyses used in the graph construction algorithm given in Sect. 4. Sections 5 and 7 investigate the problem in a more general context.

Consider a code that implements some numerical function

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$$

in the context of AD. We assume $\boldsymbol{f}$ can be represented by a computational graph $G = (V, E)$ that is a directed, acyclic graph (DAG). The set of vertices $V = X \cup Z \cup Y$ consists of vertices for the n independents X, vertices for the m dependents Y, and

vertices for p intermediate values Z occurring in the computation of $\boldsymbol{f}$. The edges E represent the direct dependencies of the $w \in Z \cup Y$ computed with elemental functions $w = \phi(\ldots, v_i, \ldots)$ on the arguments $v_i \in X \cup Z$. The computations imply a dependency relation $v_i \prec w$ and its transitive closure $\prec^*$. The ϕ are the elemental functions (sin, cos, etc.) and operators (+, −, *, etc.) built into the given programming language. All edges $(v, w) \in E$ are labeled with the local partial derivatives $\frac{\partial w}{\partial v}$; see [225] for details. The graph G is the basis for numerous investigations into strategies for the efficient computation of derivative information, such as the Jacobian

$$\boldsymbol{J}(\boldsymbol{x}) = \left[\frac{\partial y_i}{\partial x_j}\right] \in \mathbb{R}^{m \times n}, i = 1, \ldots, m, j = 1, \ldots, n$$

and Jacobian-vector products $\boldsymbol{J}\dot{\boldsymbol{x}}, \boldsymbol{J}^T\bar{\boldsymbol{y}}$ [232]. Most practical applications do not require the complete $\boldsymbol{J}$ for $\boldsymbol{f}$. Instead, Jacobians $\boldsymbol{J}_k$ of subsections k of the code for $\boldsymbol{f}$ may be *preaccumulated*, for instance for use in a subsequent reverse sweep. Such $\boldsymbol{J}_k$ contain the partials of subsection outputs $\boldsymbol{y}_k$ with respect to their inputs $\boldsymbol{x}_k$. In code-specific terms a preaccumulation of such sub-Jacobians is beneficial if the preaccumulated Jacobian has fewer nonzero entries than intermediate values or partials that would need to be stored for a subsequent reverse sweep over the sections and if computing $\boldsymbol{J}_k$ and (sparse) $\bar{\boldsymbol{x}}_k = \boldsymbol{J}_k^T\bar{\boldsymbol{y}}_k$ is cheaper than back propagation through the elemental ϕ.

For the construction of G let us start with a simple example. When we look at a sequence of expressions of scalar variables, it appears intuitive how to concatenate expression graphs representing right-hand sides of assignments; we call this process *flattening* into a graph G. Consider the example in Fig. 1. We are using C-style

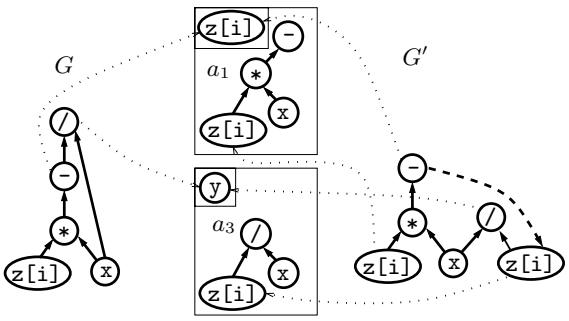


Fig. 1. Flattening of assignments a_1 : `z[i]=-(z[i]*x)`, and a_3 : `y=z[i]/x` into an unambiguous graph (left) and an ambiguous graph (right).

pseudo-code for all examples in this paper. Assume for the moment that there is no code between a_1 and a_3 and `i` is some fixed integer. We start by copying the right-hand-side expression graph of assignment a_1 into the graph G on the left side and note the fact that `z[i]` is the left-hand side by associating the maximal vertex $\ominus$ with `z[i]`. These associations are shown as the thin dotted arrows. We also remember which vertex represents the argument `x`, for instance by labeling it as in Fig. 1 or maintaining a list of vertex pairs. Even though `z[i]` was also an argument, it was overwritten and therefore is associated only with the maximal vertex $\ominus$.

Next we look at assignment a_3 and notice the use of `z[i]` in the right-hand side. This use can be identified with the preceding left-hand side; that is, the `z[i]` vertex in a_3 is identified with $\ominus$ already present in G. Furthermore, `x` appears as an argument again, which we identify with the vertex in G already associated with `x`. Now that all arguments of a_3 have been identified in G, we can simply copy the remaining vertex $\oslash$ (for the division intrinsic) and the attached two edges to G. The vertex $\oslash$ becomes the new maximal vertex and is associated with `y`, the left-hand side of a_3.

The process just described relies on the identification of references to the values of `x`, `y`, and `z[i]`. Naively done, the identification amounts to a match of symbol and scope; but in a general setting we have to ensure that two references to values are identified if and only if they use the same address in memory. Now assume that between a_1 and a_3 is some integer assignment `i=i+p`, where `p` is some parameter known only at run time. In a source transformation context a graph building algorithm cannot determine what address `z[i]` in a_3 is pointing to. As shown in the right side (graph G') of Fig. 1, the right-hand-side to left-hand-side identification is no longer unique, indicated by the dashed edge. That is, if `p` is 0, then we would have the graph on the left side, otherwise the graph on the right side (without the dashed edge). Obviously, only one of the two graphs can be the basis for an elimination sequence that produces the correct Jacobian preaccumulation code.

2 Variable Identification

In a compiler context *alias analysis* [387] is essential for determining which variables share addresses. The example discussed in Sect. 1 shows that syntactic equivalence is not sufficient for identification. Instead, one must determine whether `z[i]` points to the same address in both a_1 and a_3. The question of whether variables share an address, that is, whether they alias each other, arises from the use of pointers and the other language elements mentioned in Sect. 1. While the individual elements have distinct, language-specific characteristics, these differences are not relevant for this paper. For brevity we refer to all scalar variables, reference variables, pointers, formal arguments, and array dereferences from now on as *variables*.

Following [387], we distinguish *flow-sensitive* and *flow-insensitive* as well as *must* and *may* alias analysis; their respective results are given in a variety of formats. For this paper we simplify by assuming a vector of virtual address sets A to illustrate the use of may and must alias information. Practical implementations have more complicated result representations but answer the same aliasing tests. The use u_v of a variable v at a specific point in the code refers to a particular set A_{u_v}, through an index, which allows us to represent flow-sensitive analysis results. Flow-insensitive alias analysis yields only a single address set in the alias vector for every use of v in the entire program, regardless of the point of use in the code. If A_{u_v} contains exactly one address, it expresses must-alias information; that is, v must use that one given address. If A_{u_v} contains more than one address, it represents may-alias information; that is, v may use one of the given addresses[2]. For instance, whenever there is a dependence of address calculations on run-time parameters, the alias analysis will be unable to narrow A_{u_v} to a single address.

[2] but it has to use one of them

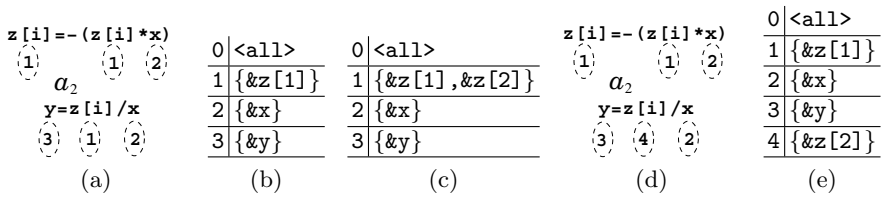


Fig. 2. Flow-insensitive alias vector: vector indices in the code (a), here with `i=1` and a_2 empty (b), or a_2 : `i=i+1` (c); flow-sensitive alias vector: vectors indices in the code (d), here for `i=1` and a_2 : `i=i+1` (e).

For our example from Fig. 1 we show some of variations of the results of alias analyses in Fig. 2. The indices into the alias vector are depicted in the dashed ovals. The conservative default lets all variables be aliased to everything, here by referring to a special entry `<all>` in the alias vector representing the entire address space. If in our example a_2 : `i=i+p` we have to assume at least $A_4 = \{\&\mathtt{z}[0], \ldots, \&\mathtt{z}[\mathtt{n}-1]\}$. Permitting out-of-bounds access, we would even have to default to the `<all>` entry.

To positively identify a use of variable w with a use of variable v, we therefore check

$$|A_{u_v}| = |A_{u_w}| = 1 \wedge A_{u_v} = A_{u_w} \quad ;$$

that is, the must alias test that u_w and u_v occupy the same single address.

3 Removing Ambiguity by Splitting

In the ambiguous version of the example in Sect. 1 we could not identify `z[i]` from the left-hand side of a_1 and its subsequent use in a_3, and we created a new vertex in G'. The creation of possibly extraneous vertices already applies to variables within a right-hand-side expression; see Fig. 3 (a,b) for an example expression `*q+*r+*s`. The

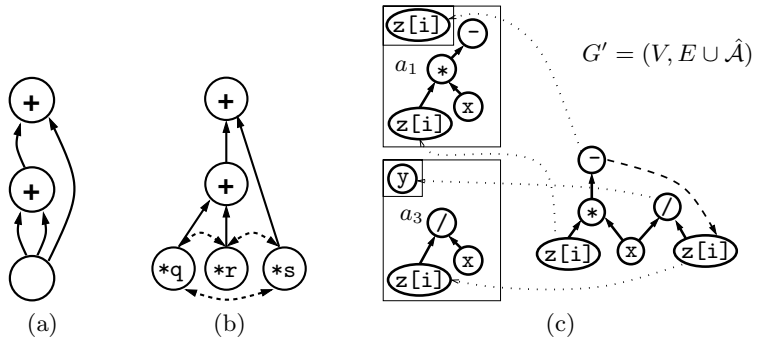


Fig. 3. Variable identification: intra right-hand side with (a): $|A_{u_{*\mathtt{q}}}| = 1$, $A_{u_{*\mathtt{q}}} = A_{u_{*\mathtt{r}}} = A_{u_{*\mathtt{s}}}$, (b): $|A_{u_{*[\mathtt{q},\mathtt{r},\mathtt{s}]}}| > 1, A_{u_t} \cap A_{u_{t'}} \neq \emptyset, t, t' \in \{*\mathtt{q}, *\mathtt{r}, *\mathtt{s}\}$, and (c): ambiguous G' as in Fig. 1 with $|A_{u_{\mathtt{z}[\mathtt{i}]}}| > 1$.

dashed edges indicate may-aliasing, which in the context of G can be expressed with additional edges $\mathcal{A} = \{(v, w) \in G : A_v \cap A_w \neq \emptyset\}$. Consider a code consisting of an ordered list of assignments $L = (a_1, \ldots, a_p)$. Unless all these assignments are mutually independent, the order is essential for the semantic. Flattening the assignments into a graph G replaces this order in the assignment list by the evaluation order imposed by the directed edges in G. If $\mathcal{A} = \emptyset$, then G has minimal vertex count and represents the dependency information exactly, thereby preserving the semantics of the code. $\mathcal{A} \neq \emptyset$ in Fig. 3 (b) is benign; there are merely more vertices in the graph than in (a). $\mathcal{A} \neq \emptyset$ in Fig. 3 (c) has ambiguous dependency information. To preserve semantics through correct dependency information, we therefore need to ensure that the subset $\hat{\mathcal{A}} = \{(v, w) \in \mathcal{A} : w \in rhs(a_i) \wedge v = lhs(a_k), k < i\} = \emptyset$. The graph $G' = (V, E \cup \hat{\mathcal{A}})$ as shown in Fig. 3(c) can be viewed as an ambiguous graph in which we leave open the question whether the additional edges $\hat{\mathcal{A}}$ exist. Alternatively, we can view $G' = (V, E \cup \hat{\mathcal{A}})$ as representing a set of unambiguous graphs in which each unambiguous graph has either one or none of the edges $(v, w) \in \hat{\mathcal{A}}$ for each vertex w that has such incoming edges in $\hat{\mathcal{A}}$.

A simple and practical approach is to iterate through L such that whenever there is an ambiguous reference of a right-hand-side variable in assignment a_i to a preceding left-hand side, we remove all parts of a_i that have been added to G and start flattening into a new graph. In other words we *split* G into an ordered sequence of graphs G_l that preserve the semantics as described in detail in Sect. 4. In Sect. 5 we look at the problem from a more general point of view.

4 Practical Solution

To simplify the formal description of the flattening algorithm, we assume the following canonicalizations of the input code.

C1: All computations $w = \phi(\ldots, v_i, \ldots)$ are elements of an *assignment* statement a of the form $v := e$ with a single variable $v = lhs(a)$ on the left-hand-side and a right-hand-side expression $e = rhs(a)$ that is side-effect free[3].

C2: All assignment statements are elements of an ordered list L of statements contained in a basic block; see also Sect. 6.

C3: Subroutine calls may have side effects. Any subroutine call implies a split of L into a preceding and succeeding list of assignments; see also Sect. 6.

C4: All functions and operators that are not intrinsic or have side effects are canonicalized into subroutine calls. Intrinsics have closed-form expressions for their partial derivatives, as is the case for all elementals ϕ.

C5: Intrinsics without arguments, for instance a function returning a constant value, are inlined.

Restricting our algorithm to flatten only consecutive sections of L at the granularity of whole assignments allows for simplifications over the general approach described in Sect. 5. In particular, we can use *du/ud-chains* [387], which are built by using alias information and present the dependency information in a suitably enhanced format. In short, a ud-chain (read use-define-chain) D_{u_v} contains for a

[3] This requires, for instance, the canonicalization of C++ increment operators.

particular use u_v of a variable v the locations of possible definitions, that is, assignments to v. Similarly, a du-chain (read define-use-chain) U_{u_v} contains for a particular definition, that is, v is a left-hand side, the locations of all possible references to the assigned value (uses). Because of canonicalizations C1, C2, and C4 we can simply equate these locations with the statements contained in basic blocks. Traditionally, ud-chains are introduced for the use of a variable following alternative definitions in separate branches in the control flow. If a ud-chain for v refers to exactly one statement, then this is the most recent assignment to v with respect to the control flow and $|A_{u_v}| = 1$. If $|A_{u_v}| > 1$, then the chain may refer to more than one statement even in the same basic block. In principle there is no limit to the locations du/ud-chains may refer to. Dereferencing a global pointer variable can entail chains referring to locations in the entire code. Limiting the flattening algorithm to basic blocks, we can reduce the needed information from ud/du-chains to statements within basic blocks and a placeholder "0" for defining locations outside its scope. For instance in the example in Fig. 1, the ud-chain for the use of `z[i]` in the right-hand side of a_3 is (a_1) in the unambiguous case and $(a_1,0)$ in the ambiguous case.

4.1 Graph-Splitting Algorithm

The following algorithm is a simplified version of the practical implementation mentioned in Sect. 7.

Algorithm [Semantic-Preserving Flattening] *Consider a sequence of assignments $L = (a_1, \ldots, a_l)$ to be flattened into an ordered sequence of directed acyclic graphs $G_i = (V_i, E_i)$. We maintain two tracking lists P_{var} (variables) and P_{intr} (intrinsics) of pairs (v_e, v_{G_i}) of vertices v_e from the expression $e = rhs(a)$ associated with vertices $v_{G_i} \in V_i$. Perform the following steps.*

init: $i := 0$
$k := k' := 1$
split: $i := i + 1; \quad k' := k$
$G_i := (V_i := \emptyset, E_i := \emptyset)$ // start with an new graph
$P_{var} := P_{intr} := S_{lhs} := \emptyset$ // and empty lists
loop: $e := rhs(a_k)$
$\forall v \in V_e$
if (v *is a variable*)
if $[(D_{u_v} = (a_j) \wedge j < k') \bigwedge$ //defined outside of G_i and
$(\nexists\, (w, .) \in P_{var} : D_{u_v} = D_{u_w})]$ //not already there
$\bigvee (D_{u_v} = (0)))$ //or defined outside of the scope
add new vertex v' *to* V_i; $P_{var} := P_{var} + (v, v')$
if $(|D_{u_v}| > 1 \wedge \exists a_j \in D_{u_v} : j \geq k')$ //ambiguous
remove all additions to G_i *done for* a_k
goto split:
elseif $(|\{(w, v) : (w, v) \in E_e\}| > 0)$ //must be an intrinsic
add new vertex v' *to* V_i; $P_{intr} := P_{intr} + (v, v')$
$\forall (v, w) \in E_e$
add new edge (v', w') *to* E_i *where*

$((v, v') \in P_{var} \cup P_{intr}) \vee ((t, v') \in P_{var} \wedge D_{u_v} = D_{u_t}) \bigwedge (w, w') \in P_{intr}$
if $(\exists (w, w') \in P_{var} : A_{u_w} \cap A_{u_v} \neq \emptyset$ *with* $v = lhs(a_k))$
  $P_{var} := P_{var} - (w, w')$ // keep the graph acyclic by
$P_{var} := P_{var} + (lhs(a_k), v'_{max})$ // only tracking the top node
$S_{lhs} := S_{lhs} \cup lhs(a_k)$
if $(k < |L|)$
  $k := k + 1$
  goto loop:
else
  done

In the algorithm the first statement in each G_i has index k'. This allows us to distinguish definitions of variables inside or outside the currently considered G_i. Reducing the ud-chain information to basic block scope has the drawback that variables with the same outside-of-scope definition cannot be identified. Hence, while preserving semantical correctness, we do not achieve the minimal vertex count.

The algorithm will copy only leaf nodes from expression graphs that are variables. Intrinsics cannot be represented by minimal variables because of canonicalization C5, and constants always imply a zero edge label and therefore are ignored for our purposes. To limit the formalism, we exclude special cases such as purely constant assignments.

The algorithm keeps the G_i acyclic, a requirement that becomes an issue if a variable in any of the assignments in L is used and then overwritten. Acyclicity is maintained by tracking only the "most recent" vertex in G representing an assignment to a given variable. This tracking is sufficient because of canonicalizations C1 and C4. In the unambiguous case of Fig. 1, we show the situation of `z[i]` being overwritten. While processing a_1, we first add the pair `(z[i],`w`)` to P_{var}, where w is the left minimal vertex in G. After looping through all vertices and edges of $rhs(a_1)$, we find the left-hand side `z[i]` exists in P_{var}. We remove `(z[i],`w`)` and add `(z,`$\ominus$`)` instead.

In the ambiguous case of Fig. 1, we start with $k = 1$, process a_1, and `x` and `z[i]` are tracked in P_{var}. The integer statement a_2 :`i=i+p` is passive with respect to the derivative computation and therefore can be skipped. Assume we continue with processing `x` in the right-hand side of a_3. If `x` is defined outside the current basic block and ud-chains are basic block scoped, then $D_{u_x} = (0)$, and we add a new vertex. Next is `z[i]`, for which $D_{u_{\texttt{z[i]}}} =$`(`a_1`,0)`; the algorithm determines ambiguity, removes the just-added vertex for `x`, and flattens a_3 into a new graph G_2.

In practice the AD tool then determines the cross-country eliminations in G_1 and G_2 separately, and produces the respective preaccumulation code in terms of the original variables `x`, `y` and `z[i]` first for G_1, followed by statement a_2 and then followed by the code for eliminating in G_2. By keeping the generated elimination code in the same order as the graphs G_i, one can easily see that the semantics is indeed preserved.

The collected S_{lhs} for each G_i can be used in conjunction with du-chains to narrow the proper set of dependent variables; see Sect. 5.1.

5 Splitting into Edge Subgraphs

Abandoning the convenient restriction of splitting exactly along assignment limits, we revisit $(V, E \cup \hat{\mathcal{A}})$ introduced as a set of unambiguous graphs in Sect. 3. We define an *edge subgraph* $G_s = (V_s, E_s)$ of a graph $G = (V, E)$ with $V_s \subseteq V$ and $E_s \subseteq E$ such that if $(v, w) \in E_s$, then $v, w \in V_s$, and if $(t, u), (v, w) \in E_s \wedge (u, v) \in E$, then $(u, v) \in E_s$. A *split* of G into edge subgraphs $G_i = (V_i, E_i)$ is defined such that $(E = \bigcup E_i) \wedge (E_i \cap E_j = \emptyset)$. A split duplicates vertices v for all pairs $(u, v), (v, w) \in E \wedge (u, v) \in E_i \wedge (v, w) \in E_j$ such that they occur in both graphs G_i and G_j. The identity between vertices $v = v_i$ in G_i and $v = v_j$ in G_j can be expressed with the set of (virtual) identity edges $\mathcal{I} = \{(v_i, v_j)\}$. In the graph $(\bigcup V_i, \bigcup E_i \cup \mathcal{I})$ we find all G_i and the pairs of duplicated vertices connected by the edges in $\mathcal{I}$. With these identities one may interpret the splitting into edge subgraphs as the inverse of the flattening.

Consider all edges $(v, w) \in \hat{\mathcal{A}}$ as possible identities like those in $\mathcal{I}$. In $(V, E \cup \hat{\mathcal{A}})$ for each such w only one (or no) element of $\hat{\mathcal{A}}$ is the actual identity; that is, $(v, w) \in \mathcal{I}$. Like $(\bigcup V_i, \bigcup E_i \cup \mathcal{I})$ we view $(V, E \cup \hat{\mathcal{A}})$ as consisting of edge subgraphs G_i that satisfy the following criterion.

Edge subgraph criterion: The G_i have $\hat{\mathcal{A}}_i = \emptyset$, that is, locally unambiguous dependency information, and can be (partially) ordered with "$\prec$" such that $\forall (v, w) \in \hat{\mathcal{A}} : v \in G_j$, then $w \in G_k, G_j \prec G_k$, and vice versa, $\forall (t, u) \in \hat{\mathcal{A}} : u \in G_j$, then $t \in G_i, G_i \prec G_j$.

In other words, all virtual in-edges are out-edges of preceding graphs, and all virtual out-edges are in-edges of succeeding graphs. Figure 4 shows $G' = (V, E \cup \hat{\mathcal{A}})$ from Fig. 1 with two edge subsets as dash-dot-encircled areas on the left. The split results in the two shaded, unambiguous subgraphs on the right; the connecting virtual edge imposes the order between the subgraphs. The ordering between the graphs

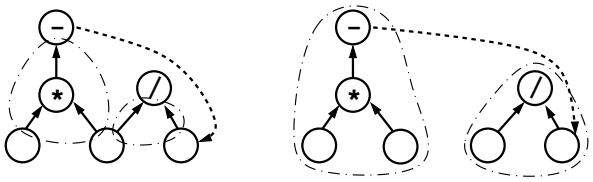


Fig. 4. $G' = (V, E \cup \hat{\mathcal{A}})$ for the example in Fig. 1 before (left) and after (right) split.

preserves the semantics. The edge subgraph criterion implies a minimal number of subgraphs, but it does not determine the subgraphs G_i uniquely. For example, consider $\hat{\mathcal{A}} = \{(v, w), (v', w)\}$ resulting in G_1 and G_2. Any edge (t, u) for which there are no paths $P_{u,v}, P_{u,v'}, P_{w,t}$ can be made part of either G_1 or G_2. More generally we can define the set of edges that are *movable* between edge subgraphs G_i and G_j as $\{(t, u) \in E : \forall (v, w) \in \hat{\mathcal{A}} : v \in V_i \wedge w \in V_j : \nexists P_{u,v}, P_{w,t}\}$. Consequently, one can formulate objectives to determine an optimal split that we will briefly investigate in Sect. 6. Reconsidering the algorithm in Sect. 4 in this more general context, we can prove the following

Proposition: The semantic-preserving flattening algorithm attains the minimal number of subgraphs satisfying the edge subgraph criterion.

Proof. A split into two subgraphs $G_1 = (v_1, E_1), G_2 = (v_2, E_2)$ with $v \in V_1, w \in V_2$ can cover all virtual edges (v', w') in a set S with $S = \{(v, w).(v', w') \in \hat{\mathcal{A}} : \nexists\, P_{v,w'}, P_{w,v'} \in (V, E \cup \hat{\mathcal{A}})\}$. If $\exists P_{w,v'}$, then v and w', and by edge subgraph definition all vertices and edges on the path would have to be in one subgraph. Each virtual edge belongs to such an S, and $(V, E \cup \hat{\mathcal{A}})$ defines s sets. Then the minimal number of subgraphs is $s+1$. The algorithm keeps flattening into the same subgraph as long as all encountered virtual edges (v, w) originate outside G_i in another G_j (this does not actually consider outside-of-scope references as edges). All targets w lie inside G_i. The algorithm creates a new subgraph whenever it encounters the target vertex of an edge $(v', w') \in \hat{\mathcal{A}}$ with a source with G_i. That means a $P_{w,v'}$ exists and a (v', w') belongs to a new equivalence set.

5.1 Determining Jacobian Entries

We now have a split into l unambiguous subgraphs $G_1, \ldots, G_l$. Before performing any elimination in G_i, we have to determine which vertices in G_i are independent and which are dependent, in order to obey the restrictions on the vertices and edges that can be targets of eliminations. Obviously, the set of independents is exactly the set of n_i minimal vertices $\{v : \nexists\, (u, v) \in E_i\}$. The similar assumption that all maximal vertices $\{u : \nexists\, (u, v) \in E_i\}$ constitute the dependent set is not necessarily true. One might simply have a variable v assigned that is then referred to in a right-hand-side expression flattened into G_i as well as one flattened into a successor G_j. If we knew $\mathcal{I} \subseteq \hat{\mathcal{A}}$ exactly, we could determine the m_i dependents as $\{v : (v, w) \in \mathcal{I}, v \in V_i, w \in V_j, j > i\} \cup \{v : \nexists\, (v, u) \in E_i\}$. Then the elimination in G_i yields $\boldsymbol{J}_{G_i} \in \mathbb{R}^{m_i \times n_i}$. If L in the algorithm in Sect. 4.1 contains the entire code subsection k of interest (see Sect. 1), then we can write

$$\boldsymbol{J}_k = \prod_{i=1}^{l} \left(\boldsymbol{P}_i^{(r)} \begin{bmatrix} \boldsymbol{J}_{G_i} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I}_i \end{bmatrix} \boldsymbol{P}_i^{(c)} \right)$$

with the identity $\boldsymbol{I}_i \in \mathbb{R}^{s_i \times s_i}, s_i = |\{(v, w) \in \mathcal{I}, v \in V_j, w \in V_{j'}\, j < i < j'\}|$. $\boldsymbol{P}_i^{(r)}, \boldsymbol{P}_i^{(c)}$ are permutation matrices that line up the rows and columns correctly. $\mathcal{I}$ is not known, a conservatively correct approach is to consider all maximal vertices and all vertices with virtual out-edges in $\hat{\mathcal{A}}$ dependent. This is obviously suboptimal, and we can use a better and practical solution with respect to the S_{lhs} from Sect. 4.1 that relies on du-chain information. At the point of flattening a split inducing assignment a_k, we determine the set of dependent variables for G_i according to

$$\{v \in S_{lhs} : \exists a_j \in U_{u_v} : a_j \in L,\ j > k \bigvee 0 \in U_{u_v}\}\ .$$

Because of ambiguity this is in general a superset of the exact set of dependent variables but is in any case a subset of S_{lhs}. Similarly, all dependent variables $\boldsymbol{y}_k$ of code subsection k having the same scope as the du-chain information, that is, the basic block, are determined by all variables v with $0 \in U_{u_v}$.

To perform a complete vertex elimination yielding a bipartite graph, a graph with a nonmaximal dependent vertex v needs to be augmented with vertex v' and

an edge (v, v') that has a unit edge label. Then v' takes the place of v in the dependent set, and a complete vertex elimination is possible. In the case of edge eliminations we have to make sure that no edge-front eliminations are performed on in-edges of such a v. On the other hand, there are no consequences for face elimination as the construction of the dual graph properly represents v. Figure 5(a) shows the unambiguous G from Fig. 1 where independent and dependent vertices are shown with $\triangle$ and $\triangledown$, respectively. We assume for the left-hand side of a_1 that $U_{u_{\mathtt{z[i]}}} = (a_3, 0)$ and therefore the left-hand side of a_1 is a nonmaximal dependent vertex. Following [399] G is augmented (dashed elements) to construct the dual graph shown in Fig. 5(b). A dashed edge is emanating from each dependent vertex.

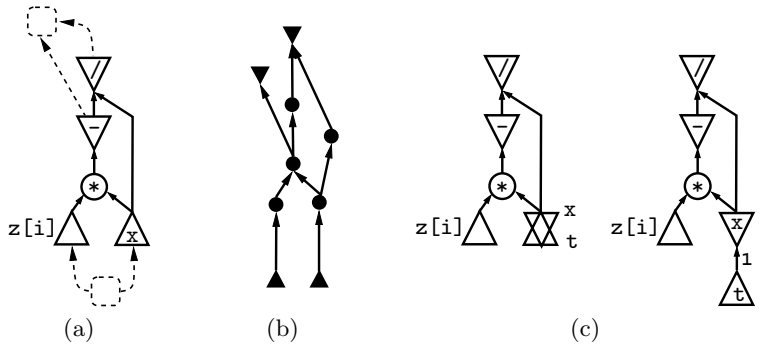


Fig. 5. Nonmaximal dependents v: (a) augmented G, (b) dual graph, (c) collapsed (left) and separated (right) independent/dependent variable.

Because the flattening algorithm identifies the left-hand-side variables with the respective maximal right-hand-side expression vertices, an assignment a_0 : `x=t`, for instance, with $0 \in U_{u_{\mathtt{x}}}$ can collapse an independent and a dependent variable into the same vertex. Generally this may lead to a single unconnected vertex or a situation depicted in the left graph in Fig. 5 (c). In both cases a split of the collapsed vertex into two vertices with a connecting edge solves the issue. The unit edge label is the corresponding Jacobian entry.

6 Outlook

In this section, we present some aspects and extensions to the graph construction that are the subject of ongoing and future work.

6.1 Scope of Flattening

We focused here on the basic block as the scope of interest for the flattening. For programs implementing a general $\boldsymbol{f}$, the control flow may depend on the values of the arguments in X. For instance, argument-dependent loop bounds do not permit the construction of a single G representing the code for $\boldsymbol{f}$ for all possible input

values. On the other hand, the construction of an argument-specific G, for example based on an execution trace, easily leads to huge computational graphs devoid of information indicating repeated structures. The actual goal of minimizing operations for the derivative computation sets practical limits for the size of G to which a minimization algorithm may be applied. Generally, the minimization is too costly to be reapplied to each argument-specific G. Therefore, we should require that G be structurally argument independent. Putting G within the scope of a basic block provides a simple criterion for structural independence.

There are at least two obvious ways to consider a computational graph in scopes larger than that of single basic blocks: 1) exploiting interface contraction and 2) inlining subroutine calls and sequentializing branches. The first approach is of clear practical value; the method and the benefits are described in [275]. Building a computational graph through the body of a subroutine that is being called from the code subsection in question amounts to inlining. In a source transformation context, inlining should be done explicitly by a compiler front-end prior to any AD code transformation. On the other hand, we can flatten "black box" subroutine calls if we can obtain all partial derivative values directly, for instance, if they are returned as a specific set of parameters. The flattening algorithm can treat such subroutine calls like intrinsics. In a ud-chain, simply referring to the subroutine call as the definition location for a variable is no longer sufficient because the subroutine may define multiple variables, and we then need additional information to find out which one.

The last extension, the sequentializing of branches, has limited practical value if the computational cost of executing all branches rather than one is too high. The source transformation algorithm will have to make the branches mutually independent, execute them sequentially, and select the proper result. With alias information and du/ud-chains we already have the prerequisites for such a transformation. However, it is beyond the scope of this paper.

6.2 Optimizing the Split

In theory, because of the edges that are movable between subgraphs G_i (see Sect. 5), we can have a variation of the number of independent and dependent vertices in the G_i, a variation in the sparsity of the J_{G_i}, and, as the G_i change, a variation of the minimal cost for preaccumulating the corresponding J_{G_i}. The first two obviously affect the storage requirements for a subsequent reverse sweep and the cost of the implied (sparse) Jacobian vector products. So far we have considered minimizing only the preaccumulation cost, which now would become the inner problem of a nested optimization. Currently, we are not able to tackle such a nested optimization and therefore stay with the simple split choice made for the semantic-preserving flattening algorithm.

Moreover, introducing splits in addition to the minimally necessary ones can yield improvements with respect to the above minimization criteria. A simple example is a sequence of $\boldsymbol{x} = \boldsymbol{a}\boldsymbol{b}^T\boldsymbol{x}; \boldsymbol{x} = \boldsymbol{a}\boldsymbol{b}^T\boldsymbol{x}; \ldots$ implemented by $z = \boldsymbol{b}^T\boldsymbol{x}; \boldsymbol{x} = \boldsymbol{a}z$ and assuming splits necessitated by an inability to identify $\boldsymbol{x}$ between the left- and right-hand sides. If we also split through z, then instead of storing n^2 Jacobian entries from $\boldsymbol{a}\boldsymbol{b}^T$, we can store $\boldsymbol{a}$ and $\boldsymbol{b}$ with $2n$ entries. Rather than introducing additional splits, this issue should be seen in the context of scarcity-preserving eliminations [226, 234].

7 Conclusions

The paper illustrates the problems arising with the construction of large computational graphs spanning multiple assignments in the presence of aliasing for source transformation based AD. We characterize the variety of semantic-preserving computational graphs that can be generated for a code with aliasing and some consequences of the choices. The algorithm in the paper presents a practical solution for the graph construction that handles the aliasing problem. It enables cross-country elimination and other AD methods that rely on the availability of such graphs in AD tools for general purpose programming languages such as C, C++ and Fortran.

The algorithm of Sect. 4.1 is implemented in the OpenAD[4] framework of the Adjoint Compiler Technology & Standards (ACTS) project. For the ACTS project the alias analysis, the generation of du/ud-chain information, and other analyses are being implemented in OpenAnalysis[5]. The canonicalizations listed in Sect. 4 are done by specific compiler front ends; see, for instance, the OpenADFortTk component in OpenAD for the Open64 Fortran compiler. The algorithm incorporates the handling of the special cases mentioned in Sect. 5.1. While they complicate the case distinction in the algorithm, they do not change its basic functionality and therefore are not added to the formal description given here. Another major addition to the implemented algorithm not covered in this paper integrates an intrinsic-specific activity analysis with the flattening. While the algorithm described here ignores only constant minimal vertices, some intrinsics such as `floor` are constant in open subdomains. Compiler-style activity analysis implemented as a dataflow analysis is typically not concerned with this level of detail. OpenAD considers such intrinsics and can recognize ensuing passive subgraphs across assignment boundaries. Since passive subgraphs do not need to be flattened into G, we do not need unique variable identification for the constant propagation. Instead, we can establish a variable v is constant if we find all assignments occurring in D_{u_v} to be constant as well.

The successful use of OpenAD for oceanographic state estimation problems shows the viability of our approach.

[4] `www.mcs.anl.gov/OpenAD`
[5] `www.hipersoft.rice.edu/openanalysis`

The Adjoint Data-Flow Analyses: Formalization, Properties, and Applications

Laurent Hascoët and Mauricio Araya-Polo

INRIA Sophia-Antipolis, TROPICS team, Sophia-Antipolis, France
{Laurent.Hascoet,Mauricio.Araya}@sophia.inria.fr

Summary. Automatic Differentiation (AD) is a program transformation that yields derivatives. Building efficient derivative programs requires complex and specific static analysis algorithms to reduce run time and memory usage. Focusing on the *reverse mode* of AD, which computes *adjoint programs*, we specify jointly the central static analyses that are required to generate an efficient adjoint code. We use a set-based formalization from classical data-flow analysis to specify *Adjoint Liveness, Adjoint Write,* and *To Be Recorded* analyses, and their mutual influences, taking into account the specific structure of adjoint programs. We give illustrations on examples taken from real numerical programs, that we differentiate with our AD tool TAPENADE, which implements these analyses.

Key words: Adjoint code, adjoint algorithm, data-flow analysis, reverse mode, TAPENADE

1 Introduction

Classically, tools that perform code optimization require information on which variables are used or overwritten by a given piece of code. This is particularly true when trying to optimize *adjoint* code produced by the reverse mode of automatic differentiation (AD), which poses serious problems of run time and memory consumption. To this end, in addition to the classical program analyses, e.g., Read-Write, several research groups have experimented with specific analyses such as *Adjoint Liveness* analysis, *To Be Recorded* analysis, and *Adjoint Write* analysis. These three analyses, defined in Sect. 4, appear tightly related.

Adjoint code has a particular structure, defined by the model of reverse AD. For example, an adjoint code consists of two sweeps with symmetric control flow [254, 257]. It also features matching pairs of instructions that *store* and *restore* values. In our particular model this is done by pushing and popping these values to and from a stack. We are going to use these features of adjoint programs to define the rules of the AD-specific data-flow analyses, by formal specialization of the rules of classical data-flow analyses.

Generic data-flow analyzers, like the ones found in optimizing compilers, are unable to detect nor take advantage of this structure. They can't find the pairs of matching push and pop's far apart, nor can they understand the mechanism used to reproduce the symmetric flow of control. Running them a posteriori on the adjoint program will return weak results. Therefore, we shall define AD-specific data-flow analyses that will run on the original code before generation of the adjoint code, incorporating knowledge of how the adjoint code will be built.

This paper gives a formal uniform specification of the three adjoint data-flow analyses above, defined on the structure of the original code. This specification relies on a preliminary description of the model of reverse AD that goes from the original code to its adjoint. This specification will be used to demonstrate data-flow properties of adjoint codes, to highlight the relationship between these three analyses, and to derive the data-flow equations implemented in our AD tool TAPENADE [257].

Section 2 summarizes knowledge about reverse AD that is necessary for this paper, and Sect. 3 gives basic notation and formulae for classical data-flow analyses. We refer to [225] for a full discussion of AD, and to [3] about data-flow analyses. Section 4 presents the reverse AD model and uses it to define and specify *Adjoint Liveness, To Be Recorded,* and *Adjoint Write* analyses. Section 5 gives an illustrative example adapted from an industrial numerical code and shows experimental measurements.

2 Adjoints by Automatic Differentiation

Automatic Differentiation differentiates *programs*. An AD tool takes as input a source computer program P that, given a vector argument $X \in \mathbb{R}^n$, computes some vector function $Y = F(X) \in \mathbb{R}^m$. We call F' the Jacobian of F. A star * denotes transposition, and the dot $\cdot$ denotes product. In reverse mode, the AD tool generates an adjoint source program $\overline{\mathtt{P}}$ that, given the argument X and a weight vector $\overline{Y}$, computes the gradient $F'^*(X)\cdot\overline{Y}$ of the scalar output $Y^*\cdot\overline{Y}$. To keep things simple, consider that P is a sequence of instructions I_k: AD identifies it with a composition of vector functions so that

$$\mathtt{P} : [I_1; I_2; \dots I_p;] \quad \text{represents} \quad F = f_p \circ f_{p-1} \circ \cdots \circ f_1 \ ,$$

where each f_k is the elementary function implemented by I_k. Call for short X_k the values of all variables after each instruction I_k, i.e. $X_0 = X$ and $X_k = f_k(X_{k-1})$. AD applies the chain rule to obtain the required gradient

$$F'^*(X)\cdot\overline{Y} = f_1'^*(X_0)\cdot f_2'^*(X_1)\cdot \ \dots \ \cdot f_{p-1}'^*(X_{p-2})\cdot f_p'^*(X_{p-1})\cdot\overline{Y} \ , \qquad (1)$$

which can be mechanically translated back into the adjoint program $\overline{\mathtt{P}}$.

We observe that (1) is most efficiently computed from right to left, because matrix-vector products are cheaper than matrix-matrix products. This yields the gradient in a time which is only a small multiple of the time of P. However, there is a difficulty because the f' instructions require the intermediate values X_k in *reverse* of their computation order. If the original program *overwrites* a part of X_k, the differentiated program must restore X_k before it is used by $f_{k+1}'^*(X_k)$. There are two main strategies for that:

- **Recompute-All (RA):** the X_k are recomputed when needed, restarting P on input X_0 until instruction I_k. The brute-force RA strategy has a quadratic time cost with respect to the number of run-time instructions p. The TAF [208] tool uses this strategy. A so-called "Efficient Recomputation Algorithm" limits recomputations to those actually needed to obtain X_k.
- **Store-All (SA):** the X_k are stored on a stack, which is filled during the *forward sweep* $\overrightarrow{\mathtt{P}}$, a preliminary run of P that stores variables on the stack just before they are overwritten. The differentiated instructions strictly speaking form the *backward sweep* $\overleftarrow{\mathtt{P}}$, which pops values from the stack when needed. The brute-force SA strategy has a linear memory cost with respect to p. The ADIFOR [96] and TAPENADE [257] tools use this strategy.

Practically, both RA and SA strategies need a special storage/recomputation trade-off in order to be really efficient. This trade-off is called *checkpointing*. Since TAPENADE uses checkpointing on subroutine calls, we describe checkpointing in this context. Let us define some vocabulary and graphical notations. Execution of a sub-

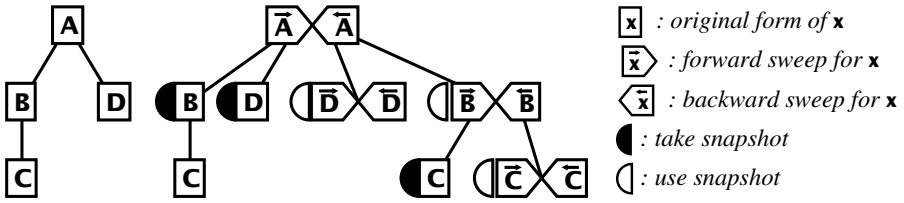


Fig. 1. Checkpointing on calls in TAPENADE reverse AD.

routine A in its original form is shown as A. The *forward sweep* is shown as $\overrightarrow{\mathtt{A}}$. The *backward sweep* is shown as $\overleftarrow{\mathtt{A}}$. The adjoint program is just $\overline{\mathtt{A}} = \overrightarrow{\mathtt{A}}; \overleftarrow{\mathtt{A}}$. Checkpointing consists of choosing a part B of A, which will be run *without* storage during $\overrightarrow{\mathtt{A}}$. When the backward sweep $\overleftarrow{\mathtt{A}}$ reaches B, it runs $\overrightarrow{\mathtt{B}}$, i.e. B again but this time with storage and then immediately runs $\overleftarrow{\mathtt{B}}$ and the rest of $\overleftarrow{\mathtt{A}}$. Duplicate execution of B requires that some variables used by B (a *snapshot*) be stored. TAPENADE applies checkpointing at each procedure call, leading to the pattern shown on Fig. 1. If the run-time call tree is well balanced, the memory size as well as the computation time required for the reverse differentiated program grow only like the depth of the call tree, i.e. like the logarithm of the run time of P.

3 Classical Data-Flow Analyses

We introduce some classical notation and formulae of data-flow analysis. They will be used in the next sections to derive formally specialized rules for adjoint data-flow analyses. Consider any fragment Z of a program P.

- The set of variables whose value at the beginning of Z is overwritten inside Z (at least partly overwritten during some possible execution of Z) is denoted by $\mathbf{out}(Z)$. For two successive pieces of program A and B:

$$\mathbf{out}(A;B) = \mathbf{out}(A) \cup \mathbf{out}(B)\ . \tag{2}$$

However we will use a refined rule when a stack is used: if a variable is PUSH'ed and later POP'ed from a stack, it is unmodified globally, so that

$$\mathbf{out}(\mathrm{PUSH}(v);A;\mathrm{POP}(v)) = \mathbf{out}(A) \setminus \{v\}\ . \tag{3}$$

- The set of variables whose value at the beginning of Z is always completely overwritten inside Z is denoted by $\mathbf{kill}(Z)$. This subset of $\mathbf{out}(Z)$ is often a strict subset, e.g. a single assignment to an array cell does not kill this array, so that $\mathbf{kill}(\texttt{T(i)=0.0}) = \emptyset$. Array region analysis [140] copes with this in some cases. In general for two successive pieces of program A and B we take the conservative under-approximation

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

- The set of variables whose value at the beginning of Z is read inside Z is denoted by $\mathbf{use}(Z)$. For two successive pieces of program A and B, the variables killed by A hide the variables read by B, so that

$$\mathbf{use}(A;B) = \mathbf{use}(A) \cup (\mathbf{use}(B) \setminus \mathbf{kill}(A))\ . \tag{4}$$

- When Z is a tail of `P` (i.e. the end of Z is the end of `P`), we define the set $\mathbf{live}(Z)$ of live variables at the beginning of Z, i.e. whose value is involved in computations that eventually influence the final result of `P`. All final outputs of `P` are live by definition, so we initialize $\mathbf{live}([\,])$ to this set. Recursively, for any two successive pieces of program A and B, B being a tail of `P`, the variables live just before B lead to the variables live just before A through $\mathbf{Dep}(A)$, the "dependence across A" information, defined as

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

and through the combinator $\otimes$, defined as

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

so that

$$\mathbf{live}(A;B) = \mathbf{live}(B) \otimes \mathbf{Dep}(A)\ . \tag{5}$$

4 Adjoint Data-Flow Analyses

We consider a piece of program `P` to be differentiated by the reverse mode of AD into its adjoint program $\overline{\texttt{P}}$. `P` can be the complete function that will be differentiated, or it can be a checkpointed sub-part. In both cases, using notation from Sect. 2, $\overline{\texttt{P}} = \overrightarrow{\texttt{P}}; \overleftarrow{\texttt{P}}$. *Adjoint Liveness* analysis observes that the only required results of $\overline{\texttt{P}}$ are the differentiated variables, and *not* the original results of `P`, which in most implementations will be overwritten and lost during $\overleftarrow{\texttt{P}}$. Therefore, several instructions at the end of $\overrightarrow{\texttt{P}}$ may be dead. This is true in particular for the last instruction of `P`. *Adjoint Liveness* analysis computes, for any location in `P` and thus in $\overrightarrow{\texttt{P}}$, the set of original variables *live* for $\overline{\texttt{P}}$. Adjoint Liveness is computed on original variables only,

but it originates from differentiated variables, which are all considered live, whereas all the original variables are assumed dead at the end of $\overline{\mathtt{P}}$. *Adjoint Liveness* analysis is strongly related to the *To Be Recorded* analysis and the *Adjoint Write* analysis, so that we will define and study the three of them jointly.

Here is an outline of the general structure of this technical section. In Sect. 4.1, we first give a precise specification (or model) of adjoint programs that defines and uses the *Adjoint Liveness, To Be Recorded,* and *Adjoint Write* analyses, to produce an efficient code. Then we formalize these analyses using this model of adjoint programs, starting with *To Be Recorded* analysis in Sect. 4.2. Notice that this might introduce a circularity into the definition. After proving in Sect. 4.3 an important lemma about the variables left unmodified by an adjoint program, we will be able in Sect. 4.4 to formally derive specific rules that define the *Adjoint Liveness* analysis. We can then show that the definitions circularity mentioned above disappears, and consequently the *Adjoint Liveness* analysis must be run first, followed by the *To Be Recorded* analysis and finally by the *Adjoint Write* analysis. Section 4.5 formally derives the specific rules that define the *Adjoint Write* analysis, and highlights its usage for the checkpointing strategy. All three analyses are defined directly on the original program, with a low computational cost.

4.1 Structure of Adjoint Programs

Strictly speaking, the fact that a variable is necessary for (a part of) $\overline{\mathtt{P}}$ depends on the architecture of $\overline{\mathtt{P}}$, i.e. on the reverse AD model and on the strategy used to make intermediate values available to the backward sweep. Here, we shall rely on the SA strategy used in TAPENADE, but the following specifications and demonstrations can be adapted to the RA strategy. Let us make our reverse AD model explicit. We define $\overline{\mathtt{P}}$ recursively on the structure of $\mathtt{P}$. To keep things simple, suppose $\mathtt{P}$ is a straight-line program of simple statements. For an empty program $\mathtt{P} = [\,]$, $\overline{\mathtt{P}}$ is simply $\overline{[\,]} = [\,]$. Recursively, for a assignment I followed by any "downstream" sequel D, the basic model is:

$$\overline{I;D} = \overrightarrow{I};\overline{D};\overleftarrow{I} = \mathrm{PUSH}(\mathbf{out}(I));I;\overline{D};\mathrm{POP}(\mathbf{out}(I));I' \,. \tag{6}$$

It states that values overwritten by I are PUSH'ed onto a stack just before I, and restored by a POP before they may be used by I', the derivative instructions for I. However, at least three refinements can be applied to the model to obtain a more efficient, yet equivalent, adjoint code.

An immediate refinement is to use *activity*, specified for example in [256]: at analysis time, some variables can be proved to have always a zero derivative with respect to the independent inputs or dependent outputs. When the variable written by assignment I is inactive, then I' can be removed. When some variable used by assignment I is inactive, I' is simplified, therefore using fewer intermediate variables. Adjoint data-flow analyses yields even better results when run after activity analysis.

Another refinement is to use *Adjoint Liveness* analysis, defined as computing $\mathbf{live}(\overline{D})$ with the initial condition on the tail of D that $\mathbf{live}(\overline{[\,]}) = \emptyset$. In model (6), $\overline{I;D}$ contains instruction I. We observe that if the results of I are used later in $\overline{\mathtt{P}}$, this can be only in $\overline{D}$ because the backward sweep $\overleftarrow{U;I}$ of I and instructions before I in $\mathtt{P}$ ("upstream") only uses intermediate values that existed *before* execution of I, and certainly not the results of I. Therefore, I and its associated PUSH and POP

can be removed if its results are out of $\mathbf{live}(\overline{D})$, i.e. if the predicate *adj-live*(I, D), defined as $(\mathbf{out}(I) \cap \mathbf{live}(\overline{D}) \neq \emptyset)$, is *false*. One can check this is the case for the last I of `P`, i.e. when $D = [\,]$.

The third refinement is to PUSH and POP a variable in $\mathbf{out}(I)$ *only* if it is really required by the following differentiated instructions. This is the goal of *To Be Recorded* analysis, abbreviated as TBR [168, 397]. For example, the derivative of the "linear" instruction `x = y+2*z` does not require the values of `y` nor `z`. Since this refinement depends on the *following* differentiated instructions, which include the backward sweep of U, the part of `P` *upstream* I, we must introduce this U as a context into definition (6). We use the notation $\vdash$ to separate U from the part of the program currently differentiated. We introduce the set of variables used by instructions I' and after, which is $\mathbf{use}(I'; \overleftarrow{U})$. The only variables actually PUSH'ed and POP'ed for instruction I are now $\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U})$.

Consequently, model (6) turns into the following refined model:

$$\begin{aligned} U \vdash \overline{I; D} = {} & [\mathrm{PUSH}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U})); I;] \ \textit{if adj-live}(I, D) \\ & [U; I] \vdash \overline{D}; \\ & [\mathrm{POP}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U}));] \ \textit{if adj-live}(I, D) \\ & I' \end{aligned} \tag{7}$$

4.2 Derived Rules for To Be Recorded (TBR) Analysis

From the classical equation (4) of the **use** analysis, we can write the rules that compute $\mathbf{use}(I'; \overleftarrow{U})$ and $\mathbf{use}(\overleftarrow{U})$, yielding a formal specification of the TBR analysis. Since I' only overwrites differentiated variables, and we study here data-flow properties of the original variables only, $\mathbf{kill}(I') = \emptyset$. Therefore

$$\mathbf{use}(I'; \overleftarrow{U}) = \mathbf{use}(I') \cup \mathbf{use}(\overleftarrow{U}) \,, \tag{8}$$

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

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

$$\mathbf{use}(\overleftarrow{U; I}) = \begin{cases} \mathbf{use}(\mathrm{POP}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U})); I'; \overleftarrow{U}) \\ \quad = (\mathbf{use}(I') \cup \mathbf{use}(\overleftarrow{U})) \setminus \mathbf{kill}(I) \quad \textit{if adj-live}(I, D) \\ \\ \mathbf{use}(I'; \overleftarrow{U}) = \mathbf{use}(I') \cup \mathbf{use}(\overleftarrow{U}) \qquad \textit{otherwise} \end{cases} \tag{9}$$

These equations translate easily into forward data-flow equations. They can be implemented efficiently as described in [256].

4.3 Adequacy Lemma for the PUSH/POP Mechanism

Equations (8) and (9) allow us to verify an important property of model (7): the PUSH/POP mechanism inside $\overline{D}$ leaves unchanged all variables used in $\overleftarrow{U; I}$, the backward sweep of the upstream instructions:

Lemma 1. *For any tail Z of program* P*, preceded by upstream instructions U:*

$$\mathbf{out}(U \vdash \overline{Z}) \cap \mathbf{use}(\overleftarrow{U}) = \emptyset \ .$$

Proof. By induction on the length of Z. **Terminal case:** if $Z = [\,]$, $U \vdash \overline{Z} = [\,]$ too, so its **out** set is empty, and the property is true. **Induction case:** if $Z = I; D$, then $U \vdash \overline{Z}$ is defined by (7):

- If *adj-live*$(I; D)$ is *false*, then $\mathbf{out}(U \vdash \overline{I; D}) = \mathbf{out}([U; I] \vdash \overline{D}; I') = \mathbf{out}([U; I] \vdash \overline{D}) \cup \mathbf{out}(I')$, from definition (2). We know that $\mathbf{out}(I') = \emptyset$ because I' overwrites only differentiated variables. By the induction hypothesis, $\mathbf{out}([U; I] \vdash \overline{D}) \cap \mathbf{use}(\overleftarrow{U; I}) = \emptyset$. From (9), we find that $\mathbf{use}(\overleftarrow{U; I})$ contains $\mathbf{use}(\overleftarrow{U})$ and therefore $\mathbf{out}([U; I] \vdash \overline{D}) \cap \mathbf{use}(\overleftarrow{U}) = \emptyset$, and the property is true.
- On the other hand, if *adj-live*$(I; D)$ is *true*, consider any variable $v \in \mathbf{use}(\overleftarrow{U})$. This implies through (8) that $v \in \mathbf{use}(I'; \overleftarrow{U})$.
  - Either $v \in \mathbf{out}(I)$. Since *adj-live*$(I; D)$ is *true*, and since v is also in $\mathbf{use}(I'; \overleftarrow{U})$, v will be PUSH'ed and then POP'ed. Thus from (3), v is unchanged just after the POP. Since I' overwrites only differentiated variables, v is unchanged through execution of $U \vdash \overline{I; D}$.
  - Or $v \notin \mathbf{out}(I)$. In that case, the only part of $U \vdash \overline{I; D}$ that might overwrite v is $[U; I] \vdash \overline{D}$. Equation (9) says that $\mathbf{use}(\overleftarrow{U; I}) = (\mathbf{use}(I') \cup \mathbf{use}(\overleftarrow{U})) \setminus \mathbf{kill}(I)$. Since $v \notin \mathbf{out}(I)$, $v \notin \mathbf{kill}(I)$ because the **kill** set is always included in the **out** set. So from $v \in \mathbf{use}(\overleftarrow{U})$ we get $v \in \mathbf{use}(\overleftarrow{U; I})$. The induction hypothesis ensures that $\mathbf{out}([U; I] \vdash \overline{D}) \cap \mathbf{use}(\overleftarrow{U; I}) = \emptyset$, and therefore $v \notin \mathbf{out}([U; I] \vdash \overline{D})$, so v is unchanged through execution of the whole $U \vdash \overline{I; D}$.

  Therefore $v \notin \mathbf{out}(U \vdash \overline{Z})$, and the property is true. □

4.4 Derived Rules for Adjoint Liveness Analysis

We can now specify the rules of Adjoint Liveness analysis. We want to find equations which, for any tail Z of P, build the set $\mathbf{live}(U \vdash \overline{Z})$. By definition $\mathbf{live}(\overline{[\,]}) = \emptyset$. Recursively for $Z = I; D$, recall that Adjoint Liveness originates from differentiated variables. In model (7), only $\overline{D}$ and I' write differentiated variables. Therefore $\mathbf{live}(U \vdash \overline{I; D})$ is the union of the necessary variables of the two slices of $U \vdash \overline{I; D}$ required for $\overline{D}$ and I'.

- One slice for variables that are necessary due to $\overline{D}$:

$$\begin{array}{l}[\mathrm{PUSH}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U})); I;] \ \textit{ if adj-live}(I, D) \\ [U; I] \vdash \overline{D};\end{array}$$

  From (5), the necessary variables are $\mathbf{live}([U; I] \vdash \overline{D}) \otimes \mathbf{Dep}(I)$. This formula applies even when *adj-live*(I, D) is *false* because in this case $\mathbf{out}(I) \cap \mathbf{live}([U; I] \vdash \overline{D}) = \emptyset$, i.e. I doesn't write any variable in $\mathbf{live}([U; I] \vdash \overline{D})$, and thus $\mathbf{live}([U; I] \vdash \overline{D}) \otimes \mathbf{Dep}(I) = \mathbf{live}([U; I] \vdash \overline{D})$.
- Another slice for variables that are necessary due to I'. These variables can be found by a simple analysis of I, not requiring I'. From Lemma 1, the variables necessary for I', which belong to $\mathbf{use}(\overleftarrow{I})$, are left unmodified by $[U; I] \vdash \overline{D}$. The slice is thus

$$
\begin{array}{l}
[\mathrm{PUSH}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U})); I;] \ \textit{if adj-live}(I, D) \\
[\mathrm{POP}(\mathbf{out}(I) \cap \mathbf{use}(I'; \overleftarrow{U}));] \ \textit{if adj-live}(I, D) \\
I'
\end{array}
$$

which is equivalent to I', whatever U, D, and *adj-live*(I, D). Its necessary variables are $\mathbf{live}(I')$.

We end up with the following formula, which does not use the context U:

$$\mathbf{live}(\overline{I; D}) = \mathbf{live}(I') \cup (\mathbf{live}(\overline{D}) \otimes \mathbf{Dep}(I)) \ . \tag{10}$$

A priori, there was a risk of circularity in this specification, since it used the *adj-live* property in many places. However, (10) turns out to be independent from U and *adj-live*(I, D), so there is no circularity. In practice, it suffices to run Adjoint Liveness analysis that computes $\mathbf{live}(\overline{Z})$ before TBR analysis that computes $\mathbf{use}(\overleftarrow{U})$. Equation (10) extends to basic blocks instead of instructions: for any block B followed by a downstream code D

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

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

4.5 Derived Rules for Adjoint Write Analysis

Suppose `P` contains a checkpointed piece C, and thus is made of three parts $[U, C, D]$. Checkpointing modifies model (7), because the forward sweep runs C, and the backward sweep runs $\overline{C} = \overrightarrow{C}; \overleftarrow{C}$. This requires storing a *snapshot*, i.e. enough variables to restore the calling context of C. Storing $\mathbf{use}(C)$ is sufficient, but we can do better. We need to run $\overline{C}$ again, and not C, and we saw that $\mathbf{live}(\overline{C}) \subset \mathbf{use}(C)$. Moreover we need to restore a variable only if it was modified "in between," i.e. is in the **out** set of code sequence $C; \overline{D}$, and we shall use the fact that $\mathbf{out}(\overline{D}) \subset \mathbf{out}(D)$. Therefore, in the non-trivial case where predicate *adj-live*(C, D) is true, we define the snapshot $\mathbf{snp}(U, C, D)$ as $\mathbf{live}(\overline{C}) \cap (\mathbf{out}(C) \cup \mathbf{out}([U; C] \vdash \overline{D}))$, and the reverse AD model becomes

$$
\begin{array}{rl}
U \vdash \overline{C; D} = & \mathrm{PUSH}(\mathbf{out}(C) \cap \mathbf{use}(\overleftarrow{U})); \\
& \mathrm{PUSH}(\mathbf{snp}(U, C, D)); \\
& C; \\
& [U; C] \vdash \overline{D}; \\
& \mathrm{POP}(\mathbf{snp}(U, C, D)); \\
& [\,] \vdash \overline{C}; \\
& \mathrm{POP}(\mathbf{out}(C) \cap \mathbf{use}(\overleftarrow{U}));
\end{array}
\tag{11}
$$

Model (11) is not necessarily optimal. Other choices could perform better for some programs. For example, putting U instead of $[\,]$ as the context for the generation of $\overline{C}$ costs more PUSH/POP inside $\overline{C}$, and on the other hand makes it unnecessary to store $\mathbf{out}(C) \cap \mathbf{use}(\overleftarrow{U})$ in (11). Exploration of these tradeoffs is an open problem. In any case, we must specify the Adjoint Write analysis, to compute $\mathbf{out}(U \vdash \overline{Z})$. If $Z = [\,]$, obviously $\mathbf{out}(U \vdash [\,]) = \emptyset$. If $Z = I; D$, we use model (7) and distinguish

two cases according to *adj-live*(I, D). Using also definition (3) on PUSH/POP pairs, we get

$$\mathbf{out}(U \vdash \overline{I; D}) = \begin{cases} (\mathbf{out}(I) \cup \mathbf{out}([U; I] \vdash \overline{D})) \setminus (\mathbf{kill}(I) \cap \mathbf{use}(I'; \overleftarrow{\overline{U}})) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textit{if adj-live}(I, D) \\ \mathbf{out}([U; I] \vdash \overline{D}) \qquad\qquad \textit{otherwise.} \end{cases} \tag{12}$$

As anticipated, we see that $\mathbf{out}(U \vdash \overline{I; D})$ is always included in $\mathbf{out}(I; D)$, and often strictly thanks to the PUSH/POP pairs. Again, (12) runs backward on a flow graph.

5 Application

Consider the example procedure `FLW2D1COL` (Fig. 2) from a Navier-Stokes flow solver, shortened for readability preserving its structure. On a large mesh, this typical

```
      subroutine FLW2D1COL(nsg1,nsg2,nubo,t3,pres,vnocl,
     +          g3,g4,rh3,rh4,ns,nseg,sq)
      < omitted declarations >
      do 30 iseg=nsg1,nsg2
          is1 = nubo(1,iseg)
          is2 = nubo(2,iseg)
          qsor = t3(is1)*vnocl(2,iseg)
          qs = t3(is2)*vnocl(2,iseg)
          dplim = qsor*g4(is1)+qs*g4(is2)
          rh4(is1) = rh4(is1) + dplim
          rh4(is2) = rh4(is2) - dplim
          pm = pres(is1)+pres(is2)
          dplim = qsor*g3(is1)+qs*g3(is2)+pm*vnocl(2,iseg)
          rh3(is1) = rh3(is1) + dplim
          rh3(is2) = rh3(is2) - dplim
          call CK(pm,sq)
30    continue
      end
```

Fig. 2. An example gather-scatter loop from a real code.

gather-scatter loop accounts for many computations, and thus many derivatives. We differentiate `FLW2D1COL` in the reverse mode with TAPENADE, using the SA strategy. The call to `CK` is checkpointed. Figure 3 shows the resulting subroutine $\overline{\texttt{FLW2D1COL}}$. Differentiated variables are shown with a $\overline{\text{bar}}$ above. Since the loop's iterations are independent, the adjoining operation and the `do` loop operator commute (*cf* [254]), and therefore the resulting subroutine is a single loop, containing a forward sweep followed by a backward sweep. The benefits from the adjoint data-flow analyses are:

- From Adjoint Liveness analysis, variables `dplim`, `rh3`, and `rh4` are not necessary in the adjoint. Furthermore, the call to `CK` is the last instruction in its own

```
      subroutine FLW2D1COL(nsg1,nsg2 nubo,t3,t3b,pres,presb,vnocl,
     +          vnoclb,g3, g3b,g4,g4b,rh3, rh3b,rh4, rh4b,ns,nseg,sq,sqb)
      < omitted declarations >
      do iseg=nsg1,nsg2
          is1 = nubo(1,iseg)
          is2 = nubo(2,iseg)
          qsor = t3(is1)*vnocl(2,iseg)
          qs = t3(is2)*vnocl(2,iseg)
C         dplim = qsor*g4(is1) + qs*g4(is2)
C         rh4(is1) = rh4(is1) + dplim
C         rh4(is2) = rh4(is2) - dplim
          pm = pres(is1) + pres(is2)
C         dplim = qsor*g3(is1)+qs*g3(is2)+pm*vnocl(2,iseg)
C         rh3(is1) = rh3(is1) + dplim
C         rh3(is2) = rh3(is2) - dplim
C         call PUSH(sq)
C         call PUSH(pm)
C         call CK(pm, sq)
      < forward sweep ends, backward sweep begins >
C         call POP(pm)
C         call POP(sq)
          call CKb(pm, pmb, sq, sqb)
          dplimb = rh3b(is1) - rh3b(is2)
          qsorb = g3(is1)*dplimb
          g3b(is1) = g3b(is1) + qsor*dplimb
          qsb = g3(is2)*dplimb
          g3b(is2) = g3b(is2) + qs*dplimb
          pmb = pmb + vnocl(2,iseg)*dplimb
          vnoclb(2,iseg) = vnoclb(2,iseg) + pm*dplimb
          presb(is1) = presb(is1) + pmb
          presb(is2) = presb(is2) + pmb
          dplimb = rh4b(is1) - rh4b(is2)
          qsorb = qsorb + g4(is1)*dplimb
          g4b(is1) = g4b(is1) + qsor*dplimb
          qsb = qsb + g4(is2)*dplimb
          g4b(is2) = g4b(is2) + qs*dplimb
          t3b(is2) = t3b(is2) + vnocl(2,iseg)*qsb
          vnoclb(2,iseg) = vnoclb(2,iseg)+t3(is2)*qsb+t3(is1)*qsorb
          t3b(is1) = t3b(is1) + vnocl(2,iseg)*qsorb
      enddo
      end
```

Fig. 3. The adjoint of subroutine FLW2D1COL from Fig. 2.

checkpointed sub-part (i.e. its downstream sequel is [ ]). Therefore this call can be removed as well as its associated PUSH/POP. Removed statements are here left as comments in boxes.

- From TBR Analysis, variable `dplim` is not used in the backward sweep, and therefore is not saved before it is overwritten. Variable `qsor` is used in the backward sweep $\overleftarrow{\texttt{P}}$, but is not overwritten before this use occurs. This explains there are no PUSH/POP for these variables.
- From Adjoint Liveness and Adjoint Write analyses, **live**($\overline{\texttt{FLW2D1COL}}$) is smaller than **use**(`FLW2D1COL`), and that **out**($\overline{\texttt{FLW2D1COL}}$) is smaller than **out**(`FLW2D1COL`). Specifically, arrays `rh3` and `rh4` are excluded from the snapshot in the procedure that calls `FLW2D1COL`.

We measured the benefits of Adjoint Liveness and Adjoint Write analyses on five large applications that we use as validation tests. Activity and TBR analyses are already applied systematically in TAPENADE, so this experiment strictly shows the additional benefit coming from Adjoint Liveness and Adjoint Write. The results strongly depend on the actual application, but can be quite interesting as shown in Table 1. The speedup ranges between 7% and 18%, and the improvement in memory is between 0% and 49%. The `STICS` code is so large that its adjoint makes a heavy use of the swap. This explains the huge slowdown of the reverse mode, and makes it even more important to save 49% in memory thanks to the Adjoint Write analysis.

Table 1. Time and memory improvements on five large validation codes. We compare run times and maximum stack size for original program, AD adjoint (using activity and TBR), and AD adjoint using adjoint liveness and adjoint write analyses.

Code name:	`ALYA`	`UNS2D`	`THYC`	`LIDAR`	`STICS`
Application domain:	*CFD*	*CFD*	*Thermo*	*Optics*	*Agronomy*
Original program runtime (sec.):	0.85	2.39	2.67	11.22	1.80
Adjoint program runtime (sec.):	5.65	29.70	11.91	23.17	42.60
... after adjoint data-flow analysis:	4.62	24.78	10.99	22.99	35.70
Improvement:	18%	16%	8%	7%	16%
Adjoint program memory use (Mb):	10.9	260	3614	16.5	456
... after adjoint data-flow analysis:	9.4	259	3334	16.5	230
Improvement:	14%	0%	8%	0%	49%

6 Conclusion

We have described the adjoint data-flow analyses that help Automatic Differentiation tools improve the performances of adjoints produced by the reverse mode. These analyses rely on the special structure of adjoint programs. However, these are data-flow analyses, and they can be described with the classical set-based notations used in compiler theory. To take full advantage of the knowledge of the reverse AD model, we view these analyses as specific data-flow analyses on the original source,

rather than as generic data-flow analyses on the adjoint source. We obtain the data-flow equations of adjoint analyses on the original source by formal specialization of the standard data-flow equations with respect to the reverse AD model. In addition, we obtain a global view that clarifies the relationship between adjoint data-flow analyses, and formal proofs of fundamental properties of our reverse AD model. We advocate this sort of transposition of techniques that originate from compilation into AD technology.

The goal of producing optimal adjoint programs is still not completely reached, and several other program optimizations will be necessary. A formal description of analyses for adjoint programs is useful to define and compare these analyses yet to come. In particular, we pointed out the link between TBR analysis and snapshots: finding the optimal tradeoff that minimizes the total memory use would be a useful contribution.

Our AD tool TAPENADE progressively implements the analyses we described here, and our first experiments show that this is definitely worthwhile.

Semiautomatic Differentiation for Efficient Gradient Computations

David M. Gay

Optimization and Uncertainty Estimation Department, Sandia National Laboratories**
dmgay@sandia.gov

Summary. Many large-scale computations involve a mesh and first (or sometimes higher) partial derivatives of functions of mesh elements. In principle, automatic differentiation (AD) can provide the requisite partials more efficiently and accurately than conventional finite-difference approximations. AD requires source-code modifications, which may be little more than changes to declarations. Such simple changes can easily give improved results, e.g., when Jacobian-vector products are used iteratively to solve nonlinear equations. When gradients are required (say, for optimization) and the problem involves many variables, "backward AD" in theory is very efficient, but when carried out automatically and straightforwardly, may use a prohibitive amount of memory. In this case, applying AD separately to each element function and manually assembling the gradient pieces — semiautomatic differentiation — can deliver gradients efficiently and accurately. This paper concerns on-going work; it compares several implementations of backward AD, describes a simple operator-overloading implementation specialized for gradient computations, and compares the implementations on some mesh-optimization examples. Ideas from the specialized implementation could be used in fully general source-to-source translators for C and C++.

Key words: Semiautomatic differentiation, mesh elements, manual assembly, Jacobian-vector products, C/C++ source-to-source, TFad

1 Introduction

Many large-scale computations concern partial differential equations (PDEs) based on physical systems and thus involve discretizations that approximate physical objects on meshes. Such discretizations generally yield systems of nonlinear equations whose residuals involve the elements of a mesh. As a PDE model matures, interest often grows in optimizing some aspects of the model, i.e., of solving optimization

** Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. This document is released as SAND Number 2004-4688P.

problems with PDE constraints. Like the constraint residuals, the objectives are generally sums of contributions from functions of the mesh elements.

For solving both discretized PDEs themselves and optimization problems involving them, partial derivatives (or approximations to them) are required. Conventionally, these partials are often approximated by finite differences, but finite differences have several drawbacks. Finding suitable step sizes that balance truncation and round-off error can be tricky, and the overall error in a finite-difference approximation can contribute to computational difficulties. Moreover, when partials with respect to many variables are required, finite differences can be slow. Automatic differentiation (AD) provides a more accurate and often faster alternative to finite differences.

As Griewank [223] showed in a survey that appeared in 1989, AD has been reinvented many times. His more recent book [225] tells much more about AD than we will discuss here. Of primary interest here are first derivatives, which may be computed either by *forward AD*, in which one recurs the desired partials while carrying out each operation in the expression of interest, or by *backward AD*, in which one first evaluates an expression, then visits its operations again in reverse order to recur partials (so-called *adjoints*) of the final expression result with respect to the result of each operation.

Forward AD works well when only a few independent variables are involved, but its complexity can grow with the number of independent variables. In particular, when there is only one independent variable, forward AD can efficiently and conveniently compute derivatives of high order. Nonlinear equations can be solved by matrix-free Newton-Krylov methods, which simply use Jacobian-vector products. Such computations effectively involve just one independent variable, and are well handled by forward AD. (For example, TFad [14] works well in some applications at Sandia National Labs [22].)

When there are many independent variables, as is often the case in optimization problems, backward AD is attractive for computing gradients. It delivers function and gradient in time proportional to that required for a function evaluation alone. Unfortunately, when used straightforwardly, backward AD may require memory proportional to the number of operations in the function evaluation, which may appear prohibitive in large-scale computations.

The rest of this paper is organized as follows. The next section gives more discussion of computations on a mesh. Section 3 reviews some currently available ways to implement AD. Section 4 describes a new, simple, specialized implementation of backward AD by operator overloading in C++. Some timing results on mesh-optimization objectives appear in Sect. 5. Section 6 discusses implications for source-to-source transformation of C and C++, and Sect. 7 offers concluding remarks.

2 Action on a Mesh

As mentioned in Sect. 1, many large-scale computations involve meshes. For optimization problems whose objectives and constraints involve sums of functions of mesh elements, one can use backward AD to compute (separately) the contributions of each mesh element to the objective and constraint gradients and manually assemble these pieces into the overall gradients, an approach sketched in [1]. Preliminary

investigations suggest that this approach should work well in some problems of interest at Sandia, such as mesh optimization (moving interior mesh points to improve the quality of a given mesh) and PDE-constrained optimization. Only a few kinds of mesh elements appear in such problems, making it feasible to treat each separately, either by operator overloading or by source transformation and optimization of routines that compute the element functions. This results in what might be called semiautomatic differentiation: combining use of an AD tool with manual assembly. An advantage of this approach is that it greatly reduces the memory requirements of backward AD.

3 Some AD Alternatives

Straightforward use of AD is facilitated by various tools, such as those listed on the `autodiff.org` web site [16]. These tools work with computations expressed in a suitable programming language, such as C++ or Fortran 95, or a special-purpose language, such as MATLAB [357] or AMPL [184, 185], and use several implementation techniques, as sketched below.

3.1 Operator Overloading

Perhaps the most straightforward implementation technique is operator overloading in languages that support it, such as C++ and Fortran 90. An excellent, general, and often used example for C++ is ADOL-C [228, 229]. Use of ADOL-C requires some simple source modifications, which are typical of the sort of modifications needed by AD tools that work with conventional programming languages. Variables with respect to which derivatives are required, and all variables computed from them, must be given a special type. When such "active" variables appear in an "active section," delimited in ADOL-C by `trace_on` and `trace_off` statements, ADOL-C records arithmetic operations on the variables in a "tape," a data structure that summarizes the computation. Subsequently the tape can be "played" to carry out various AD computations. (Forward AD using the operator-overloading approach does not require use of a tape, but ADOL-C gains flexibility and generality from its use of tapes.) With ADOL-C, a special syntax involving `<<=` indicates assignment of input values to the input variables, and another syntax involving `>>=` indicates assignment of output values, partial derivatives of which can be computed subsequently by AD. The process of recording a tape is somewhat slow (as indicated by the timings in Sect. 5), but once a tape has been recorded, it can be reused with different values of the input variables, so long as all logical expressions involving active variables come out the same as during the taping. Reusing a previously recorded tape is faster than recording a new one, as illustrated in Sect. 5.

3.2 Source Transformation

Source transformation is an implementation technique that can give faster execution than straightforward operator overloading. The idea is for a tool to rewrite a computation expressed in given imperative language, such as C or Fortran, giving a more elaborate computation in the same language that carries out the original

computation along with automatic differentiation thereof. An early general-purpose instance of this approach is Kedem's use [302] of the AUGMENT preprocessor [139] to carry out forward AD or computation of Taylor coefficients for computations expressed in Fortran 66. AUGMENT effectively implemented operator overloading via source transformation and did not attempt to optimize the computations. ADIFOR [55, 57] is a more recent effort that addresses Fortran 77 and does backward AD within statements while carrying out forward AD overall, thus often achieving greater efficiency than a simple forward AD computation would give. ADIFOR does not use a tape, which also helps make its forward-mode computations faster than those of ADOL-C. A still more recent effort is TAF [164], a commercial successor to TAMC [207] that addresses Fortran 95 and with which several speakers reported impressive numerical results at the Fourth International Conference on Automatic Differentiation [15].

3.3 Special Compiler

A variant of source transformation is to have the compiler itself recognize special types and statements that cause AD computations. The NAGWare Fortran 95 compiler [388, 400] provides an example of this approach.

3.4 Implicit Domain Knowledge

Special-purpose languages can exploit automatic differentiation without the small syntactic burden imposed on users of general-purpose programming languages. For example, users of the AMPL language for mathematical programming [184, 185] merely express objectives and constraints in a mathematical notation without indicating anything about the partial derivatives that a solver might need. The system deduces "active" variables behind the scenes and arranges AD computations where needed.

3.5 Interpreted Evaluations

Interpreted evaluations are an implementation technique that offers considerable flexibility at some cost of speed. Rather than compiling problem-specific machine code, one uses expression representations that are constructed and evaluated easily "on the fly." There are many ways to handle the details (and the distinction between compiled and interpreted evaluations can become murky). One can define a virtual machine in the style of Pascal or Java. Logically equivalent to a virtual machine is the list of 4-tuples of integers that GlobSol [299] uses, the first indicating an operation, the latter three operands.

Another logically equivalent form of interpreted evaluation is to use function pointers in an expression graph. Because of its convenience, this is the approach taken by AMPL and its solver-interface library [196]. Timings involving this approach appear below, so sketching some more of its details seems appropriate. Each operation is represented by a structure with pointers to operands and to a function that carries out the operation and stores partial derivatives for use in reverse AD. For example, a binary operation in a setting where function and gradient are desired may have the form (in C notation)

```
struct expr {
            real (*op)(expr*);
            expr *L, *R; /* left and right operands */
            real dL, dR; /* left and right partials */
            };
```

and the `op` in an `expr` for a multiplication operation might be

```
real OPMULT(expr *e) {
           e->dR = (*e->L->op)(e->L);
           e->dL = (*e->R->op)(e->R);
           return e->dR * e->dL;
           }
```

In reality, there may be other fields and auxiliary variables and a different layout that considers alignment, but this illustrates the gist of the approach.

With this latter approach, when setting up the data structures, one can arrange for the backwards computation of adjoint values to be carried out by a very simple loop, as illustrated by Fig. 1, in which a `derp` describes a *der*ivative *p*ropagation operation. The initial assignment of `1.` reflects that the partial of the final result f with respect to itself is 1, i.e., $\frac{\partial f}{\partial f} = 1$. Each iteration of the loop in Fig. 1 updates the adjoint corresponding to an operand of one of the operations in the computation. Both `d->a` and `d->c` point to adjoints, and `d->b` points to a partial derivative; unary `*` dereferences pointers, so `*d->a` is the adjoint to which `d->a` points. Thus "`*d->a += *d->b * *d->c`" adds the product of `*d->b` and `*d->c` to the adjoint `*d->a`.

```
void derprop(derp *d) {
            *d->b = 1.;
            do *d->a += *d->b * *d->c;
                    while(d = d->next);
            }
```

Fig. 1. Backward propagation of adjoints in the AMPL/solver interface library.

3.6 Optimized Compiled Evaluations with *nlc*

For solving nonlinear programming problems, the above style of interpreted evaluations often suffices when the times taken by other parts of the computation dominate the times taken to carry out function and gradient evaluations. In some settings such interpreted evaluations may be too slow, so it is interesting to ask about the extent to which the evaluations can be made faster by generating and compiling problem-specific source code. For example, doing a multiplication directly rather than invoking `OPMULT` will save call-overhead time, and the computations carried

out by `derprop` often involve adding zero to a number or multiplying a number by one and thus present opportunities for optimization when we generate problem-specific source code. The *nlc* program [197] carries out such code optimizations in the process of writing C or Fortran to compute function and gradient values for the objectives and constraints expressed in a "`.nl`" file, which AMPL writes to convey problem information to solvers. The test results in Sect. 5 below include times from C produced by *nlc*.

One drawback of *nlc* is that AMPL only expresses primitive-recursive functions, i.e., those that can be turned into in straight-line code (with no loops — only forward branches). Imported functions provide an escape hatch that permits anything to be computed, but AMPL's imported functions must provide partial derivatives with respect to their arguments for use in AD computations.

4 The *RAD* Package for Reverse AD

It seems interesting to ask how efficiently we can carry out function and gradient evaluations with an implementation of operator overloading in C++ that is specialized for such computations. To this end, I have written a simple backwards AD package, *RAD* (for Reverse AD), that consists of a header file, `rad.h`, and a source file of auxiliary functions; see [200]. When a function is evaluated, *RAD* sets up data structures that permit the backwards AD sweep to take a form similar to that in Fig. 1. This form is shown in Fig. 2, in which each `aval` is an *a*djoint *val*ue, and `*d->a` is a partial derivative.

```
for(; d; d = d->next)
        d->c->aval += *d->a * d->b->aval;
```

Fig. 2. Inner loop of *RAD*'s `ADcontext::Gradcomp();`

One target use for *RAD* is computing a sum of functions defined on mesh elements, with a separate evaluation of function and gradient on one mesh element before moving on to the next one, and with manual summing of the element gradients into the overall gradient. Because of this goal, memory is allocated in large chunks that are not freed, but are retained for reuse on subsequent mesh elements, thus reducing the overhead of allocating small objects and eliminating the overhead of freeing them.

With *RAD*, "active" variables that appear in function evaluations have type `ADvar`. Independent `ADvar` variables — inputs with respect to which partial derivatives are desired — are simply assigned numeric values. Dependent `ADvar` variables are computed from expressions involving independent ones, previously computed dependent ones, and any other numeric values with respect to which partial derivatives are not needed. Dependent `ADvar` variables may be updated as desired, and all `ADvar` variables may participate in loops and function calls without restriction. Once the dependent `ADvar` variable representing the function result has been assigned its

(final) value, one invokes

```
ADcontext::Gradcomp();
```

to cause the backwards AD sweep and reclamation of memory used for the computation just completed. Because the memory is not freed, the last value assigned to an `ADvar v` and the corresponding adjoint value (computed by `ADcontext::Gradcomp()`) remain available as `v.val()` and `v.adj()`, respectively, until the next assignment to an `ADvar`, which will start reusing the allocated memory.

What enables `ADvar` values to be updated is that an `ADvar` is implemented as a pointer to a structure that contains fields for the `val()` and `adj()` values of the `ADvar`'s current value and for partial derivatives associated with the operation that gave the `val` field its value. In Fig. 2, `d->c->aval` and `d->b->aval` are `adj()` fields, and `d->a` points to a partial derivative. When an `ADvar` is updated, it is adjusted to point to a new structure.

As an example on which we report timings in Sect. 5, Fig. 3 shows source for a function, `phi1(x,g)`, that returns a quality measure, $\phi_1(A)$, for an element of a three-dimensional mesh [186, 187] and stores its gradient in the second argument. The function $\phi_1(A)$ is given by

$$\phi_1(A) = \frac{3\det(AW^{-1})^{2/3}}{\| AW^{-1} \|_F^2} , \tag{1}$$

in which the 3×3 matrix A has the form

$$A = [v_1 - v_0, v_2 - v_0, v_3 - v_0] ,$$

where v_0, v_1, v_2, and v_3 are four vertices of a mesh element. The 3×3 matrix W is constant for each kind of mesh element and represents an ideal shape; the source in Fig. 3 deals with one kind of mesh element, and the multiplication AW^{-1} is computed in the assignments to the `aw` array. The coordinates of the v_i appear in successive components of the incoming `xx` array. Note how the gradient components are read out after the invocation of `ADcontext::Gradcomp()` and how `f.val()` is returned as the function value.

5 Test Results

We report comparative timings of some alternative ways of carrying out function and gradient evaluations by backwards AD. The timings were done on two Linux machines, *Desktop* with a 3 GHz Intel Xeon processor having 512 MB of cache, and *Laptop* with a 1.6 GHz Intel Pentium M processor having no cache. Compilation was with `g++ -O` or `gcc -O`, and the same binaries ran on both machines. The reason for showing results from these two machines is to illustrate that architectural details (such as cache) can affect relative timings.

Table 1 shows timings for the function $f = \phi_1$ given by (1). The timings are relative to the time for computing f alone by C++ code similar to that in Fig. 3, with "`ADvar`" replaced by "`double`" and without references to `g` or `ADcontext::Gradcomp()`. The time per function or function and gradient evaluation behind each table entry was computed in a separate timing loop that ran for several seconds. (On *Desktop*, the computations should all have been running in the cache. This seems fair, as we would try to organize the evaluation of a mesh

```
double phi1(double *xx, double *g) {

      ADvar aw[3][3], det, f, x[4], y[4], z[4];
      int i, j;
      static double one_over_root3 = sqrt(1./3.),
            two_over_root3 = sqrt(4./3.),
            one_over_root6 = sqrt(1./6.),
            root_3_halves  = sqrt(3./2.);

      for(i = j = 0; i < 12; i += 3, j++) {
            x[j] = xx[i];
            y[j] = xx[i+1];
            z[j] = xx[i+2]; }
      for(i = 1; i <= 3; i++) {
            x[i] -= x[0];
            y[i] -= y[0];
            z[i] -= z[0]; }
      aw[0][0] = x[1];    aw[1][0] = y[1];   aw[2][0] = z[1];

      aw[0][1] = two_over_root3*x[2] - one_over_root6*x[1];
      aw[1][1] = two_over_root3*y[2] - one_over_root6*y[1];
      aw[2][1] = two_over_root3*z[2] - one_over_root6*z[1];
      aw[0][2] = root_3_halves*x[3] - one_over_root6*(x[1] + x[2]);
      aw[1][2] = root_3_halves*y[3] - one_over_root6*(y[1] + y[2]);
      aw[2][2] = root_3_halves*z[3] - one_over_root6*(z[1] + z[2]);

      for(f = 0., i = 0; i < 3; i++)
            for(j = 0; j < 3; j++)
                  f += aw[i][j]*aw[i][j];

      det =   aw[0][0]*aw[1][1]*aw[2][2]
            + aw[1][0]*aw[2][1]*aw[0][2]
            + aw[2][0]*aw[0][1]*aw[1][2]
            - aw[2][0]*aw[1][1]*aw[0][2]
            - aw[1][0]*aw[0][1]*aw[2][2]
            - aw[0][0]*aw[2][1]*aw[1][2];

      f = 3*pow(det, 2./3.) / f;

      ADcontext::Gradcomp();

      for(i = j = 0; i < 12; i += 3, j++) {
            g[i]   = x[j].adj();
            g[i+1] = y[j].adj();
            g[i+2] = z[j].adj(); }
      return f.val();
      }
```

Fig. 3. Source for `phi1(x,g)` corresponding to (1).

Table 1. Relative times for $f = \phi_1$.

	Desktop	*Laptop*
Compiled f	1.	1.
$f + \nabla f$ by *RAD* (§4)	11.0	10.1
$f + \nabla f$ by *nlc* (§3.6)	1.35	1.53
ADOL-C taped f (§3.1)	4.83	5.54
" taped $f + \nabla f$	14.5	14.9

objective so much of the inner loop would involve data and instructions from the cache.)

The last two lines of Table 1 are for ADOL-C evaluating a previously recorded tape. The computation of f from the tape looks quite efficient. That *RAD* outperforms ADOL-C when computing f and ∇f confirms that specialized operator overloading for AD can be worthwhile. The *nlc* evaluations look remarkably efficient, delivering on the promise of AD to compute f and ∇f in a small multiple of the time for computing f alone.

Some of the overhead in evaluating ϕ_1 and $\nabla\phi_1$ is masked by the time taken by the `pow` invocation in Fig. 3, i.e., by raising $\det(AW^{-1})$ to the power 2/3. We can eliminate this overhead by dealing with $\phi_2 = (\phi_1/3)^3$, i.e.,

$$\phi_2(A) = \frac{\det(AW^{-1})^2}{\| AW^{-1} \|_F^6} \ . \tag{2}$$

Using ϕ_2 for mesh optimization (the problem giving rise to ϕ_1) is not necessarily desirable because ϕ_2 penalizes "large" elements much more than ϕ_1 does, but it is interesting to see how the values in Table 1 change when the overhead of exponentiation goes away. Table 2 gives relative timings for (2); the overheads for all the variants of computing ∇f go up but are qualitatively similar to those in Table 1, and the *nlc* evaluations still give f and ∇f in less than thrice the time of computing f alone.

Table 2. Relative times for $f = \phi_2 = (\phi_1/3)^3$.

	Desktop	*Laptop*
Compiled f	1.	1.
$f + \nabla f$ by *RAD*	37.8	27.2
$f + \nabla f$ by *nlc*	2.54	2.13
ADOL-C taped f	16.6	13.7
" taped $f + \nabla f$	55.6	40.0

We conclude this section by showing timings on a more elaborate mesh-quality function [313], $\mu_1(A)$, defined by (3)–(5):

$$\tau = \det(AW^{-1}), \tag{3}$$

$$h = \frac{1}{2}(\tau + \sqrt{\tau^2 + 4\delta^2}), \tag{4}$$

$$\mu_1(A) = h^{-2/3} \| AW^{-1} - I \|_F^2 \ . \tag{5}$$

The 3×3 matrices A and W in (3) and (5) are as in (1), and δ in (4) is a constant. Note that evaluating $f = \mu_1$ involves extra overhead from both exponentiation and a square-root computation.

Relative timings for $f = \mu_1$ appear in Table 3. All times are for evaluations of f and ∇f. The "Compiled f" times are for hand-coded function and gradient evaluations. They factor A, compute $\det(A)$ from the factorization, and use the identity

$$\frac{\partial \log \det A}{\partial t} = \operatorname{trace}\left(A^{-1}\frac{\partial A}{\partial t}\right)$$

in computing ∇f, in part because this machinery is useful in computing $\nabla^2 f$, a matter discussed briefly in Sect. 7 below. Even with the factorization, etc., done with inline, loop-free code, the calculation is slightly slower than the corresponding one derived by applying *nlc* to the AMPL model shown in Fig. 4, so the times in Table 3 are relative to these *nlc* times.

```
var xyz{i in 0..2, j in 0..2};
var winv{0..2, 0..2};   # really a constant param
var delta := .1;        # really a constant param

var aw{i in 0..2, j in 0..2} = sum{k in 0..2} xyz[i,k]*winv[k,j];

var det = aw[0,0]*aw[1,1]*aw[2,2]
        + aw[1,0]*aw[2,1]*aw[0,2]
        + aw[2,0]*aw[0,1]*aw[1,2]
        - aw[2,0]*aw[1,1]*aw[0,2]
        - aw[1,0]*aw[0,1]*aw[2,2]
        - aw[0,0]*aw[2,1]*aw[1,2];

var h = 0.5 * (det + sqrt(det^2 + 4*delta^2));

var mu1a = 0.5 * sum{i in 0..2, j in 0..2}
                        (aw[i,j] - if i == j then 1)^2;
minimize mu1: mu1a / h^(2/3);
```

Fig. 4. AMPL model for μ_1.

The ASL times are for interpreted evaluations of Fig. 4 with the AMPL/solver interface library, as in Sect. 3.5. When set up to do Hessian computations, these evaluations incur the extra overhead during function evaluations of storing some second partial derivatives. This overhead is reflected in the "ASL for $\nabla^2 f$" line of Table 3.

The "ADOL-C new tape" times in Table 3 show the cost with ADOL-C of recording a tape. These times are to be contrasted with those in the "ADOL-C old tape" line for reusing a previously recorded tape, and with those in the *RAD* line for overloading specialized to f and ∇f.

Table 3. Relative times for $f = \mu_1$ and ∇f.

	Desktop	*Laptop*
Hand-coded	1.07	1.12
ASL	11.3	11.6
ASL for $\nabla^2 f$	13.0	13.4
RAD	9.14	7.06
nlc	1.	1.
ADOL-C new tape	55.0	37.7
ADOL-C old tape	15.4	14.1

6 Implications for Source Transformation

The optimizations done by the *nlc* program could also be done (at least on straight-line code) by a source-to-source translator or special compiler that focused on automating gradient computations. The gap between the times for *RAD* and *nlc* in Table 3 reflects the opportunities mentioned in Sect. 3.6 for optimization in such transformations. Of course, like *RAD*, such transformations should handle completely general source, with only the usual limitations on AD computations. (For example, AD applied to"`(x == 3 ? 5 : x + 2)`" would compute 0 rather than 1 for the derivative at `x` = 3.) The approach taken in *RAD* could work well in such transformations, at least as long as sufficient memory is available. This approach would present various opportunities to further reduce overheads by computing some things at compile (or transformation) time and thus to speed up the computations.

7 Concluding Remarks

One motivation for this work was to research AD approaches that might work well on an objective function defined on elements of a mesh, particularly when the objective is the sum of functions computed on individual mesh elements. Although the memory required for straightforward backward AD could be prohibitive on large meshes, little memory may be needed to compute a function and its gradient on an individual mesh element, and assembling the individual mesh-element gradients into an overall objective gradient "by hand" may be straightforward. Thus we obtain a reliable and efficient way to carry out function and gradient evaluations for some problems (albeit not necessarily for problems with objectives or constraints that involve integration over time — unless time is treated analogously to the spacial dimensions).

An AD approach introduced in this paper is the *RAD* package for function and gradient computations via operator overloading in C++. Since it is fully general and easy to use, *RAD* may find uses in various applications. The implementation techniques described in Sect. 4 could prove useful in special source-to-source translators or compilers meant to facilitate AD computations.

A growing number of nonlinear programming solvers use Hessians (matrices of second partials), so it is of interest to see how easily we can arrange for their efficient computation. The interpreted Hessian evaluations offered by the AMPL/solver interface library [197,198] are convenient but not very fast. For example, for the function

$f = \mu_1$ given by (5), hand-coded evaluations of f, ∇f, and $\nabla^2 f$ run about 100 times faster than computations with the AMPL/solver interface library. It would be interesting to see how much these computations could be sped up by an extension of *nlc* that addressed Hessian computations along with functions and gradients.

Acknowledgment

I thank Scott Mitchell and the referees and editors for their helpful comments on the manuscript.

Computing Adjoints with the NAGWare Fortran 95 Compiler

Uwe Naumann[1]* and Jan Riehme[2]

[1] Software and Tools for Computational Engineering, RWTH Aachen University, Aachen, Germany
naumann@stce.rwth-aachen.de
[2] Department of Mathematics, Humboldt University Berlin, Berlin, Germany
riehme@mathematik.hu-berlin.de

Summary. We present a new experimental version of the differentiation-enabled NAGWare Fortran 95 compiler (from now on referred to as "the AD compiler") that provides support for the computation of adjoints in the reverse mode of automatic differentiation (AD) [42, 136, 227]. Our implementation uses split program reversal [225, Chapter 10] in conjunction with a stack of gradients of all assignments executed inside the active section. Two papers describe the modifications of the compiler infrastructure that were required to provide forward-mode AD capabilities [126, 401]. The reverse mode presented in this paper makes extensive use of these features.

Special emphasis is put on the presentation of the new user interface that provides a very easy and intuitive way for initiating derivative computations as well as for addressing the results. Various language extensions are introduced for this purpose. The compiler front-end is modified to accept these new constructs syntactically and semantically. The use of the language extensions triggers the automatic generation of derivative codes of various kinds by the compiler.

Key words: AD compiler, reverse mode, preaccumulation, taping, CompAD

1 Aims of the CompAD Project

The Compiler-based Automatic Differentiation (CompAD) project is a collaboration between the authors, the Numerical Algorithms Group (NAG), Oxford, and the University of Hertfordshire, Hatfield, UK. Initial funding was provided by the U.K. Engineering and Physical Sciences Research Council under its joint grant schema

* Supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38 and by the National Science Foundation under Information Technology Research contract OCE-0205590.

with MoD: GR/R55252/01 ("Differentiation-enabled Fortran 95 Compiler Technology") until 2002. The results described in this paper are based on work done during this first part of the project. The aims of the CompAD project can be summarized as follows.

- Investigation of AD algorithms in the context of an industrial-strength Fortran 95 compiler with the objective to provide AD technology to a wide range of applications in science and engineering;
- Explicit use of Fortran 95 language features, such as user-defined types, modules, dynamic memory allocation;
- Proof-of-concept implementation covering a relevant subset of Fortran 95;
- Collection of feedback from potential users who are working in the field of numerical simulation and optimization;
- Search for external support to secure continued funding with the aim of improving robustness and efficiency of the implementation through algorithmic advances.

We have emphasized the provision of a suitable infrastructure allowing us to perform AD on the abstract internal representation of the NAGWare Fortran 95 compiler. Several algorithms for generating tangent-linear and adjoint code based on preaccumulated gradients of scalar assignments were developed and implemented. The integration of AD-specific language extensions into the syntax and semantics accepted by the compiler results in a very convenient and intuitive user interface. The scientific computing community will draw substantial benefits from the availability of the new features. Further work (and funding) is required to promote the compiler as a standard tool even for large-scale numerical simulation and optimization. We are convinced that the provision of limited (in terms of the complexity of the computation) adjoint mode capabilities is a step in the right direction.

2 Compiler AD – A Motivating Example

As a motivating example we consider the computation of the gradient for the elastic plastic torsion problem from the MINPACK-2 [17] test problem collection. The source code of the subroutine `ept` that implements a function $f : \mathbb{R}^n \to \mathbb{R}$ is rather small, consisting of only 40 lines of Fortran code. However, the number of independent variables can be increased arbitrarily, making this problem suitable for comparing forward and reverse modes of the AD compiler.

The gradient $f' = \frac{\partial f}{\partial \mathbf{x}}$ can be computed in forward mode as $\dot{f} = f'\dot{\mathbf{x}}$ at a computational cost proportional to n. For this purpose, $\dot{\mathbf{x}}$ is set equal to the identity matrix in $\mathbb{R}^n$. It is well known that for large n this dependence leads to unacceptable execution times. The method of choice is the reverse mode, which computes $\bar{\mathbf{x}} = (f')^T\bar{f}$, where $\bar{f} = 1$, at a small multiple of the cost of running the original function. An example of how the reverse mode is used with the AD compiler is shown below.

```
SUBROUTINE EPT_REVERSE(nx,ny,x,f,grad)
  USE ACTIVE_MODULE
  INTEGER :: nx, ny
  REAL :: x(nx*ny), f
  REAL :: c=1.4
```

```
  REAL, DIMENSION(:), ALLOCATABLE, INTENT(OUT) :: grad

  DIFFERENTIATE(AD_REVERSE)
  INDEPENDENT(x,ADJOINT=grad)
  CALL EPT(nx,ny,x,f,c)
  DEPENDENT(f)
  END DIFFERENTIATE

END SUBROUTINE EPT_REVERSE
```

The computation to be differentiated is included in the active section framed by `DIFFERENTIATE ... END DIFFERENTIATE`. It can be arbitrarily complex including subroutine calls as well as non-trivial flow of control. The mode of differentiation is passed as an argument to `DIFFERENTIATE`. We declare `x` to be `INDEPENDENT` and want its adjoint (the gradient of the `DEPENDENT` variable `f` with respect to `x`) to be stored in `grad`. The `USE` of `ACTIVE_MODULE` and the compilation with the AD compiler results in an executable that computes `grad` alongside with `f` at a computational complexity that does not depend on `nx` or `ny`. Further description of the individual features of the AD interface is given in Sect. 4.

To quantify the differences in scaling between the compiler's forward and reverse modes, we run both algorithms with increasing numbers of independent variables. The results are summarized in the following table.

`n=nx*ny`	Forward (sec.)	Reverse (sec.)	Divided Differences (sec.)
2,500	3.1	0.1	2.3
5,000	12.3	0.2	9.4
10,000	50.0	0.5	36.5
20,000	782	2.5	158
30,000	-	5.5	*
250,000	-	220	*
500,000	-	1200	*

We see the expected behavior. Vector forward mode aborts when reaching the limits of the available virtual memory (1.5GB) for 30,000 independent variables. The reason for this behavior is the allocation of derivative components with 30,000 elements for each intermediate variable. On the same problem divided differences runs for about 50 minutes. Scalar forward mode performance is similar to that of divided differences. Reverse mode scales well, succeeding on much larger problems. It uses 514MB for a problem with 500,000 independent variables. The numerical values computed by the code that is generated by the AD compiler match with those obtained by running the available hand-written gradient code.

The paper is structured as follows. In Sect. 3 we recall the theoretical background on which the proposed approach to reverse mode is built. The integration of adjoint capabilities into the compiler is described in Sect. 4. In Sect. 5 we discuss *seeding* in the context of the new user interface and its use in the computation of the compressed Jacobian and its transpose for a practically relevant test problem. Conclusions are drawn in Sect. 6.

3 Linearization of the Computational Graph

Our version of the reverse mode is based on the interpretation of the vector function

$$F : I\!R^n \to I\!R^m, \quad \mathbf{y} = F(\mathbf{x})$$

as a linearized computational graph. Moreover, local gradients are preaccumulated at the level of scalar assignments as described in [396], leading to a reduction of the graph's size. The reduced graph is generated and stored during the evaluation of F, and it is subsequently used for the reverse propagation of adjoints by exploiting the chain rule. We describe this technique with the help of a simple example.

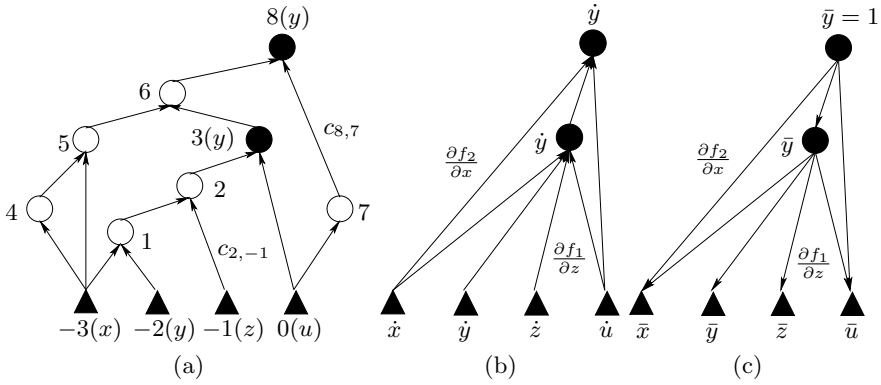


Fig. 1. Linearization and Preaccumulation.

For a given argument $\mathbf{x}$ the computation of F can be regarded as a sequence of assignments $(f_1, f_2, \ldots)$, for example,

$$f_1 : \quad y = x * y * z/u \tag{1}$$

$$f_2 : \quad y = \sin(x) * x * y * \cos(u)\ . \tag{2}$$

As in the classical case [530], we decompose each of these assignments into a list of assignments of the results of the *elemental* functions (arithmetic operators and intrinsic functions) to unique intermediate variables. For example, we set

$$v_{-3} = x;\ v_{-2} = y;\ v_{-1} = z;\ v_0 = u$$

to decompose f_1 into

$$v_1 = v_{-3} * v_{-2};\ v_2 = v_1 * v_{-1};\ v_3 = v_2/v_0$$

and f_2 into

$$v_4 = \sin(v_{-3});\ v_5 = v_4 * v_{-3};\ v_6 = v_5 * v_3;\ v_7 = \cos(v_0);\ v_8 = v_6 * v_7\ .$$

This representation induces a computational graph $\mathbf{G} = (V, E)$ (see Fig. 1(a)) with vertices V for each of the variables v_j, $j = 1-n, \ldots, q$, and edges E indicating

direct dependences between an argument and the result of an elemental function. Linearization is performed by introducing new statements for computing the nonzero partial derivatives $c_{j,i}$ for $i, j = 1-n, \ldots, q$, $i < j$, and inserting them in front of each elemental assignment. These values can be associated with the edges in $\mathbf{G}$. For example, $c_{2,-1} = v_1$ and $c_{8,7} = v_6$.

Based on the linearized version of $\mathbf{G}$ directional derivatives $F'(\mathbf{x}) \cdot \dot{\mathbf{x}}$ and adjoints $(F'(\mathbf{x}))^T \cdot \bar{\mathbf{y}}$ can be computed by the well-known recurrences that are given by (3) and (4), respectively. We present the nonincremental version of these equations in terms of the computational graph.

$$\dot{v}_j = \sum_{i:(i,j)\in E} c_{j,i} \cdot \dot{v}_i \ , \quad j = 1, \ldots, q \quad \text{(forward mode)} \tag{3}$$

$$\bar{v}_i = \sum_{j:(i,j)\in E} c_{j,i} \cdot \bar{v}_j, \quad j = q, \ldots, 1-n \quad \text{(reverse mode.)} \tag{4}$$

To avoid computational overhead the vector modes of forward and reverse propagate vectors of length l instead of the scalar quantities $\dot{v}_j$ and $\bar{v}_i$ if several products with the Jacobian or its transposed are required. The number of scalar multiplications can be assumed as an approximate measure of complexity, and it is equal to $l \cdot |E|$ in both cases.

The number of edges $|E|$ can be decreased by preaccumulating the local gradients f'_k for each assignment f_k in the original program. The AD software tool ADIFOR [57] uses *statement-level reverse mode* for this purpose. An accumulation algorithm that minimizes the number of scalar floating-point operations by computing an optimal vertex elimination sequence [232,399] in the corresponding linearized computational graph has been proposed in [396]. This algorithm is implemented in the AD compiler as described in [401]. The reduced graph $\tilde{\mathbf{G}} = (\tilde{V}, \tilde{E})$ is such that $|\tilde{E}| \leq |E|$, which results in a decreased complexity of the derivative propagation both in forward and reverse mode. For example, the local gradients for (1) and (2) are accumulated as

$$c_{3,1} = c_{3,2} \cdot c_{2,1}; \quad f'_1 = \begin{pmatrix} c_{3,1} \cdot c_{1,-3} \\ c_{3,1} \cdot c_{1,-2} \\ c_{3,2} \cdot c_{2,-1} \\ c_{3,0} \end{pmatrix} \tag{5}$$

$$c_{5,-3} += c_{5,4} \cdot c_{4,-3}; \quad c_{8,5} = c_{8,6} \cdot c_{6,5}; \quad f'_2 = \begin{pmatrix} c_{8,5} \cdot c_{5,-3} \\ c_{8,6} \cdot c_{6,3} \\ c_{8,7} \cdot c_{7,0} \end{pmatrix} \ . \tag{6}$$

The number of edges is reduced from 14 to 7 at the one-time cost of 9 multiplications (4 for f'_1 and 5 for f'_2). A reduction in the overall operation count can be observed for $l > 1$. The reduced linearized computational graph is used by the forward and reverse modes of the AD compiler as visualized in Fig. 1 (b) and Fig. 1 (c), respectively. It is built and stored explicitly for use in reverse mode as the values associated with the edges are accessed in reverse order, indicated by the reversed orientation of the edges in Fig. 1 (c). The integration of these ideas into the NAGWare Fortran 95 compiler is discussed in the following section.

4 Putting AD into the Compiler

The Fortran module `ACTIVE_MODULE` provides a number of support routines and the type definition for the active data type that is used internally, as well as for the tape to store local derivative information during the forward sweep. All semantic transformations of the original code are restricted to the active section marked by `DIFFERENTIATE` and `END DIFFERENTIATE`. The mode of differentiation is passed as an argument to `DIFFERENTIATE`. Currently, the user can choose between the forward (`AD_FORWARD`) and reverse (`AD_REVERSE`) modes of AD. Independent and dependent variables are specified by the corresponding keywords. One use of `INDEPENDENT` or `DEPENDENT` is required per independent or dependent program variable, respectively. Seeding of the adjoint component of `f` is performed automatically. To exploit sparsity within the Jacobian for arbitrary vector functions, the seed matrix can be provided as an argument to either the `INDEPENDENT` (in forward mode) or the `DEPENDENT` statements. An example is discussed in Sect. 5. The computation inside the active section can be arbitrarily complex.[3] At compile time the program is transformed into derivative code that generates a specific representation of the linearized computational graph $\mathbf{G}$. The keyword `DEPENDENT` triggers the interpretation of $\mathbf{G}$ to compute the gradient of `f` with respect to `x`. The result is stored in the variable `grad`.

Our reverse mode uses preaccumulation to generate efficient code for the computation of local gradients for all active assignments (see [256] for a discussion of activity) as described in the previous section. The linearized computational graphs of these assignments are implemented as specially structured extended Jacobians that are preaccumulated as described in [401].

For given inputs the program to be differentiated in reverse mode can be regarded as a sequence of n_s scalar assignments

$$s_l : \; v_{j(l)} = f_l(v_{i_1(l)}, \ldots, v_{i_{r(l)}(l)}) \; , \tag{7}$$

for $l = 1, \ldots, n_s$. The indexes of the local arguments on the right-hand side and of the result on the left-hand side according to the enumeration schema introduced in the previous section depend on the statement index, as does the number of local arguments $r = r(l)$ on the right-hand side. For example, in

```
do i=1,ub
   y(i)=sin(x(i))
end do
```

we have for `ub` $= 2$

$$\begin{aligned} s_1 &: \; v_1 = \sin(v_{-1}); \quad \text{that is } j(1) = 1 \text{ and } i_1(1) = -1; \\ s_2 &: \; v_2 = \sin(v_0); \quad \text{that is } j(2) = 2 \text{ and } i_1(2) = 0; \end{aligned}$$

Moreover, $r(1) = r(2) = 1$. As before, we set $v_{-1} =$ `x(1)`, $v_0 =$ `x(2)`, and `y(1)` $= v_1$, `y(2)` $= v_2$.

[3] Due to the academic nature of the project the main focus is on the presentation of proof-of-concept implementations. Various restrictions are imposed with regard to language coverage that can be overcome by some additional programming effort.

We assume that code for computing the local gradients f_l' can be generated by preaccumulation. The set of scalar active variables[4] is given as a subset of $\{v_k, k = 1-n, \ldots, q\}$ consisting of those v_k that appear either as arguments or as results of one of the statements. These variables define the address space of the program that evaluates F at the given argument. The dependent variables y_k, $k = 1, \ldots, m$, are a subset of those variables that appear on the left-hand side of some statement. The independent variables x_k, $k = 1, \ldots, n$, are a subset of all variables that appear on the right-hand side of some statement, such that no preceding statement has this variable on its left-hand side. Every scalar active variable v_k is uniquely identified by $k \equiv Id(v_k)$.

Algorithm 1 is used to generate a representation of the linearized computational graph as a tape T. A single entry $T(i)$ consist of four elements. For a given gradient of a scalar assignment s_l as in (7) we create $r = r(l)$ tape entries. The unique identifier of the variable on the left-hand side is stored in $T(i).y$. For a variable on the right-hand side the identifier is stored in $T(i).x$. The corresponding entry of the gradient is stored in $T(i).g$.[5] Moreover, a Boolean flag $T(i).f$ indicates whether the current entry is the first one[6] for a given statement. Its default value is `false`.

Algorithm 1 (Generation of the Tape)

$c := 0$
For $l := 1, \ldots, n_s$ **Do**
  compute the local gradient $f_l' \equiv (f_1'^l, \ldots, f_{r(l)}'^l)^T := \left(\frac{\partial v_{j(l)}}{\partial v_{i_1(l)}}, \ldots, \frac{\partial v_{j(l)}}{\partial v_{i_{r(l)}(l)}} \right)$
  using the code that resulted from preaccumulation;
  $T(c+1).f :=$ `true`
  For $k := 1, \ldots, r(l)$ **Do**
    $T(c+k).x := Id(v_{i_k(l)})$
    $T(c+k).y := Id(v_{j(l)})$
    $T(c+k).g := f_k'^l$
  Enddo
  $c := c + r(l)$
Enddo

A new tape is generated if the values of the inputs change. The possible reuse of an existing tape will be considered in the future.

The reverse sweep operates on a virtual adjoint address space in the form of an array whose size is equal to the number of active variables in the reduced linearized computational graph. The adjoints of all dependent variables are initialized. The adjoints of the remaining active variables are assumed to be equal to zero. For each tape entry the adjoint of the variable on the right-hand side is incremented by the product of the adjoint of the variable on the left-hand side with the corresponding gradient entry. A temporary adjoint variable $\bar{a}$ (initially equal to zero) is used if the same variable occurs on both the left-hand and right-hand sides. When the first

[4] Arrays are handled on a per element basis.

[5] There is no aliasing problem here because the identifiers are assigned to active variables at runtime.

[6] in the order of the arguments on the right-hand side. That is, $T(i).f =$ `true` $\Rightarrow$ $T(i).x = i_1(l)$.

entry of a statement is reached, we set the adjoint of the variable on the left-hand side to zero or to the value of the temporary adjoint, if applicable. This procedure is formalized in Algorithm 2. The counter c points to the last entry in the tape.

Algorithm 2 (Interpretation of the Tape)

initialize adjoints; alias = `false`
For $i := c, \ldots, 1$ **Do**
  If $T(i).x = T(i).y$ **Then**
    $alias :=$ `true`
    $\bar{a} := \bar{a} + \bar{v}_{T(i).y} \cdot T(i).g$
  Else
    $\bar{v}_{T(i).x} := \bar{v}_{T(i).x} + \bar{v}_{T(i).y} \cdot T(i).g$
  Endif
  If $T(i).f$ **Then**
    If $alias$ **Then**
      $alias :=$ `false`
      $\bar{v}_{T(i).y} := \bar{a}$
      $\bar{a} := 0$
    Else
      $\bar{v}_{T(i).y} := 0$
    Endif
  Endif
Enddo

5 Case Study: Seeding in Forward and Reverse Mode

The intention of this case study is to present the convenience and flexibility of the new interface provided by the compiler to support the computation of both directional derivatives and adjoints. We compute the $n \times n$ Jacobian ($n = nx \cdot ny$) of the solid fuel ignition problem from the MINPACK-2 test problem collection [17] in both forward and reverse modes. Compression techniques are used to exploit the regular sparsity pattern resulting from the use of a five-point stencil for discretization. A driver program calls the MINPACK routine `dsfifj` inside the active section. Since we have equal numbers of independent and dependent variables we expect reverse mode to be slower than forward mode due to the interpretive overhead during the reverse sweep through the tape. This expectation is met by the actual run times.

5.1 Seeding in Forward Mode

A suitable {0,1}-seed matrix `seed` can be computed by coloring the vertices in the column incidence graph as described in [141]. This matrix is passed as a parameter to the `INDEPENDENT` statement. Internally, the computation is performed with directional derivative vectors of length l, where l is the number of columns in `seed`. The compressed Jacobian is stored in `y_der`. The named parameter `DERIVATIVE` is provided by the `DEPENDENT` statement for this purpose.

```
PROGRAM DRIVER_SFI
  USE ACTIVE_MODULE
  ...

  DOUBLE PRECISION, DIMENSION(:), ALLOCATABLE :: x,y
  INTEGER :: nx, ny
  INTEGER, DIMENSION(:,:), ALLOCATABLE :: seed
  DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE :: y_der
  ...

! get values for nx, ny; allocate x,y
! get values for x
! determine structure of seed by coloring
! allocate and initialize seed
  ...

  DIFFERENTIATE(AD_FORWARD)
  INDEPENDENT(x,SEED=seed)
  CALL dsfifj(nx,ny,x,y,...)
  DEPENDENT(y,DERIVATIVE=y_der)
  END DIFFERENTIATE
  ...

CONTAINS
  include "dsfifj.f"

END PROGRAM DRIVER_SFI
```

The various steps that need to be performed ahead of the active section are described by comments. Following the active section the compressed Jacobian `y_der` is available for further use in the context of the numerical algorithm.

5.2 Seeding in Reverse Mode

Analogous to forward mode the {0,1}-seed matrix `seed` can be computed by coloring the vertices in the row incidence graph for a given sparsity pattern of the Jacobian. `seed` is passed as a parameter to the `DEPENDENT` statement. The adjoint computation is performed with vectors of length l, where l is the number of rows in `seed`. The compressed transposed Jacobian is stored in `x_adj`.

```
  ...

  INTEGER, DIMENSION(:,:), ALLOCATABLE :: seed
  DOUBLE PRECISION, DIMENSION(:,:), ALLOCATABLE :: x_adj
  ...

  DIFFERENTIATE(AD_REVERSE)
  INDEPENDENT(x,ADJOINT=x_adj)
  CALL dsfdfi(nx,ny,x,y,...)
```

```
DEPENDENT(y,SEED=seed)
END DIFFERENTIATE
...
```

Due to symmetry the same seed matrix as in forward mode can be used to compute the compressed transposed Jacobian. Notice that the user interface is symmetric in the sense of forward and reverse modes being symmetric. The design of the interface contributes to the intuition of potential users of this new compiler feature.

5.3 Numerical Results

We ran extensive tests for varying problem sizes on a Athlon-XP2600 computer with 2GB of virtual memory (1GB physical) recording the following characteristic values:

Number of independent variables: $\mathtt{nx} = \mathtt{ny} = \sqrt{n}$;
Number of columns/rows in seed matrix: l;
User time: user;
System time: system;
Cpu usage in percent: cpu;
Number of main memory misses (paging): F+;
Number of cache misses: F- .

In dense forward (F) and reverse (R) modes we are able to compute the 8100×8100 Jacobian resulting from $\mathtt{nx} = \mathtt{ny} = 90$ before running out of virtual memory. This is due to the cumulative memory requirement of the derivative components of all active variables. As expected the use of reverse mode does not lead to run time savings. Paging increases the overall run time considerably.

	$\sqrt{n}$	l	user	sys	real	cpu	F+	F-
F	60	3600	1.76	0.37	2.16	98.00%	156	55340
R	60	3600	4.91	0.56	6.75	81.00%	153	76789
F	80	6400	9.07	1.25	11.93	86.33%	156	174807
R	80	6400	25.55	1.98	35.24	79.67%	767	244607
F	90	8100	14.62	2.53	23.37	73.00%	367	279224
R	90	8100	41.83	6.69	328.57	14.67%	41660	522964

Seeding allows us to compute much larger problems going up to a problem size of 1,690,000 independent variables. The reverse mode run took 1.75GB of memory. Again, paging leads to a large increase of the run time in this case. The number of active variables grows to 3,380,014. A total of 37,174,808 partial derivatives are stored on the tape whose final size is 37,180,000 allocated in chunks of 10,000. The tape management overhead can potentially be decreased by increasing the chunk size.

	$\sqrt{n}$	l	user	sys	real	cpu	F+	F-
F	900	5	2.26	0.41	2.68	99.00%	155	57782
R	900	5	4.48	1.05	5.53	99.00%	153	145846
F	1100	5	3.38	0.59	3.97	99.00%	155	86303
R	1100	5	6.82	1.45	8.29	98.67%	153	218187
F	1300	5	4.67	0.84	5.51	99.00%	155	120527
R	1300	5	9.56	4.09	35.52	38.00%	7644	399387

So far, we have not been competing with established source transformation tools for AD of Fortran programs such as ADIFOR [96], TAPENADE [257], or TAF [209]. At the current state we do not expect to generate more efficient adjoint code due to the lack of data-flow analysis and explicit reversal of the flow of control. Still we consider the obvious ease-of-use of the AD compiler as a major advantage. Improvements in efficiency will be addressed in the future as outlined below.

6 Summary, Conclusion, and Outlook

The source transformation AD algorithm that is part of a research prototype of the NAGWare Fortran 95 compiler uses statement-level preaccumulation of local gradients in connection with an overall forward mode strategy. A new version of the compiler has been described in this paper that makes reverse mode AD available for the efficient computation of adjoints. The local gradients of all scalar assignments are written to a tape that is interpreted as part of the adjoint computation. A highly convenient and intuitive user interface has been designed allowing for the computation of matrix products involving both the Jacobian and its transpose. The runtime behavior of the adjoint code compared with the tangent-linear code is as expected.

Our work was motivated by optimization problems that require gradients with respect to a large number of independent variables. Forward mode turned out to be infeasible for practically relevant problem sizes. The applicability of the current solution is limited by the size of the tape with respect to the memory resources. Some problems may result in tapes that are simply too large. Our plans involve pushing the limits by generating adjoint code that uses real instead of virtual memory (no tape) as the result of reversing the flow of control as in [402]. Further significant improvements can be expected from the planned implementation of checkpointing techniques.

Acknowledgments

This work would have been impossible without the Numerical Algorithms Group's support while setting up the interface between the compiler and the automatic differentiation algorithms. In particular Malcolm Cohen made considerable contributions to the underlying software infrastructure.

The flexibility of the University of Hertfordshire in Hatfield, UK, regarding the management of the project has been another important factor.

Last but not least, we would like to thank the referees for many useful remarks that contributed to the improved quality of the final version of this paper.

Extension of TAPENADE toward Fortran 95

Valérie Pascual and Laurent Hascoët

INRIA, TROPICS team, Sophia-Antipolis, France
{Valerie.Pascual,Laurent.Hascoet}@sophia.inria.fr

Summary. We present extensions to the automatic differentiation tool TAPENADE to increase coverage of the Fortran 95 language. We show how the existing architecture of the tool, with a language independent kernel and separate front-ends and back-ends, made it easier to deal with new syntactic forms and new control structures. However, several new features of Fortran 95 required us to make important choices and improvements in TAPENADE. We present these features, sorted into four categories: about the top-level structure of nested modules, subprograms, and interfaces; about structured data types; about overloading capabilities; and about array features. For each category, we discuss the choices made, and we illustrate their impact on small Fortran 95 examples. Dealing with pointers and dynamic memory allocation is delayed until extension to C begins. We consider this extension to Fortran 95 as a first step towards object-oriented languages.

Key words: TAPENADE, Fortran 95, program transformation

1 Introduction

We present extensions to the automatic differentiation [136,225] tool TAPENADE [256, 257] to increase coverage of the Fortran 95 language [365]. Other AD tools already took this direction, such as TAF [210,211,293] and the NAGWare AD-enabled Fortran 95 compiler [400]. ADIFOR [96] has already considered the differentiation of code with structured data types.

Given the source of an original program, plus a description of which output variables must be differentiated, and with respect to which input variables, TAPENADE produces a new source program that computes the requested derivatives.

We recall the internal architecture of TAPENADE [257] in Fig. 1, with a central module for program analysis and transformation, surrounded by language-specific front-ends and back-ends. This allows the central module to forget about mostly syntactic details of the analyzed language and to concentrate on the language's semantic constructs. To this end, TAPENADE defines an internal abstract language, called IL, which can represent all constructs of classical imperative languages. In particular, extension to Fortran 95 drove us to add several new constructs into IL.

Some of these constructs will thus be directly available when we start extension to C. Furthermore, programs are internally represented as Call Graphs, Control Flow Graphs [3], and, only at the deepest level of individual statements, Syntax Trees. This yields a general representation for all control structures.

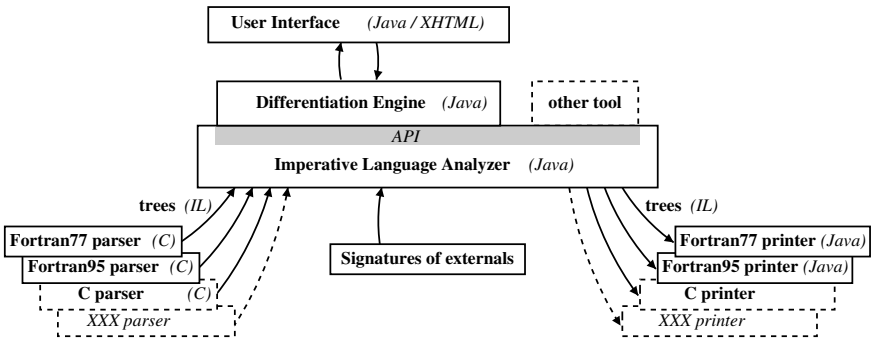


Fig. 1. Architecture of TAPENADE.

Concerning Fortran 95 syntax, everything is taken care of by a specific new parser (front-end) and pretty-printer (back-end). A major difference compared to Fortran 77 is the free format source form, where statements may start in any column. Our new Fortran 95 parser accepts programs that combine the old fixed format and the free format, and the back-end can regenerate programs using both formats. Thanks to the internal representation as Control Flow Graphs, new constructs such as the `SELECT CASE` were easily added and treated by the differentiation engine as any other flow of control. The same remark applies to the new `CYCLE` and `EXIT` constructs.

In this paper, we focus on the features of Fortran 95 that required us to make important choices and improvements in TAPENADE. We put these features into four categories: Sect. 2 deals with the nesting of modules, subprograms, and interfaces, Sect. 3 deals with the treatment of structured types, called "derived" types in Fortran 95, Sect. 4 deals with the overloading capabilities, and Sect. 5 deals with array features. Sect. 6 concludes with the soon to come pointer analysis, and the more distant extension to object-oriented programming.

2 Nesting of Modules and Subprograms

The internal representation has to be extended to capture the new top-level nesting of procedures, with modules, internal/external subprograms, and interfaces. In comparison, the structure of Fortran 77 was flat, apart from statement functions and internal subprograms in some dialects.

We choose to introduce an internal tree representation of module nesting, in addition to the existing Call Graph. Each node stands for one "unit," i.e. subprogram or module, and holds the list of its enclosed units. In particular, the regenerated differentiated program must comply with this unit tree structure, so that this program is stand-alone and can be compiled directly. Each unit defines two symbol tables,

for the public and private symbols (i.e. arguments, variables, subprograms, types, etc). The private symbol table naturally inherits from the public one. Symbol table nesting already existed in TAPENADE for scoping. The `USE` statement just states that a unit has access to the public symbol table of a module.

Classically, program analyses and transformations need to run in an appropriate order on the subprograms, depending on the Call Graph. The novelty is that this order now must take into account the `USE` of modules. Moreover, recursion may introduce cycles in this dependence, so the best order can only be an approximation.

Where differentiation is concerned, the question is what must belong to a differentiated unit? When Fortran 77 was considered, differentiated symbols could be defined independently from their original symbols. Now that modules can define their own private symbols, some differentiated unit must often be declared in the same context as its original unit, i.e. must live inside the same enclosing module. In general the question is whether the differentiated object can exist independently of the original object, or must they be attached inside the same enclosing level. For example, a differentiated statement must be in the same subprogram as the original statement because they share a common control. Similarly a differentiated subprogram must be in the same module as the original if both access a private entity of this module. On the contrary, a differentiated component `x` of a derived type `T` need not be added into `T`, but can rather go into a stand-alone "differentiated" derived type `T'`, therefore saving memory space.

To illustrate, consider the source program of Fig. 2, containing a type definition, variables, and a function. The corresponding differentiated program contains the same declarations as the source program, plus the differentiated function.

3 Derived Types

Fortran 95 allows the user to define "derived" types (this name has no relation with differentiation) in order to manipulate composite objects containing several components. As we said in Sect. 2, our choice during differentiation is to define a differentiated derived type, whose components hold the derivatives of the original components. However, it happens that some variables of some derived type have only some components that are active. Then the differentiated type need not allocate space for the other components. Therefore, differentiated derived types depend on the activity pattern. On the other hand, we don't want to *specialize* too far, creating several differentiated types for a given derived type. Therefore, our choice is very similar to differentiation of subprograms with several activity patterns: there is only one differentiated type `T'` for each derived type `T`. During activity analysis, if a component `x` of some variable of type `T` can be active, the differentiated type `T'` must hold a component `x` too. The price for this non-specialization is that sometimes a differentiated variable of type `T'` will not use all of its components.

In the previous example, the differentiated type of the "vector" type is equal to the initial type as all components are active, so no differentiated type appears in the differentiated module.

In the example of Fig. 3, the "name" and "y" components of type "vector" are not active. Therefore they do not appear in the differentiated type "vector_d."

```
module example1
 implicit none
 type vector
   real :: x,y,z
 end type vector

 type(vector) :: u,v,w

contains
 function dot_prod(a,b)


  type(vector) :: a,b

  real :: dot_prod

  dot_prod = a%x*b%x + &
&       a%y*b%y + a%z*b%z




 end function dot_prod









end module
```

```
!  Generated by TAPENADE
!     Version 2.0.12
MODULE EXAMPLE1_D

  TYPE VECTOR
      REAL :: x,y,z
  END TYPE VECTOR

  TYPE(VECTOR) :: u, v, w

CONTAINS
  FUNCTION DOT_PROD_D(a, ad, &
&       b, bd, dot_prod)
    IMPLICIT NONE
    TYPE(VECTOR) :: a, b
    TYPE(VECTOR) :: ad, bd
    REAL :: dot_prod
    REAL :: dot_prod_d
    dot_prod_d = ad%x*b%x + &
&     a%x*bd%x + ad%y*b%y + &
&     a%y*bd%y + ad%z*b%z + &
&     a%z*bd%z
    dot_prod = a%x*b%x + &
&      a%y*b%y + a%z*b%z
  END FUNCTION DOT_PROD_D

  FUNCTION DOT_PROD(a, b)
    IMPLICIT NONE
    TYPE(VECTOR) :: a, b
    REAL :: dot_prod
    dot_prod = a%x*b%x + &
&      a%y*b%y + a%z*b%z
  END FUNCTION DOT_PROD
END MODULE EXAMPLE1_D
```

Fig. 2. Differentiation of nested modules and subprograms.

4 Overloading

The term overloading refers to calling different subprograms or operators by the same generic name. Whereas overloading in object-oriented languages is resolved only at run time, the limited form of overloading in Fortran 95 can be resolved statically at compile time, and therefore at differentiation time. This is done during the type-checking phase. Furthermore, Fortran 95 also allows the user to overload predefined operators such as `+`, `-`, `*`, `/`, or assignment `=`.

We must thus modify the type-checking algorithm carefully. Each use of a predefined operator or call to a subprogram is compared to available overloaded subprograms according to the arguments' types. If necessary, it is replaced by the

```
module example2
 implicit none
 type vector
   character(256) :: name
   real :: x,y,z
 end type vector

 type(vector) :: u,v,w

contains
 function test(a,b)

  type(vector) :: a,b

  real :: test
  print *, a%name, b%name
  test = a%x + b%x + u%z

 end function test

end module
```

```
!  Generated by TAPENADE
!     Version 2.0.12
MODULE EXAMPLE2_D

  TYPE VECTOR_D
      REAL :: x,z

  END TYPE VECTOR_D

  TYPE VECTOR
      CHARACTER*(256) :: name
      REAL :: x,y,z
  END TYPE VECTOR

  TYPE(VECTOR) :: u, v, w
  TYPE(VECTOR_D) :: ud

CONTAINS
  FUNCTION TEST_D(a, ad, b, &
&   bd, test)
    IMPLICIT NONE
    TYPE(VECTOR) :: a, b
    TYPE(VECTOR_D) :: ad, bd
    REAL :: test, test_d
    PRINT*, a%name, b%name
    test_d = ad%x + bd%x + ud%z
    test = a%x + b%x + u%z
  END FUNCTION TEST_D

  FUNCTION TEST(a, b)
    IMPLICIT NONE
    TYPE(VECTOR) :: a, b
    REAL :: test
    PRINT*, a%name, b%name
    test = a%x + b%x + u%z
  END FUNCTION TEST
END MODULE EXAMPLE2_D
```

Fig. 3. Differentiation of derived types.

appropriate subprogram call, and treated as such in the following differentiation phase. Therefore, at the end of the type-checking phase, overloading is completely resolved.

When differentiation is concerned, the predefined operators are treated in a very particular, built-in manner, so we must be careful not to replace these operators by ordinary subprograms calls when not necessary.

In our third example of Fig. 4, the addition of vectors is overloaded.

5 Array Features

The array programming features of Fortran 95 are represented through syntactic notations and intrinsic functions. In Fortran 95 programs, it is possible to use whole array operations. In the differentiated program, we keep this property whenever possible, and we also use array operations on differentiated arrays.

For example, for arrays A, B and scalar x, the loop

```
do i=1,N
    A(i) = 3*B(i-1) + x
end do
```

can be written equivalently as:

```
A(1:n) = 3*B(0:n-1) + x
```

Array features can be advantageous for static data flow analysis. A reset of a whole array to a constant, for example, can be easily detected, whereas the equivalent loop requires array region analysis [140] to reach the same conclusion.

When differentiation is concerned, two intrinsic array functions play a very special role: the `SUM` intrinsic and the spread operation (which is often implicit or otherwise is done by the `SPREAD` intrinsic). In particular the adjoint of a `SUM` is a spread, and vice-versa. For example the adjoint statement of

```
x = x + SUM(B(:))
```

is the following, with an implicit spread on $\overline{x}$

```
B̄(:) = B̄(:) + x̄
```

Actually these two intrinsics blend into the internal representation for partial derivatives and reappear when generating the differentiated code.

To illustrate how this is effectively performed by TAPENADE, we need a more complete example. Consider the following array assignment of a scalar value to array `A`, at indices 0 to 100 by stride of 3, where `x` is a scalar:

```
A(0:100:3) = x * SUM(B(:))
```

The elements of `A` that are not in the section `A(0:100:3)` play no role in this statement, and can therefore be neglected. In the rest of this example, we shall thus simplify the notation by writing `A` instead of `A(0:100:3)` when there is no ambiguity. This array assignment is differentiated in the reverse mode using the transposed local Jacobian:

```
module example3
 implicit none
 type vect
   real :: x,y
 end type vect
 type(vect) :: u,v,w
 interface operator (+)
  module procedure addvect
 end interface

contains
 function test(a,b)


  type(vect) :: a,b,test


  test = a + b + u



 end function test

 function addvect(a,b)


  type(vect),intent(in)::&
&           a,b
  type(vect) :: addvect

  addvect%x = a%x + b%x

  addvect%y = a%y + b%y
 end function addvect









end module
```

```
!  Generated by TAPENADE
!     Version 2.0.12
MODULE EXAMPLE3_D

  TYPE VECT
      REAL :: x,y
  END TYPE VECT
  TYPE(VECT) :: u, ud, v, w
  INTERFACE OPERATOR(+)
      MODULE PROCEDURE addvect
  END INTERFACE

CONTAINS
  FUNCTION TEST_D(a, ad, b, &
&                 bd, test)
    IMPLICIT NONE
    TYPE(VECT) :: a, ad, b, bd
    TYPE(VECT) :: test, test_d
    TYPE(VECT) :: arg1, arg1d
    arg1d = ADDVECT_D(a, ad,&
&                  b, bd, arg1)
    test_d = ADDVECT_D(arg1,&
&           arg1d, u, ud, test)
  END FUNCTION TEST_D

  FUNCTION ADDVECT_D(a, ad,&
&     b, bd, addvect)
    IMPLICIT NONE
    TYPE(VECT),INTENT(IN):: a,b
    TYPE(VECT) :: ad, bd, &
&     addvect, addvect_d
    addvect_d%x = ad%x + bd%x
    addvect%x = a%x + b%x
    addvect_d%y = ad%y + bd%y
    addvect%y = a%y + b%y
  END FUNCTION ADDVECT_D

  FUNCTION TEST(a, b)
    ...
  END FUNCTION TEST
  FUNCTION ADDVECT(a, b)
    ...
  END FUNCTION ADDVECT
END MODULE EXAMPLE3_D
```

Fig. 4. Differentiation of overloaded procedures and operators.

$$\left\{\frac{\partial[\mathtt{A},\mathtt{x},\mathtt{B}]_{out}}{\partial[\mathtt{A},\mathtt{x},\mathtt{B}]_{in}}\right\}^* = \begin{pmatrix} 0 & \mathtt{SUM(B(:))} & \mathtt{x} \\ 0 & Id & 0 \\ 0 & 0 & Id \end{pmatrix}^* = \begin{pmatrix} 0 & 0 & 0 \\ \mathtt{SUM(B(:))} & Id & 0 \\ \mathtt{x} & 0 & Id \end{pmatrix}$$

which is a block matrix, whose structure is emphasized in Fig. 5, using rectangles to represent blocks. Notice that the four larger blocks are actually rectangular blocks, whereas the other blocks are two row matrices, two column matrices, and a 1x1 block. The *Id* blocks correspond to the identity matrix. The `SUM(B(:))` block is a

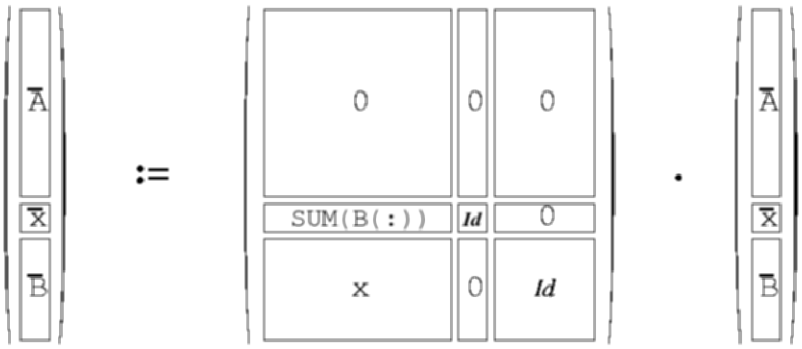


Fig. 5. Transposed Jacobian block matrix for an array assignment.

row matrix where each entry is equal to `SUM(B(:))`. $\overline{\mathtt{A}}$ is a column matrix whose values are $\overline{\mathtt{A}}$`(0:100:3)`. Figure 5 shows the vector assignment that the differentiated assignments must implement.

The shape of the blocks determines where `SUM`'s and spreads must appear in the differentiated array assignments. For example, the assignment that updates $\overline{\mathtt{x}}$ implements the product of the `SUM(B(:))` row vector by the $\overline{\mathtt{A}}$ column vector, yielding the following reduction:

```
x̄ = x̄ + SUM(B(:))*SUM(Ā(0:100:3))
```

The complete set of differentiated assignments is:

```
x̄ = x̄ + SUM(B(:))*SUM(Ā(0:100:3))
B̄(:)= B̄(:)+ x*SUM(Ā(0:100:3))
Ā(0:100:3) = 0.0
```

Only `SUM` and spread blend with the local Jacobian notation as described above. All the other array intrinsics are treated like black-box routines, whose differentiation is given to TAPENADE in special library files.

6 Conclusion

We have described extensions of the AD tool TAPENADE towards full coverage of Fortran 95. This extension is made easier by the fact that Fortran 95 derives from Fortran 77, and also by TAPENADE's internal representation of programs independent

from the language. For example, the notion of structured data types was already in TAPENADE even before extension to Fortran 95 was considered. Obviously, all constructs that exist both in Fortran 77 and Fortran 95 required no new development at all. The features of Fortran 95 that called for new developments are those which had been overlooked or not fully tested because Fortran 77 did not use them. Similarly, TAPENADE already handles nested scoping blocks inside a procedure, and this feature will be available immediately for the extension to C, just like structured types.

One major feature of Fortran 95 is still omitted from the present work: pointers and dynamic allocation. The internal representation already captures pointers, but the program analyses do not take them into account. This is the next development on our list. We plan to share this with the soon-to-come extension of TAPENADE to C. In addition to the development of a classical pointer analysis, this will require a conceptual study of a differentiation model for pointers, especially in the reverse mode of AD. Differentiation of dynamic allocation and pointers is only problematic for reverse mode AD. Tangent AD, as well as reverse AD implemented through a tape as in ADOL-C [229] handle these features in a straightforward fashion.

In the long run, we shall consider extending TAPENADE to the object programming concepts from C++ or JAVA. This will introduce dynamic overloading, which is still a challenge for automatic differentiation. For example, the activity pattern of the actual parameters of a given call may contribute to the activity pattern of several subprograms. However, the module nesting management developed here is a major step towards handling the global structure of object-oriented programs.

There are two ways you can use TAPENADE. It can be used as a server at the url

`http://tapenade.inria.fr:8080/tapenade/index.jsp`

Alternatively, it can be downloaded from

`ftp://ftp-sop.inria.fr/tropics/tapenade`

and locally installed. In that case it is run by a simple command line, which can be included into a Makefile. TAPENADE also provides a user-interface to visualize the results in a HTML browser. An on-line documentation is available at the url

`http://www.inria.fr/tropics`

We encourage you to use TAPENADE and to report any problems, therefore helping us making it an industrial quality AD tool for Fortran 95.

A Macro Language for Derivative Definition in ADiMat

Christian H. Bischof, H. Martin Bücker, and Andre Vehreschild

Institute for Scientific Computing, RWTH Aachen University, Aachen, Germany
{bischof,buecker,vehreschild}@sc.rwth-aachen.de

Summary. Any automatic differentiation tool for MATLAB needs to cope with the large number of functions provided by the toolboxes. For many of these functions, derivatives have to be defined. A powerful macro language for the derivative definition, embedded in the source transformation tool ADiMat, is introduced. The macro language consists of a part where the signature of a function is matched and another part specifying the derivative of that function. Several examples illustrate the expressiveness and use of the macro language. A subset of the macro language is available to the user of ADiMat to improve the performance of the generated derivative code by exploiting high-level structure.

Key words: MATLAB, ADiMat, automatic differentiation, macro language, derivative definition

1 Introduction

MATLAB[1] is a well-known high-level programming language geared towards scientific computations. Its attractiveness stems to a considerable degree also from the large set of powerful functions provided by the MATLAB toolboxes. Such functions are called toolbox functions hereafter. As a first step to handle the numerous toolboxes, we concentrate on the standard MATLAB toolbox in the present study. Examples of toolbox functions from the standard MATLAB toolbox include `deal()` and `mesh()`. For the sake of simplicity, the term "toolbox function" is also used to refer to functions built into the MATLAB core. That is, operators such as $+$, $*$ or $\backslash$ and also functions like `sin()`, `norm()`, `fft()` are called toolbox functions in this article. In the context of automatic differentiation (AD) [42, 136, 225, 227], derivatives of many toolbox functions are needed to transform MATLAB code in the same way as derivatives of all elemental functions are needed to transform Fortran code. Since derivatives of a few hundreds toolbox functions are required, the specification of derivatives for these toolbox functions is of crucial importance when implementing an AD tool for MATLAB [51, 130, 178, 180, 467, 521]. To this end, we

[1] MATLAB is a registered trademark of The Mathworks, Inc.

introduce a macro language and its embedding in the source transformation tool ADiMat [51]. The macro language is not only used in the development of ADiMat, the user of ADiMat can also access a restricted set of the same macro language to specify derivatives of user-defined functions, offering a powerful mechanism to potentially improve the performance of AD-generated code by exploiting the structure of the given code.

The outline of this paper is as follows. In Sect. 2, we describe elements of MATLAB relevant for the design of an AD tool based on the source transformation approach. In Sect. 3, the macro language for concisely defining the signatures of functions is presented. This section also describes how to define the derivative of a function. An example is given in Sect. 4 where the derivative of a user-defined function is assumed to be known. Then, using the macro language, ADiMat allows the user to explicitly specify this derivative, thus exploiting the structure of the given code.

2 MATLAB in the Context of an AD Tool

MATLAB is a high-level language containing arithmetic statements, control flow structures, modularization, and object oriented paradigms. The AD tool ADiMat applies the source transformation approach to MATLAB codes. It keeps high-level operations such as matrix-vector products intact and augments the code with additional high-level operations. For example, the code fragment `x = A * b`, where `A` is a matrix of size $m \times n$ and `b` is a vector of size n, is transformed to

```
g_x = g_A * b + A * g_b;
x = A * b;
```

where `g_A` and `g_b` are the derivative objects associated with `A` and `b`.

In addition to these high-level operations, MATLAB consists of a rich set of toolbox functions, many of which are not written in MATLAB but in different languages such as C or Fortran. There are approximately 1200 identifiers for functions and variables in the standard MATLAB toolbox. Around 300 toolbox functions need at least one derivative definition.

Because of polymorphism, a toolbox function may need more than one derivative definition. In MATLAB the following polymorphic features are among the most frequently used. In this article, the term parameter is used to denote formal parameters, whereas the term argument is used for actual parameters.

- Number of arguments and results: a MATLAB function may be called with a varying number of results and arguments. Parameters that are not initialized by the call may be set to default values, or are optional in the body of the function. For example, the function `norm(x,p)`, which computes the `p`-norm of vector or matrix `x`, may be called with one or two arguments. The second parameter `p`, if omitted, is set to the default value of `2` to compute the Euclidean norm.
- Type of arguments: depending on the type of an argument a MATLAB function may behave differently. For example, if `x` is a vector the function `norm(x)` evaluates the standard formula of the Euclidean norm. However, if `x` is a matrix, the function `norm(x)` computes the Euclidean norm by applying the singular value decomposition.

- Value of arguments: functions may be parameterized by flags or values. For example, if `x` is a matrix, `norm(x, inf)` computes the infinity norm, whereas `norm(x, 1)` computes the largest column sum.

From an AD point of view, differentiation in MATLAB is complicated by the fact that many toolbox functions are written in languages other than MATLAB. Several ways to provide the derivatives of those toolbox functions are feasible:

- Divided-difference approximations may be computed. This approach is always feasible, but suffers from well-known accuracy shortcomings.
- An alternative is to apply an AD tool for a different language such as Fortran or C if the source code is available. Here, identifying the active variables, setting up the AD process, the compilation of the derivative code, and the generation of the mex-interface (see `help mex` in MATLAB) currently is not automated.
- Another approach is to rewrite the function in MATLAB and differentiate it with ADiMat. However, this approach is laborious and error-prone. Also, writing MATLAB code for functions specifically tuned for high-performance is a potential source of inefficiency.

ADiMat offers an alternative way for the specification of derivatives by providing a macro language that allows the definition of differentiation rules at a high level.

3 The Macro Language

The definition of a derivative of a toolbox function is divided into two parts. The first part specifies the "signature" of the function to be differentiated, and the second is the "action part."

3.1 Signature Definition

The signature classifies the use of an identifier contained in a toolbox as a function or variable. To this end, three keywords are used: (a) the definition of a variable or constant is started with the keyword `BVAR`; (b) a command or script is preceded by `BCOMMAND`; (c) for a function, the keyword `BMFUNC` is used.

In Table 1 the grammar of the "signature part" is described in enhanced Backus-Naur-Form as used in [242]. The terminals used in the grammar are double-quoted strings such as `"BMFUNC"`, and tokens are denoted by all capitals, such as `ID` or `STRING`. Non-terminals are written in lowercase like `variable_spec` or `num_param`. A question mark denotes an optional part. An asterisk matches a rule zero or multiple times. The non-terminal `action` appearing in the fourth rule of Table 1 will be described in the next subsection.

For example, the signature `BVAR pi` defines the constant π. The rule `BVAR` does not allow a specification of any action. The action of `BVAR` is hardwired to be differentiated to a zero derivative object whose creation is omitted most of the time for performance reasons.

Consider a second example. The signature

```
BMFUNC $$ = sin($1) ...
```

Table 1. EBNF Grammar of signatures

```
kindspec := variable_spec | command_spec | function_spec;
variable_spec := "BVAR" identifier;
command_spec := "BCOMMAND" ID ( parameters )?;
function_spec := "BMFUNC" signature action;
signature := ( results )? ID ( parameters )?;
parameters := "(" ( num_param ( "," num_param )* )?
   ( ( "," )? "$#" ( type )? )? ")";
results := ( ( single_result | multiple_results ) "=" )?;
single_result := "$$" ( type )?;
multiple_results := "[" ( num_res ( "," num_res )* )?
   ( ( "," )? "$$#" ( type )? )? "]";
num_res := "$$" INT ( type )?;
num_param := "$" INT ( "=" default )? ( type )? | default ;
identifier := ID ( type )?;
type := ":" ID ;
default := ID | FLOAT | STRING;
```

gives a derivative definition for the sine function, where the part related to `action` is not shown here for simplicity. This definition of the signature tells ADiMat that a toolbox function of MATLAB with identifier `sin` exists, which needs exactly one argument `$1` and returns exactly one result `$$`.

An example of a more complex signature is given by

```
BMFUNC $$ = norm($1, $2) ...
```

representing the function `norm`. The definition of this signature declares the function `norm` taking two arguments and returning one result. The second argument of the function `norm` is optional. If omitted the default value of `2.0` is taken. This can be expressed by the signature

```
BMFUNC $$ = norm($1, $2 = 2.0) ... ,
```

eliminating the need to specify a second rule. More precisely, a rule containing just one parameter and using the default value in the action is not necessary.

Many functions in MATLAB use a variable number of parameters and results. For example, the function `deal` assigns its inputs to its outputs. If the number of its arguments matches the number of its results, the first argument is assigned to the first result, the second argument to second result and so on. If `deal` is called with one argument only, then each result is assigned a copy of the argument. The function `deal` uses the feature `varargin` and `varargout`, handling a variable number of input and output arguments, respectively. This feature is translated to the macro language of ADiMat using `$#` for a variable number of input arguments and `$$#` for a variable number of results. Both symbols are only accepted as the last or only entry in a list of input or output parameters. The signature of `deal` may be given by

```
BMFUNC [$$#] = deal($#) ...
```

defining the syntax as described above.

Alternatively, a signature definition may be partially specialized That is, some placeholders of parameters (e.g. `$1`, `$2`) may be replaced by constant values, e.g.,

```
BMFUNC $$ = norm($1, inf) ... .
```

A shortcut exists for functions that do not return a result. Such a function is called a command, and the `BMFUNC` is replaced by `BCOMMAND`. The command will never be differentiated, i.e., it is ignored as described in the next subsection. An example of a command requiring at least one argument is

```
BCOMMAND plot($1, $#) .
```

3.2 The Actions

The *action* of a derivative definition is an essential part in creating derivatives for toolbox functions. Here, three different actions are described as shown in Table 2. The terminal `MATLAB_EXPRESSION_WITH_$-SYMBOLS` matches a MATLAB code fragment containing a so-called placeholder at any position. In the sequel, a placeholder is any string starting with a dollar `$`. These placeholders are replaced by ADiMat with arguments and their derivatives.

Table 2. EBNF Grammar of actions

```
action := "IGNORE" | "NODIFF" | diffto;
diffto := "DIFFTO" MATLAB_EXPRESSION_WITH_$-SYMBOLS;
```

Action: `IGNORE`

The action `IGNORE` defines a toolbox function to be ignored during differentiation. Whenever ADiMat encounters an identifier that is declared to be ignored, then it does not generate any derivative code for the subexpression of the call of the toolbox function. Also, this toolbox function is ignored in the activity analysis that determines which variables need to have derivatives associated with them. For example, the complete specification for defining the toolbox function `size` is given by

```
BMFUNC [$$#] = size($1, $#) IGNORE .
```

Action: `NODIFF`

For a large set of toolbox functions, in particular for those that are most frequently used, derivatives are specified in ADiMat. However, there are certain toolbox functions whose differentiation is currently postponed for reasons of simplicity. The action `NODIFF` is designed to indicate that the current version of ADiMat does not differentiate a specific toolbox function yet. The activity analysis for a toolbox function specified as `NODIFF` is performed as usual. If during the step of augmentation a toolbox function declared as `NODIFF` is encountered, then a default error message is issued on the standard output and a comment is inserted into the output code. The derivative code generated by ADiMat is probably incorrect. As an example, consider the statement `z = eval(str)`, where `str` is a string assigned the value `'sin(x)'` at runtime. It is hard for an AD-tool to recognize that $z = \sin(x)$ within the current context. Therefore, the specification of `NODIFF` is appropriate:

```
BMFUNC [$$#] = eval($1, $#) NODIFF .
```

Action: DIFFTO

The expression to compute the derivative of a toolbox function is appended to the keyword `DIFFTO`. It consists of valid MATLAB expressions and some placeholders `$*`, where the asterisk is a wildcard for one or more characters. For instance, valid placeholders are `$1`, `$2`, ..., `$N` where `N` is the number of parameters specified in the signature definition. For example, the derivative definition of the sine function is

```
BMFUNC $$ = sin($1) DIFFTO (cos($1) .* ($@1)) .
```

The placeholder `$@1` denotes the derivative of the expression `$1`. In general, for each active placeholder `$K`, a derivative is accessible through the symbol `$@K`. The parentheses, encapsulating the whole expression as well as every placeholder such as `$1` and `$@1`, are necessary, as the placeholders may be replaced by expressions and not only by variables. For example, the augmented code of the assignment `z = sin(x + y);` is given by

```
g_z = (cos(x + y) .* (g_x + g_y));
z = sin(x + y);
```

where `g_x` and `g_y` are the derivative objects associated with `x` and `y`, respectively.

In Table 3, all placeholders and their associated derivative placeholders used by the `DIFFTO` action are listed. The symbol `@` may also be used to specify the handle of a function (see `help function_handle` in MATLAB). This table introduces the derivative placeholder `$@#` associated with a variable number of input parameters, `$#`. To illustrate the use of this derivative placeholder representing a variable number of derivative parameters, the example of the function `deal` is continued. The derivative definition of this toolbox function is given by

```
BMFUNC [$$#] = deal($#) DIFFTO calln(@deal, $@#) .
```

The function `calln` supplied with ADiMat applies its first argument, which must be a function handle or function name, to all directional derivatives of all supplied derivative objects in turn. Differentiating the statement `[a, b, c]=deal(d, e, f);`, carrying out three assignments, leads to

```
[g_a, g_b, g_c] = calln(@deal, g_d, g_e, g_f);
[a, b, c] = deal(d, e, f); .
```

Here, the function `calln` executes a for-loop over the number of directional derivatives stored in the derivative objects (`g_*`) applying the function `deal`. All derivative objects must have the same number of directional derivatives.

Another feature of ADiMat is its capability to reuse the result of the original function computation. To this end, the placeholders `$$`, `$$1`, ..., `$$N`, and `$$#` may be used in a `DIFFTO` action. This is illustrated by the following example where the derivative of the p-norm with respect to a vector v is considered. Taking into account the derivative rule,

$$\frac{\partial ||v||_p}{\partial v} = ||v||_p^{1-p} \sum_i \left(|v_i|^{p-1} \frac{\partial}{\partial v} |v_i| \right) ,$$

the derivative definition is given by

Table 3. Placeholders and derivatives used by the action `DIFFTO`.

placeholder	derivative placeholder	description
`$1`	`$@1`	Derivative placeholder associated with the first to `N`th parameter of the toolbox function signature
⋮	⋮	
`$N`	`$@N`	
`$#`	`$@#`	The comma-separated list of derivative arguments associated with all arguments represented by `$#`

```
BMFUNC $$ = norm($1, $2 = 2) DIFFTO (($$) .^ (1 - $2) .*
 calln(@sum, abs($1) .^ ($2 - 1) .* g_abs($@1, $1))) ,
```

where the default value $p = 2$ is used. The first symbol (`$$`) after the `DIFFTO` keyword is the reference to the original result of the function `norm`. Given the code

```
n2 = norm(v);
n3 = norm(v,3);
```

where `v` is a vector, ADiMat generates

```
n2=norm(v);
g_n2=((n2).^(1-2).*calln(@sum, abs(v).^(2-1).*g_abs(g_v,v)))
n3=norm(v,3);
g_n3=((n3).^(1-3).*calln(@sum, abs(v).^(3-1).*g_abs(g_v,v))) .
```

The first two statements of the derivative code show the statements generated if the second argument is omitted and replaced by the default value of `2`. The constant expressions in these lines are intentionally unfolded to clarify the process of replacing the placeholders `$*`. The function `g_abs()` computes the derivative of the function `abs()`. It stops the evaluation of the expression and raises an error if `v` is zero. The order of the statements is exchanged; usually ADiMat computes the derivative before the original function. Because the derivative needs the original result, the statements are exchanged. The constant folding techniques applied to the derivative of `n2` optimizes the expression to

```
g_n2=((n2).^(-1).*calln(@sum, abs(x).*g_abs(g_v, v))) .
```

Constant folding techniques are applied twice: first after the code canonicalization and a second time after the augmentation process. The constant folding techniques are based on the ones given in [387]. The techniques are implemented by analyzing the abstract syntax tree of the MATLAB program. The constants are propagated up the tree as long as they are combined with other constant expressions only. During this process, checks to prevent under- and overflows in the IEEE floating point arithmetic are performed. A constant expression is not folded if a loss of precision is possible.

Consider a MATLAB fragment containing `norm(x, inf)`, where the infinity norm of a matrix `x` is computed. In MATLAB, this largest row sum is computed by `max(sum(abs(x')))`. A sophisticated derivative code would not only compute the infinity norm but also the index k of the maximum value of `sum(abs(x'))` which is not available in the original expression. That is, ADiMat should be capable of

rewriting the original expression in the generated derivative code. Given the value of k and taking into account the derivative rule

$$\frac{\partial ||x||_\infty}{\partial x} = \sum_i \frac{\partial}{\partial x} |x_{ki}| \ ,$$

a sophisticated derivative code would avoid any derivative computations associated with rows different from k. This feature is still under development. Therefore the action `NODIFF` is currently specified for the infinity norm of a vector or a matrix.

4 Exploiting Structure of a Given Code

Sometimes the performance of AD-code can be significantly improved by exploiting the structure of the given code. Examples include the use of interface contraction [82, 83, 275] and the integration of analytic derivatives for a small part of the given program [50]. Another example is the reuse of parts of the given code in the derivative code [81,210]. Using the macro language, it is not too difficult for the user of ADiMat to reuse parts of given code when generating the AD-code. Consider the case where a large MATLAB program uses a function implementing the Fast Hartley Transformation (FHT). Such a function may be given by

```
function y=fht(x)
t = fft(x);            # Fast Fourier transform (t may be complex)
y = real(t) - imag(t); # real part minus imaginary part
```

Since the FHT is a linear transformation of a given vector `x`, the derivative of FHT is nothing but the application of the FHT to the derivative object associated with `x`. This can be described by the specification

```
BMFUNC $$ = fht($1) DIFFTO calln(@fht, $@1) .
```

5 Conclusion and Future Work

The mathematically oriented programming language MATLAB extends its powerful core by providing collections of functions, called toolboxes. Such toolboxes may contain several hundreds of identifiers that need derivatives. Besides other techniques involving divided differences, use of third party AD tools, or re-implementation of functions in MATLAB, a powerful macro language is designed to help in defining efficient derivatives.

The macro language allows the declaration of identifiers, specification of their use as functions, constants, commands, or variables, definition of their interface, and design of a derivative expression. Function interface definition allows the narrowing of the interface to a specific class of functions by specifying the types of arguments or their values. This enables the design of highly optimized derivatives. The definition of a derivative is supported by powerful macro expansion capabilities.

Future versions of ADiMat will not only provide the ability to add derivative statements to a given code without significantly changing the original function computation, but will also rewrite the original code substantially to improve the performance. Future directions will also consider differentiating MATLAB code that calls functions written in other languages, requiring interaction of ADiMat with other AD tools.

Transforming Equation-Based Models in Process Engineering*

Christian H. Bischof[1], H. Martin Bücker[1], Wolfgang Marquardt[2], Monika Petera[1], and Jutta Wyes[2]

[1] Institute for Scientific Computing, RWTH Aachen University, Aachen, Germany
{bischof,buecker,petera}@sc.rwth-aachen.de
[2] Institute for Process System Engineering, RWTH Aachen University, Aachen, Germany
{marquardt,wyes}@lpt.rwth-aachen.de

Summary. For the solution of realistic dynamic optimization problems, the computation of derivative information is typically among the crucial ingredients in terms of both numerical accuracy and execution time. This work aims to incorporate automatic differentiation into the DyOS framework for dynamic optimization. In this framework, the optimization algorithms and the mathematical models of the process systems under consideration are implemented in separate modules. In real-life settings, a process system is formed by integrating different submodels which are possibly formulated by means of different equation-oriented modeling languages such as gPROMS or Modelica. DyOS is currently redesigned to be capable of handling such component-based models by relying on a common intermediate format called CapeML, which defines a layer of abstraction so that various models can be expressed in a manner independent from a specific modeling language. Hence, CapeML is the adequate format to which automatic differentiation is applied in this dynamic optimization framework. A novel system called ADiCape is proposed, implementing the forward mode for models written in the XML-based language CapeML. This AD transformation is expressed in the form of an XSLT stylesheet.

Key words: ADiCape, CapeML, XML, process engineering

1 Introduction

Over the past decades, modeling, simulation and optimization have gained ever-increasing attention in various scientific and engineering areas including process engineering. A general-purpose dynamic optimization software DyOS [71] has been developed at RWTH Aachen University. DyOS is used for parameter estimation,

* This work is partially funded by the German Research Foundation (DFG) in the Research Training Group 775 "Hierarchy and Symmetry in Mathematical Models."

optimal control, and optimal experimental design. The goal of an interdisciplinary research project is to provide DyOS with truncation error-free sensitivities of first and second order obtained from automatic differentiation (AD) [225, 450].

In the framework of DyOS, an engineering model is represented by a variety of equation-oriented modeling languages such as gPROMS [25, 440] or Modelica [188,371]. Interoperability of models described by a set of mathematical equations formulated in different modeling languages is provided by a common intermediate format called CapeML [525]. This XML-based representation is specifically designed as a model exchange language for process engineering. It enables the reuse and combination of submodels in a new larger model. In the context of AD, the layer of abstraction provided by CapeML is well-suited for code transformations. CapeML is similar to the XML Abstract Interface Form (XAIF) [61, 276] already in use in the implementation of AD tools. However, the two formats differ in that XAIF provides a language-independent representation of constructs common in imperative languages, whereas CapeML is designed for mathematical equations used to represent process engineering models. In this paper, we propose a prototype implementation of a new AD tool called ADiCape developed to support the evaluation of first-order directional derivatives using the forward mode for models represented in CapeML.

In Sect. 2 of this paper, a typical optimization problem arising from process engineering is described briefly, and the DyOS system is summarized. In Sect. 3, the intermediate format CapeML is introduced. The approach taken by transforming CapeML code into derivative code is sketched in Sect. 4. The purpose of Sect. 5 is to show the overall structure of the DyOS system when AD is tightly integrated.

2 Dynamic Optimization

Consider a system of differential-algebraic equations

$$M\dot{x} = F(x, p, t), \qquad t \in (t_0, t_f), \quad x \in R^n, \quad p \in R^q \tag{1}$$

with initial conditions

$$x(t_0) = x_0 \ .$$

Here, x and p denote the vectors of state variables and system parameters, respectively. The aim is to minimize some objective function $\Phi(x(t_f))$, where t_f is the final time step. An important ingredient in the optimization algorithms is the evaluation of the derivatives of the states with respect to the parameters,

$$S = \frac{\partial x}{\partial p} \ .$$

The sensitivity equation results from differentiating the system (1) with respect to the system parameters:

$$M\dot{S} = A \cdot S + K_1 \ , \quad \text{where } A = \frac{\partial F}{\partial x} \ \text{ and } K_1 = \frac{\partial F}{\partial p} \ .$$

The sensitivities are required to compute gradient information for the solution of the dynamic optimization problem. Hence, this sensitivity system must be solved in

every iteration of the algorithm for the solution of a discretized form of the original problem (1). The algorithm uses recursions of the form

$$\begin{aligned} &\text{for } k = 0, \ldots, j-1 \\ &\quad x_{k+1} = x_k - (L_j U_j)^{-1} \cdot F(x_k) \\ &\quad S_{k+1} = S_k - (L_j U_j)^{-1} \cdot (A_k \cdot S_k + K_1) \ , \end{aligned}$$

where, in an outer loop, the decomposition $L_j U_j = A_0 - M/h_j$ with step size h_j is computed. In this algorithm, the notation A_k is used for the Jacobian matrix evaluated at $x = x_k$. To set up the right-hand sides of the linear systems in the sensitivity statement, the whole Jacobian matrix A_k is not needed explicitly; only products of A_k by a multivector S_k are necessary.

The optimization scheme is separated from the description of the engineering problem. The situation is schematically depicted in Fig. 1, where the DyOS system interacts with a process model. The definition of large sets of equations of any kind generally requires a large amount of relatively complex data. In Computer-Aided Process Engineering (CAPE), this has led to introducing the concept of an Equation Set Object (ESO) [303] in a CAPE-OPEN standard definition [93]. ESO is an information representation layer, so that it can be used by other systems, e.g., nonlinear algebraic or differential-algebraic equation systems. It is an abstraction representing a square or rectangular set of equations.

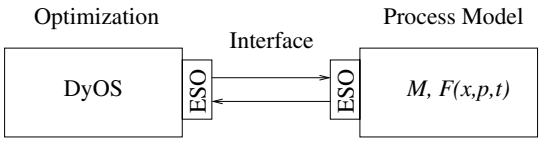


Fig. 1. Scheme of the DyOS system.

The communication between DyOS and the process model is performed through the CAPE-OPEN compliant ESO interface. This interface to the ESO object allows a solver object of DyOS to obtain information about the equations implemented in the process model. It is capable of returning, for instance, the size and structure of the equation system, of adjusting the values of the variables occurring in it, and of computing the resulting residuals.

3 The Intermediate Format CapeML

A model of a process system is a set of mathematical equations that are used to study and predict its behavior. The advantage of the equation-oriented approach for describing the model is that it does not make any assumptions about how to solve the model, or what quantities are considered known or unknown during the solution. This is an important property when considering the reuse of a model in a larger model or in a different application.

Representation of such equation-oriented models in a common and accessible way has gained a lot of interest among software developers. In particular, reducing the human effort to exchange models is crucial in building large-scale technical systems. The language XML (eXtensible Markup Language) is designed to support the exchange of a wide variety of different data. The World Wide Web Consortium released an XML representation called MathML [557] as a standard markup language, providing a low level specification format for describing and exchanging mathematics between mathematical packages and other specialized application tools. MathML seems to be a reasonable choice for a model exchange format. However, MathML does not meet some of the modeler's needs. The definition of the variables and matrices using MathML requires predetermining its dimensions, which limits the flexibility of the exchange language. Additionally, MathML does not allow one to decompose the equation system into a number of independent parts that can be recombined later using an aggregated equation system specifying the relations between the decomposed parts (submodels). Such decomposition in MathML would cause ambiguity because the variable references in MathML only use the variable names, that may occur in more than one independent part of the equation subset. See [525] for a more detailed discussion of the connections between MathML and CapeML.

A group of process modelers consisting of academic and industrial partners have taken the initiative of standardizing a representation of equation-oriented models. The goal is to develop a modeling language-neutral intermediate format which is dedicated to the exchange of models among different modeling and simulation tools. For the definition of this intermediate format, XML is used, providing standardized rules for defining other languages. The XML-based intermediate format is called CapeML [525] and is suitable for exchanging mathematical models used in process engineering. The purpose of CapeML is to abstract from a specific modeling language commonly used in this area such as gPROMS or Modelica. CapeML is designed to provide interoperability among the variety of existing modeling tools. In Fig. 2, the DyOS system is sketched where CapeML is used as an additional layer of abstraction. The language elements of CapeML include expressions, variables, constants, vectors, and equations. There are some compilers provided for Modelica and gPROMS that are capable of transforming equation systems to CapeML format.

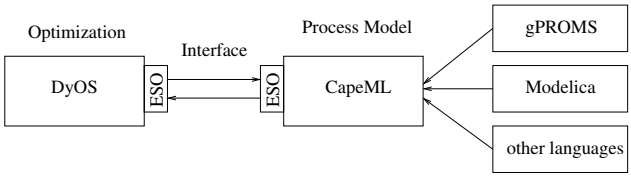


Fig. 2. DyOS system with intermediate format.

As an example, suppose that, in a given modeling language, a process model called `mod` contains the expression

```
4*sin(x)+y ,                                    (2)
```

```
<VariableDefinition myID='V-mod-x' name='x' xlink:href='Real'
specification='STATE'></VariableDefinition>
<VariableDefinition myID='V-mod-y' name='y' xlink:href='Real'
specification='STATE'></VariableDefinition>
<Expression>
  <Term>
    <Factor>
      <Number  value='4'/>
    </Factor>
    <Factor mul.op='MUL'>
      <FunctionCall fcn.name='sin'>
        <Expression>
          <Term>
            <Factor>
              <VariableOccurrence  definition='V-mod-x'>
              </VariableOccurrence>
            </Factor>
          </Term>
        </Expression>
      </FunctionCall>
    </Factor>
  </Term>
  <Term add.op='ADD'>
    <Factor>
      <VariableOccurrence  definition='V-mod-y'>
      </VariableOccurrence>
    </Factor>
  </Term>
</Expression>
```

Fig. 3. Sample code of CapeML.

where `x` and `y` are variables. This expression given in Modelica syntax can be transformed to the corresponding CapeML representation shown in Fig. 3. A CapeML code starts with the definition of variables using a separate element `VariableDefinition` for each variable. The expression consists of two elements `Term` combined by the addition operation specified in the attribute value `ADD` of the second `Term`. The first `Term` consists of two elements `Factor` combined by multiplication given in the attribute value `MUL` of the second `Factor`. The resulting tree structure, taking into account operation priorities, is given in Fig. 4. Each node of this computational graph contains a variable, a constant value, an operation, or a function call. The next section describes how to apply AD on these computational graphs.

4 ADiCape: Automatic Differentiation of CapeML

When applying the forward mode of AD to equations written in CapeML, we do not perform an activity analysis but treat all variables as active. The proposed prototype of an AD tool called ADiCape is based on a source transformation approach. For

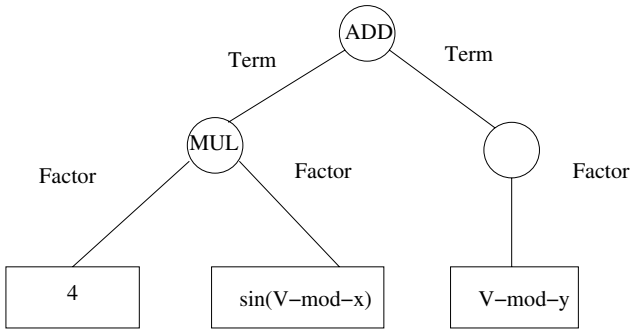


Fig. 4. Computational graph.

the transformation of CapeML code, the XSLT language [505, 556] is used. XSLT is a template-based transformation language, which can be applied on any XML-like code. To transform a given CapeML code, an XSLT stylesheet defining the rules of the transformation is needed. The CapeML code and the XSLT stylesheet are taken by an XSLT processor that generates the new CapeML code according to the specified transformations. The overall process is shown in Fig. 5.

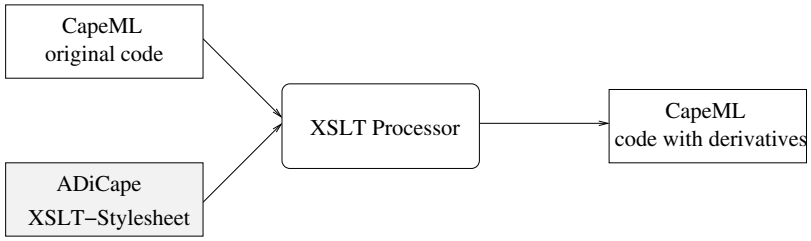


Fig. 5. Processing ADiCape.

An XSLT stylesheet consists of a set of templates specifying the transformation rules for a specific XML element. The attribute `match` of a template is used to indicate an `ELEMENT` to which a transformation is applied:

```
<xsl:template match="ELEMENT"> .
```

Within the template, directives specify how to transform the selected element. A sample template for augmenting a CapeML code with declarations for additional derivative objects is shown in Fig. 6.

This template searches for the element `VariableDefinition`. In the first step, it copies the element name, denoted by `*`, together with all its attributes, represented by `@*`, to the new CapeML code. This is specified by `"select="@*|*"`. When copying, it takes into account all other templates that satisfy the `match` field `<xsl:apply-templates/>`.

Vectors are used to propagate directional derivatives through the AD code. In CapeML, a vector consists of a tuple called domain, denoting the first and last

```
<xsl:template match="VariableDefinition">
   <!-- copy the original variable with its atributes -->
<xsl:copy>
     <xsl:apply-templates select="@*|* "/>
</xsl:copy>
   <!-- call the template "DomainVO" which will create the domain
   for the gradient vector -->
<xsl:call-template name="DomainVO"></xsl:call-template>
   <!-- create the gradient vector with new ID, name, specification
   atribute and assign distribution from the created domain-->
<xsl:element name="VariableDefinition">
     <xsl:attribute name="myID">
          <xsl:value-of select="concat('V-',$model,'g_',@name)"/>
     </xsl:attribute>
     <xsl:attribute name="name">g_<xsl:value-of select="@name"/>
     </xsl:attribute>
     <xsl:attribute name="xlink:href">
          <xsl:value-of select="@xlink:href"/></xsl:attribute>
     <xsl:attribute name="specification">
          <xsl:value-of select="@specification"/>
     </xsl:attribute>
     <xsl:element name="Distribution">
          <xsl:attribute name="domain">
             <xsl:value-of select="concat('D-',$model,'g_',@name)"/>
          </xsl:attribute>
     </xsl:element>
</xsl:element>
</xsl:template>
```

Fig. 6. Sample template of ADiCape XSLT stylesheet.

index that can be used in referencing a vector. Therefore, the next step is to create a domain for the new gradient vector object associated with the selected variable. This is done by calling another template named `DomainVO`.

Finally, the new element `VariableDefinition` with four attributes is created. A unique key is composed using the name of the model and the name of the selected variable. The name of the generated derivative object is constructed by adding the prefix `g_` to the name of the selected variable. The remaining two attributes are copied from the corresponding attributes of the selected variable. The element `Distribution` points out the adequate domain for the created gradient object. The result of applying this template to the declaration of the variable `x` given in the first two lines of Fig. 3 is shown in Fig. 7. Here, the domain is expressed in the form of two numbers each of which needs a complete expression tree in CapeML. The number of directional derivatives is assumed to be two in this transformation so that accesses to any position between one and two are valid for vectors in CapeML.

Applying the forward mode to the expression (2) whose CapeML representation is given in Fig. 3 leads to

```
4*cos(x)*g_x+g_y .
```

```
<VariableDefinition myID="V-mod-x" name="x" xlink:href="Real"
specification="STATE"></VariableDefinition>
<Domain myID="D-mod-g_x" symbol="d-1" type="DISCRETE" name="V-0">
  <Expression>
    <Term>
      <Factor>
        <Number value="1"/>
      </Factor>
    </Term>
  </Expression>
  <Expression>
    <Term>
      <Factor>
        <Number value="2"/>
      </Factor>
    </Term>
  </Expression>
</Domain>
<VariableDefinition myID="V-mod-g_x" name="g_x"
xlink:href="Real" specification="STATE">
  <Distribution domain="D-mod-g_x"/>
</VariableDefinition>
```

Fig. 7. Variable definition of `x` and `g_x` after transformation with the template.

This expression is recognized again in the CapeML code shown in Fig. 8 obtained by processing the CapeML expression from Fig. 3 and the ADiCape XSLT stylesheet with an XSLT processor. In summary, the current version of the XSLT stylesheet implementing ADiCape is about 1000 lines long. We considered a simple expression in Modelica (2) whose CapeML representation consists of 25 lines, from which ADiCape generated approximately 45 lines of forward mode AD code in CapeML.

5 The Overall Structure of the System

The current implementation of ADiCape creates two separate files with CapeML code. The first file is a copy of the original CapeML file containing the equations describing the process model; the second file contains the equations representing the derivative information exclusively. The overall structure of the DyOS system using derivative information provided by ADiCape is presented in the Fig. 9. As without the ADiCape system, the communication with the process model is realized via an ESO object. With the ADiCape system, however, there is an additional ESO object for the derivative equations. Furthermore, a new XML ESO class is introduced. This new class identifies what kind of information is needed by the DyOS system. It acts as a relay by sending the requests to the appropriate ESO. More precisely, the computation of the residuals of a specified equation can only be calculated by the ESO from the original code, whereas the Jacobian of the equation system and also the directional derivatives can be returned from the second ESO object.

```
<Expression>
  <Term>
    <Factor>
      <Number value="4"/>
    </Factor>
    <Factor mul.op="MUL">
      <FunctionCall fcn.name="cos">
        <Expression>
          <Term>
            <Factor>
              <VariableOccurrence definition="V-mod-x">
              </VariableOccurrence>
            </Factor>
          </Term>
        </Expression>
      </FunctionCall>
    </Factor>
    <Factor mul.op="MUL">
      <VariableOccurrence definition="V-mod-g_x">
        <ArrayIndex>
          <Expression>
            <Term>
              <Factor>
                <DomainOccurrence domain="V-mod-i"/>
              </Factor>
            </Term>
          </Expression>
        </ArrayIndex>
      </VariableOccurrence>
    </Factor>
  </Term>
  <Term add.op="ADD">
    <Factor>
      <VariableOccurrence definition="V-mod-g_y">
        <ArrayIndex>
          <Expression>
            <Term>
              <Factor>
                <DomainOccurrence domain="V-mod-i"/>
              </Factor>
            </Term>
          </Expression>
        </ArrayIndex>
      </VariableOccurrence>
    </Factor>
  </Term>
</Expression>
```

Fig. 8. Differentiated CapeML code.

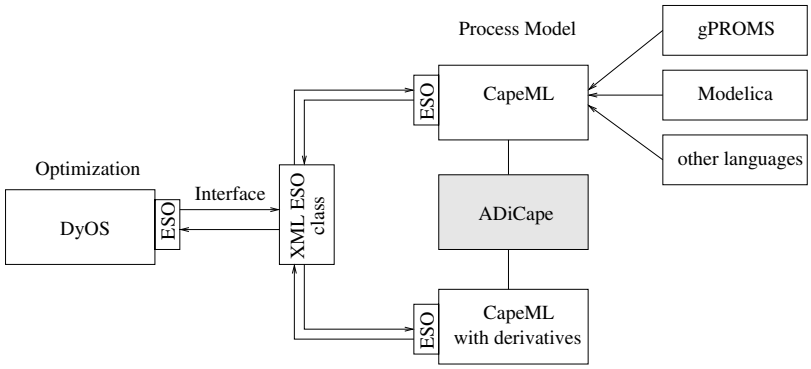


Fig. 9. DyOS system after the integration of ADiCape.

6 Concluding Remarks and Directions for Future Work

DyOS is a specific framework for dynamic optimization geared toward process engineering. The system currently is redesigned to be capable of using an XML-based intermediate format, CapeML, to represent a process model. The advantage of this additional level of abstraction is that DyOS easily can work with different models regardless of their generic description. The CapeML format not only enables interoperability between different modeling languages; it is also an appropriate level of abstraction on which automatic differentiation is implemented. Truncation error-free first-order derivative information is brought to DyOS by the forward mode applied to process models. The prototype system transforming equations expressed in CapeML is called ADiCape. The functionality of ADiCape was demonstrated on a small example. The extensions of the DyOS system needed for interaction with ADiCape are minimal. Essentially, a new XML ESO class is needed, redirecting a DyOS request to the appropriate ESO object.

Recall from Sect. 2 that not only the full Jacobian is needed but also Jacobian-multivector products are to be computed. The current ESO specification does not support these operations. Hence, a new definition of the ESO interface is needed to fully benefit from the advantages of AD. Another extension of the ESO is reasonable when second-order derivatives enter the picture. Recent work [19,423] demonstrates the importance of accurate Hessian information in the context of dynamic optimization. The aim of future work is to additionally use second-order derivatives by observing that the sensitivity equations for second-order derivatives are linear [519]. These sensitivity calculations will be added to the integration algorithm in DyOS. The ADiCape stylesheet will then construct a third CapeML file with the directions to calculate Hessians or Hessian-vector products. The XML ESO class, redirecting the requests from DyOS to the adequate ESO object, will then also be extended to communicate with a third ESO instance responsible for second-order derivatives. It is also interesting to find out to what extent ADiCape can interact with the ACTS project [403], also relying on an XML-based intermediate format.

Simulation and Optimization of the Tevatron Accelerator

Pavel Snopok[1], Carol Johnstone[2], and Martin Berz[3]

[1] Fermi National Accelerator Laboratory, Michigan State University
snopok@pa.msu.edu
[2] Fermi National Accelerator Laboratory
cjj@fnal.gov
[3] Michigan State University
berz@msu.edu

Summary. The Tevatron accelerator, currently the particle accelerator with the highest energy in the world, consists of a ring with circumference of four miles in which protons are brought into collision with antiprotons at speeds very close to the speed of light. The accelerator currently under development at Fermilab represents a significant upgrade, but experienced significant limitations during initial operation. The correction of some of the problems that appeared using techniques of automatic differentiation are described. The skew quadrupole correction problems are addressed in more detail, and different schemes of correction are proposed.

Key words: Tevatron, optimization, skew quadrupoles, COSY INFINITY, tracking, Fermilab, high-order multivariate AD, beam physics, flows, ODE's, transfer map, stability analysis

1 Introduction

The dynamics in a large particle accelerator are governed by relativistic equations of motion that are usually solved relative to those of a reference orbit. The simulation of an accelerator in this manner is a very demanding undertaking since particles orbit for in the order of 10^9 revolutions, and it is necessary to study many different orbits. Thus from the early days of particle accelerators, it has been customary to determine Taylor expansions of the flow, usually to orders two and three. Automatic differentiation methods, in particular in combination with ODE solving tools based on differential algebraic methods, have allowed us to increase this computation order very significantly, and now orders around 10 are routinely used in the code COSY INFINITY [39]. For the tracking pictures presented here, the order of calculation is usually taken to be seven for speed, but for some final results orders 11 or 13 are used. Furthermore, it is now possible to represent the devices by much more accurate models. A wide range of standard elements with the ability to simulate all nonlinearities and associated error fields is available in COSY INFINITY.

2 The Tevatron Accelerator – Machine Description

The Tevatron is currently the most powerful particle accelerator in the world with a circumference of the ring of four miles. The beams of protons and antiprotons moving in opposite directions are brought to collision at energies close to 1 TeV each. Hence, their relativistic kinetic energy is more than 1000 times that of their rest mass. Besides colliding two beams, it is necessary to make as many particles interact as possible. The effectiveness of the collision is characterized by a single value called luminosity. Calculating and optimizing the dynamics of the particle motion in the accelerator to reach higher and higher luminosity is one of the main goals of this work.

Particles in the Tevatron have velocities close to the speed of light, a fact which has several advantages and disadvantages for the modeling. COSY INFINITY takes all the resulting relativistic effects into account automatically.

The Tevatron consists of six arcs connected with six straight sections. Two of the straights are the well-known collision detectors CDF and D0. Each arc is a periodic structure having 15 FODO cells with 8 dipoles each to provide the necessary bending. However, each of these magnets has some error terms that are commonly called multipole moments. One of these errors is due to the fact that the coils of the dipoles are not parallel as they should be, which introduces a skew quadrupole term. This has a very detrimental effect on the motion of the particles, because it provides a coupling of the otherwise independent horizontal and vertical motion and thus affects the stability of the particles. A circuit of skew quadrupole correctors serves to compensate for these errors. The main problem addressed in this article is the scheme of such a correction and the view that only a part of the errors in the dipoles can be removed during the next Tevatron planned shutdown.

3 The Model, Criteria and Parameters to Control

We began with the basic model of the machine by V. Lebedev currently available at Fermi National Accelerator Laboratory [328–330] and converted the source code to run under COSY INFINITY. The tool converting the lattice description works automatically, so all the updates to the lattice easily can be taken into account. The current model implements elements: dipoles, quadrupoles, skew quadrupoles, sextupoles, skew sextupoles, solenoids with fringe field, and separators.

The recent upgrade of the Tevatron accelerator led to an undesirable coupling between the horizontal and vertical motion [202, 355, 493–496], while usually great care is taken to keep these two motions decoupled. Mere integration of orbits makes the task of decoupling very difficult, since it is very hard to assess from ray coordinates whether a coupling happens. On the other hand, in the framework of the Taylor expansion of final coordinates in terms of initial coordinates, decoupling merely amounts to

$$\frac{\partial x_k^{(f)}}{\partial y_j^{(i)}} = 0 \quad \text{and} \quad \frac{\partial y_k^{(f)}}{\partial x_j^{(i)}} = 0, \quad k, j = 1, 2 \tag{1}$$

for the respective linear effects, and to

$$\frac{\partial^{i_1+i_2} x_k^{(f)}}{\partial \left(y_1^{(i)}\right)^{i_1} \partial \left(y_2^{(i)}\right)^{i_2}} = 0 \quad \text{and} \quad \frac{\partial^{i_1+i_2} y_k^{(f)}}{\partial \left(x_1^{(i)}\right)^{i_1} \partial \left(x_2^{(i)}\right)^{i_2}} = 0, \quad k = 1, 2 \tag{2}$$

for nonlinear effects, where x_1 is the horizontal position, $x_2 = p_x/p_0$ is the reduced horizontal momentum, y_1 is the vertical position, $y_2 = p_y/p_0$ is the reduced vertical momentum, p_0 is some previously chosen scaling momentum; i_1, $i_2 : i_1 \geq 0$; $i_2 \geq 0$; $i_1 + i_2 < n$, n is the order of calculations, and the superscripts (i) and (f) denote the initial and final conditions, respectively.

At the same time, the main operating parameters of the machine, the two tunes (phases of the eigenvalues of the linear map), have to be kept constant. This is vital for the stability of the motion of the particles to help avoid resonances. In terms of partial derivatives, this condition amounts to the preservation of

$$\frac{\partial x_1^{(f)}}{\partial x_1^{(i)}} + \frac{\partial x_2^{(f)}}{\partial x_2^{(i)}} \quad \text{and} \quad \frac{\partial y_1^{(f)}}{\partial y_1^{(i)}} + \frac{\partial y_2^{(f)}}{\partial y_2^{(i)}} \; . \tag{3}$$

Thus with the availability of Taylor expansions, it is merely necessary to adjust suitable system parameters such that the ten conditions (in the linear case) described by (1) and (3) are met. While by no means an easy feat, this task is significantly more manageable than the attempt to optimize performance based on particle coordinates.

We are to control the strengths of several skew quadrupole correctors around the ring both in arcs and straight sections. Each arc has six to eight correctors, but it is preferable to have them all at the same strength as they have one power supply. Moreover, it would be advantageous to optimize them all to the same strength in all the arcs. The study shows that this can be done effectively and efficiently with AD methods implemented in COSY INFINITY. Without use of AD techniques all the calculations and especially optimization of the structure requiring intensive multiple recalculating of the transfer maps would take a prohibitively long time and could never be done for such a high order.

To keep the tune of the system constant, the strength of the main bus quadrupoles can be slightly changed. As all the quadrupoles have the same strength this task is a one-parameter optimization; besides it does not require high-order calculation. Even in this problem the high-order calculations are unavoidable, as checks should be made that changing the tune back to the original value after skew quadrupole correction optimization does not lead to degradation in the behavior of the particles described by the multi-revolution tracking picture.

4 Map Methods

The particles in most accelerators (the Tevatron is not an exception) usually stay close together, forming a beam. Therefore, it is convenient to choose a reference particle – the one that moves undisturbed through the centers of all the magnets of the machine and make use of perturbative techniques to obtain good approximation of the dynamics of motion of all the particles in the beam relative to this reference particle.

The motion of the particles is considered in 2-dimensional phase space: each particle has four coordinates x_1, $x_2 = p_x/p_0$, y_1, and $y_2 = p_y/p_0$, where p_0 is the

momentum of the reference particle, and the arc length s along the center of the ring is used as an independent variable. Thus, the dynamics is described by the vector

$$\mathbf{z}(s) = \begin{pmatrix} x_1 \\ x_2 \\ y_1 \\ y_2 \end{pmatrix} ,$$

which depends on s. The action of the accelerator lattice elements can be expressed by how they change the components of the vector $\mathbf{z}(s)$. Denoting by $\mathbf{z}_0$ the initial coordinates at s_0, the final coordinates of each particle at s can be obtained from the system of equations

$$\mathbf{z}(s) = \mathcal{M}(s_0, s)\,(\mathbf{z}_0, \boldsymbol{\delta}) \ ,$$

relating $\mathbf{z}_0$ and a set of control parameters $\boldsymbol{\delta}$ at s_0 to $\mathbf{z}$ at $s > s_0$, where $\mathcal{M}(s_0, s)$ is a function which formally summarizes the action of the system. $\mathcal{M}$ is called the transfer function or transfer map of the system. The transfer map satisfies

$$\mathcal{M}(s_1, s_2) \circ \mathcal{M}(s_0, s_1) = \mathcal{M}(s_0, s_2) \ .$$

Therefore, the transfer map of the system can be built up from the transfer maps of individual elements. As the accelerator structure is regular, the set of different elements is not that big, a fact that allows us to seriously reduce the amount of calculations.

$\mathcal{M}$ is usually weakly nonlinear and can be considered as a sum of two maps: a purely linear part M, and a purely nonlinear part $\mathcal{N}$, i.e. $\mathcal{M} = M + \mathcal{N}$. For a simple analysis of the relative motion, often a linear approximation is enough, but for full understanding of the motion, the understanding of the nonlinear effects is essential.

Map methods are particularly useful for the study of the motion in circular accelerators, as one has to run the particles through the same system many times. The number of revolutions necessary to estimate the behavior of the particles in the Tevatron lies in the orders of $10^5 - 10^7$. Having a transfer map of one full revolution, one easily can perform repetitive tracking of the particles through the system.

In case of the first order of calculations, the transfer map can be represented as a matrix of coefficients:

$$\begin{pmatrix} x_1^{(f)} \\ x_2^{(f)} \\ y_1^{(f)} \\ y_2^{(f)} \end{pmatrix} = \begin{pmatrix} 2.0756 & 0.0023 & 0.1123 & 0.0047 \\ 63.3276 & 0.5497 & 1.1253 & 0.1065 \\ 0.0003 & 0.0021 & 1.4570 & 0.0223 \\ -3.2571 & 0.0834 & 87.8001 & 2.0272 \end{pmatrix} \begin{pmatrix} x_1^{(i)} \\ x_2^{(i)} \\ y_1^{(i)} \\ y_2^{(i)} \end{pmatrix} .$$

For higher orders the numbering and processing of the coefficient is much more sophisticated. As an example, we here show a piece of a high-order transfer map:

```
I  COEFFICIENT             ORDER EXPONENTS
1  -.7246219112151764        1   1 0  0 0
2  -.9122414947039915        1   0 1  0 0
3  0.1588622636737591E-07    1   0 0  1 0
4  -.5419413125727635E-09    1   0 0  0 1
5   603.2477358626691        2   2 0  0 0
6  -1149.461216254035        2   1 1  0 0
```

```
  7   783.4790397865372       2   0 2  0 0
  8  -23.83485869335665       2   1 0  1 0
  9   50.34274836239869       2   0 1  1 0
 10  -222.4936370747880       2   1 0  0 1
 11   98.21519926034055       2   0 1  0 1
 12  -73.88432808806901       2   0 0  2 0
 13   149.0913750043664       2   0 0  1 1
 14  -223.3498687001470       2   0 0  0 2
 15  -71088.03707913239       3   3 0  0 0
 16   164852.2891153482       3   2 1  0 0
...  ..................       .   . .  . .
325  -16401453090653.15       7   0 0  4 3
326   29169856173501.98       7   0 0  3 4
327  -42361303120466.99       7   0 0  2 5
328   54860608468975.03       7   0 0  1 6
329  -43643493779626.68       7   0 0  0 7
```

Each row describes one term of the Taylor expansion of final coordinates in terms of initial coordinates. The columns labeled "EXPONENTS" describe the exponents of each of the independent variables appearing in the respective term. The "ORDER" column contains the total order of the term, i.e. the sum of the exponents, and the first column lists the double precision coefficient belonging to the respective term. As an example, the sixth row describes the Taylor series term depending on the power one of variable 1 and the power one of variable 2.

The map coefficients are the results of integrating the equations of motion of the particles through different lattice elements: quadrupoles, sextupoles, solenoids. The built-in ODE solver in COSY INFINITY works with differential algebra vectors – coefficients of Taylor expansion for the coordinates of the particles, which achieves very high orders of computations even on somewhat slow machines.

For the optimization of the linear coupling, the linear map is sufficient, as (1) and (3) affect only the linear part of the map. For subsequent correction of the nonlinear effects as in (2), it is necessary to determine higher order Taylor expansions of the map. More on the work with map methods and differential algebra approaches can be found in [38, 41].

The optimization works the following way.

1. Choose a correction scheme with different skew quadrupole circuits settings;
2. Perform the two-stage optimization. The first stage removes the linear coupling in each of the six arcs (the objective function is the sum of the derivatives in the left hand sides of the expressions (1)). The second stage removes the coupling for the whole machine (the objective function is the sum of the derivatives in the left hand sides of the expressions (1)) and brings the tune back to its initial value (the objective function is the sum of the expressions

$$\frac{\partial x_1^{(f)}}{\partial x_1^{(i)}} + \frac{\partial x_2^{(f)}}{\partial x_2^{(i)}} - C_x \quad \text{and} \quad \frac{\partial y_1^{(f)}}{\partial y_1^{(i)}} + \frac{\partial y_2^{(f)}}{\partial y_2^{(i)}} - C_y \ ,$$

   where C_x and C_y are some fixed values);
3. Perform high-order tracking to check the stability of the motion after optimization.

The qualitative results of the optimization for two different skew quadrupole circuits are presented in Sect. 5, the quantitative results are discussed in Sect. 6.

5 Different Optimization Schemes and Proposals

Currently, 85% of the dipoles in the Tevatron still have the above-mentioned coil displacement. As a result, skew quadrupole components act on the particles. A first study consisted of leaving all the errors in place, but moving some of the correctors. The results for such optimization scheme are not shown, because this scheme was only interesting as a starting point, since subsequently it proved impossible to move any of the skew quadrupole correctors. At the same time, the scheme shows that good correction can be performed even if all the errors stay the same.

More realistic are the schemes with some of the errors in dipoles fixed and all or only the part of the correctors in their places. The forecast says up to 50% of the dipoles can be fixed during the upcoming Tevatron shutdown. The high-order analysis helps to determine exactly what should be done, which dipoles to correct and what the strength of the correctors should be to achieve the most predictable particle behavior, avoid resonances, and decouple the motion.

					Sector E scheme I		
FODO 1	D*2	FQ	SQC	D*2	D*2 FIX	DQ	D*2FIX
FODO 2	D*2	FQ		D*2	D*2 FIX	DQ	D*2FIX
FODO 3	D*2	FQ	SQC	D*2	D*2 FIX	DQ	D*2FIX
FODO 4	D*2	FQ		D*2	D*2 FIX	DQ	D*2FIX
FODO 5	D*2	FQ	SQC	D*2	D*2 FIX	DQ	D*2FIX
FODO 6	D*2	FQ		D*2	D*2 FIX	DQ	D*2FIX
FODO 7	D*2	FQ	SQC	D*2	D*2 FIX	DQ	D*2FIX
FODO 8	D*2	FQ		D*2	D*2 FIX	DQ	D*2FIX
FODO 9	D*2	FQ	SQC	D*2	D*2 FIX	DQ	D*2FIX
FODO 10	D*2	FQ		D*2	D*2 FIX	DQ	D*2FIX
FODO 11	D*2	FQ	SQC	D*2	D*2 FIX	DQ	D*2FIX
FODO 12	D*2	FQ		D*2	D*2 FIX	DQ	D*2FIX
FODO 13	D*2	FQ	SQC	D*2	D*2 FIX	DQ	D*2FIX
FODO 14	D*2	FQ		D*2	D*2 FIX	DQ	D*2FIX
FODO 15	D*2	FQ	SQC	D*2	D*2 FIX	DQ	D*2FIX

Fig. 1. Correction scheme I, errors in dipoles fixed in each cell around defocusing quadrupole.

The first scheme layout is shown in Fig. 1 in the form of the description of one sector. All the other sectors look similar except for some dipoles that have been fixed before this study was initiated. This scheme proposes to fix skew quadrupole errors in the dipoles (D*2 FIX) on both sides of the defocusing quad (DQ). All the skew quadrupole correctors stay in their places (SQC). The errors in dipoles around focusing quadrupole (FQ) remain unfixed (D*2).

The results of the optimization are given in Figs. 2–5. The first two pictures show the phase portraits in x_1, $x_2 = p_x/p_0$ and y_1, $y_2 = p_y/p_0$ planes for particle

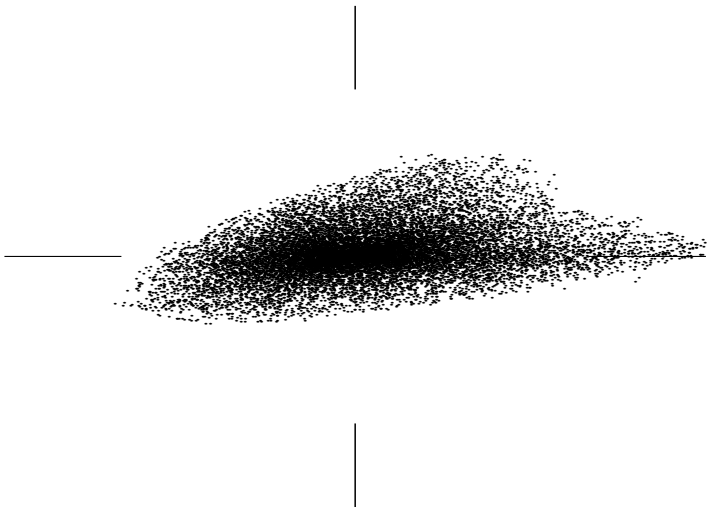


Fig. 2. x-plane phase portrait before the optimization with 85% skew quadrupole errors in dipoles.

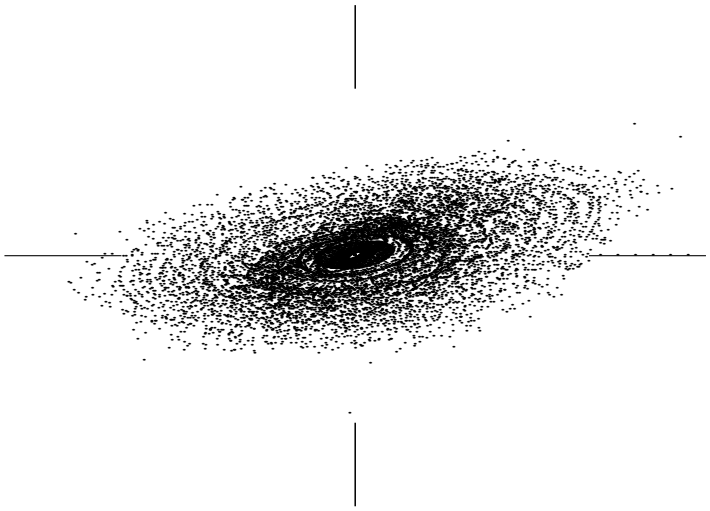


Fig. 3. y-plane phase portrait before the optimization with 85% skew quadrupole errors in dipoles.

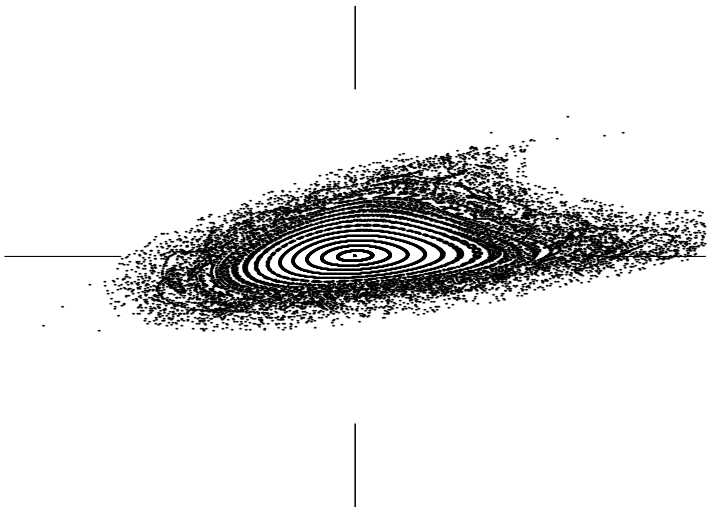


Fig. 4. x-plane phase portrait after the optimization with 50% skew quadrupole errors in dipoles, errors are fixed around each defocusing quadrupole.

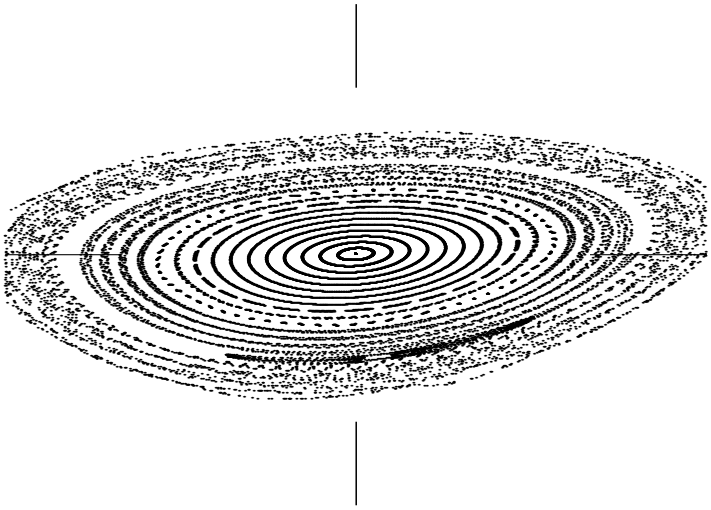


Fig. 5. y-plane phase portrait after the optimization with 50% skew quadrupole errors in dipoles, errors are fixed around each defocusing quadrupole.

trajectories before the optimization. The next two show the results after the optimization. The scale of each picture is $2.4 \times 10^{-3}\, m$ horizontal and 4.0×10^{-3} vertical. For stable particles the trajectories look like closed curves. For the y plane picture they are very close to ellipses, for the x plane the trajectories have a somewhat triangular shape because of the proximity of a resonance. For unstable particles, the trajectories are fuzzy. For some extremely bad cases (like the one in the Figs. 2 and 3), the particles do not show stable behavior at all. After several turns most of the particles can be considered lost (their traces go beyond the scale of the picture).

The figures show great improvement in the behavior of the particles: fewer particles are lost, and the motion remains stable further from the reference particle that goes through the centers of all the magnets undisturbed. The picture of the x plane is still not perfect; there appears to be a possibility that the stable region can be increased further.

FODO 1	D*2	FQ	SQC RMV	D*2	D*2	DQ	D*2
FODO 2	D*2 FIX	FQ		D*2 FIX	D*2 FIX	DQ	D*2 FIX
FODO 3	D*2	FQ	SQC RMV	D*2	D*2	DQ	D*2
FODO 4	D*2 FIX	FQ		D*2 FIX	D*2 FIX	DQ	D*2 FIX
FODO 5	D*2	FQ	SQC	D*2	D*2	DQ	D*2
FODO 6	D*2 FIX	FQ		D*2 FIX	D*2 FIX	DQ	D*2 FIX
FODO 7	D*2	FQ	SQC	D*2	D*2	DQ	D*2
FODO 8	D*2 FIX	FQ		D*2 FIX	D*2 FIX	DQ	D*2 FIX
FODO 9	D*2	FQ	SQC	D*2	D*2	DQ	D*2
FODO 10	D*2 FIX	FQ		D*2 FIX	D*2 FIX	DQ	D*2 FIX
FODO 11	D*2	FQ	SQC	D*2	D*2	DQ	D*2
FODO 12	D*2 FIX	FQ		D*2 FIX	D*2 FIX	DQ	D*2 FIX
FODO 13	D*2	FQ	SQC	D*2	D*2	DQ	D*2
FODO 14	D*2 FIX	FQ		D*2 FIX	D*2 FIX	DQ	D*2 FIX
FODO 15	D*2	FQ	SQC RMV	D*2	D*2	DQ	D*2

Fig. 6. Correction scheme II, skew quadrupole errors in dipoles are fixed in each even FODO cell.

Scheme II (Fig. 6) answers this question. There are two differences in the second scheme: the dipoles are to be fixed in all the even FODO cells, while skew quadrupole correctors are located in odd cells. In addition, some of the correctors can be removed, and that makes the results even better (marked SQC RMV in Fig. 6).

Figures 7 and 8 present the results of particle tracking for correction scheme II. Clearly the improvement can be seen with the naked eye. All the particles remain stable for $10,000$ turns. This result is achieved with only one corrector strength per arc, which will work fine with one power supply for all the skew quadrupole correctors in each arc. Moreover, optimization gives only slightly worse results for the case where all the skew quadrupole correctors have the same strength around the entire ring, which means the correction scheme under consideration appears to be the best choice to implement.

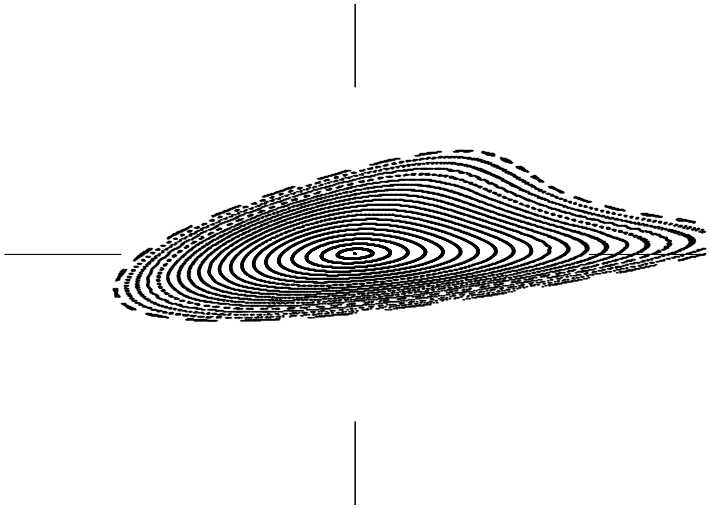


Fig. 7. x-plane phase portrait after the optimization with 50% skew quadrupole errors in dipoles, errors are fixed in each even FODO cell.

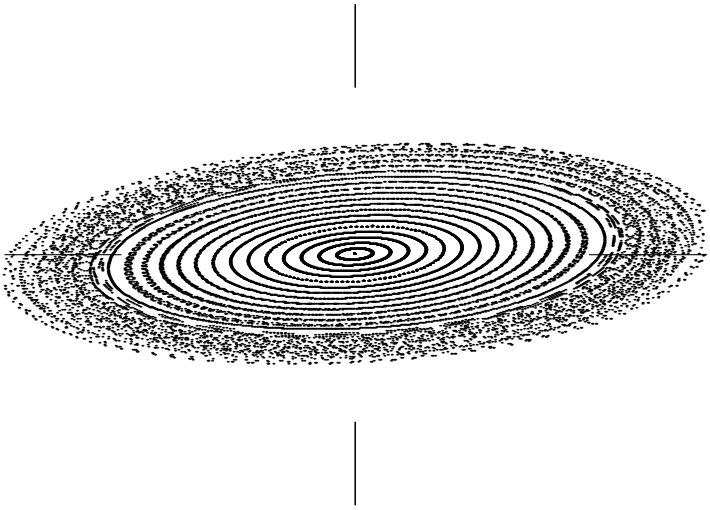


Fig. 8. y-plane phase portrait after the optimization with 50% skew quadrupole errors in dipoles, errors are fixed in each even FODO cell.

6 Transfer Map Comparison

Since one of the aims of the optimization was the removal of coupling between the x and y planes, it is worth showing the first order transfer map of the machine before and after optimization. The map before optimization looks like this:

$$\begin{pmatrix} -0.7553149 & 0.3637584 & 0.0663166 & -0.0971958 \\ -0.8640196 & -0.9507699 & -0.4305946 & 0.1421480 \\ -0.0251393 & -0.0621722 & -0.8060247 & 0.3353297 \\ -0.4966847 & 0.0614653 & -0.7083212 & -0.9862029 \end{pmatrix}, \tag{4}$$

and after optimization:

$$\begin{pmatrix} -0.8023857 & 0.3107970 & \mathbf{0.0059011} & \mathbf{-0.00254055} \\ -0.7143330 & -0.9696470 & \mathbf{-0.0058075} & \mathbf{-0.00487758} \\ \mathbf{0.0043365} & \mathbf{-0.0022290} & -0.8445417 & 0.28890307 \\ \mathbf{-0.0077313} & \mathbf{-0.0060658} & -0.5409385 & -0.99907987 \end{pmatrix}. \tag{5}$$

The coupling terms shown in bold (four terms in the upper-right and lower-left corners) in (5) became up to 74 times smaller than in (4).

7 Conclusions

Without the use of AD techniques implemented in COSY INFINITY, achieving the results would be a hard, if not impossible task. The speed COSY tracks particles is remarkable. One procedure tracking in both x and y planes takes about 3 minutes for 7th order calculation or up to 8 hours (depending on the one-turn map) for order 11 on a 1.5 GHz Pentium processor.

The results of the study are very promising, and the correction scheme II presented above is the one being implemented during the Tevatron shutdown planned for Fall 2004.

Periodic Orbits of Hybrid Systems and Parameter Estimation via AD*

Eric Phipps[1], Richard Casey[2], and John Guckenheimer[3]

[1] Sandia National Laboratories, Albuquerque, NM
etphipp@sandia.gov
[2] Center for Applied Mathematics, Cornell University, Ithaca, NY
rjc20@cornell.edu
[3] Mathematics Department, Cornell University, Ithaca, NY
gucken@cam.cornell.edu

Summary. Periodic processes are ubiquitous in biological systems, yet modeling these processes with high fidelity as periodic orbits of dynamical systems is challenging. Moreover, mathematical models of biological processes frequently contain many poorly-known parameters. This paper describes techniques for computing periodic orbits in systems of hybrid differential-algebraic equations and parameter estimation methods for fitting these orbits to data. These techniques make extensive use of automatic differentiation to evaluate derivatives accurately and efficiently for time integration, parameter sensitivities, root finding and optimization. The resulting algorithms allow periodic orbits to be computed to high accuracy using coarse discretizations. Derivative computations are carried out using a new automatic differentiation package called ADMC++ that provides derivatives and Taylor series coefficients of matrix-valued functions written in the MATLAB programming language. The algorithms are applied to a periodic orbit problem in rigid-body dynamics and a parameter estimation problem in neural oscillations.

Key words: Periodic orbits, hybrid systems, parameter estimation, Taylor series, differential equations, rigid-body dynamics, neural oscillations, MATLAB, ADMC++

Rhythmic, periodic processes are ubiquitous in biological systems; for example, the heart beat, walking, circadian rhythms, and the menstrual cycle. Modeling these processes with high fidelity as periodic orbits of dynamical systems is challenging because

- (most) nonlinear differential equations can only be solved numerically
- accurate computation requires solving boundary value problems

* This research was partially supported by the Department of Energy and the National Science Foundation. While at Sandia, the first author was supported by Sandia's ASC and CSRF programs.

- many problems and solutions are only piecewise smooth
- many problems require solving differential-algebraic equations
- computing sensitivities of solutions with respect to model parameters requires solving variational equations
- truncation errors in numerical integration degrade performance of optimization methods for parameter estimation.

In addition, mathematical models of biological processes frequently contain many poorly-known parameters, and the problems associated with this impede the construction of detailed, high-fidelity models. Modelers are often faced with the difficult problem of using simulations of a nonlinear model, with complex dynamics and many parameters, to match experimental data. Improved computational tools for exploring parameter space and fitting models to data are clearly needed.

This paper describes techniques for computing periodic orbits in systems of hybrid differential-algebraic equations and parameter estimation methods for fitting these orbits to data. These techniques make extensive use of automatic differentiation to evaluate derivatives accurately and efficiently for time integration, parameter sensitivities, root finding and optimization. The boundary value problem representing a periodic orbit in a hybrid system of differential-algebraic equations is discretized via multiple-shooting using a high-degree Taylor series integration method [244,433]. Numerical solutions to the shooting equations are then estimated by a Newton process, yielding an approximate periodic orbit. A metric is defined for computing the distance between two given periodic orbits, which is then minimized using a trust-region minimization algorithm [143] to find optimal fits of the model to a reference orbit [99].

The use of Taylor series integration in the context of computing periodic orbits in systems of ordinary differential equations has been studied previously [244], and provides several key advantages that motivate the extensions to hybrid DAE systems and parameter estimation methods presented here. The high accuracy and large step sizes associated with Taylor series integration allow periodic orbits to be computed accurately using coarse discretizations. Moreover, Taylor series integration provides dense output allowing event location without interpolation and no loss of order of accuracy. Also, sensitivities of the computed trajectories can be easily computed using automatic differentiation. As will be demonstrated below, Taylor series integration has a simple extension to problems with algebraic constraints, and the computed trajectories satisfy these constraints with high accuracy. Finally, multiple-shooting frameworks are fairly simple to design, implement, and allow us to treat many classes of problems uniformly with a single software implementation. Thus Taylor series integration and multiple-shooting provide a natural setting for developing the algorithms presented here.

In addition to these properties, there are two goals that further motivate the work presented here. The first is to provide a simple and powerful framework for studying periodic motions in mechanical systems. Formulating mechanically correct equations of motion for systems of interconnected rigid bodies, while straightforward, is a time-consuming and error prone process. Much of this difficulty stems from computing the acceleration of each rigid body in an inertial reference frame. The acceleration is computed most easily in a redundant set of coordinates giving the spatial positions of each body, since the acceleration is just the second derivative of these positions. Rather than providing explicit formulas for these derivatives, automatic differentiation can be employed to compute these quantities efficiently

during the course of a simulation. The feasibility of these ideas was investigated by applying these techniques to the problem of locating stable walking motions for a disc-foot passive walking machine [131, 194, 363].

The second goal for this project was to investigate the application of smooth optimization methods to periodic orbit parameter estimation problems in neural oscillations. Others [48, 183, 517] have favored non-continuous optimization methods such as genetic algorithms, stochastic search methods, simulated annealing and brute-force random searches because of their perceived suitability to the landscape of typical objective functions in parameter space, particularly for multi-compartmental neural models. Here we argue that a carefully formulated optimization problem is amenable to Newton-like methods and has a sufficiently smooth landscape in parameter space that these methods can be an efficient and effective alternative.

The plan of this paper is as follows. In Sect. 1 we provide a definition of hybrid systems that is the basis for modeling systems with discontinuities or discrete transitions. Sections 2, 3, and 4 briefly describe the Taylor series integration, periodic orbit tracking, and parameter estimation algorithms. For full treatments of these algorithms, we refer the reader to [99, 433]. The software implementation of these algorithms is briefly described in Sect. 5 with particular emphasis on the ADMC++ automatic differentiation software package. Finally, these algorithms are applied to the bipedal walking and Hodgkin-Huxley based neural oscillation problems discussed above in Sect. 6.

1 Hybrid Systems

An important feature of many practical nonlinear problems is the existence of discontinuities or discrete transitions in the problem's dynamics. For example, in the bipedal walking problem presented below, an impact occurs each time a foot strikes the ground. Modeled as a plastic collision, these impacts create discontinuities in the external force on the system. When solving these systems numerically, it is important to avoid stepping over these discontinuities since this can create convergence problems for the numerical method. To this end, we model systems such as these as hybrid systems [18] and treat the discontinuities and/or transitions explicitly.

Informally, a hybrid system consists of a set of regions (called "charts") upon each of which a dynamical system is defined, typically by an ODE or DAE. Charts are allowed to overlap, and may even belong to different spaces. Each chart V contains an open set U (called a "patch") whose boundary is contained within the union of the zero sets of a set of smooth scalar-valued functions (called "event functions"). It is assumed the closure of U is contained within V and that each event function is positive in U. A trajectory of the hybrid system starts with an initial point in some patch U in a chart V and evolves according to the dynamical system on V. This continues until the boundary of U is reached, at which point some event function g must be zero. Then a transition function is applied mapping that point to a new point in a new patch $\bar{U}$ in a new chart $\bar{V}$. It is assumed these transitions are instantaneous. The evolution then continues according to the dynamical system on $\bar{V}$. A periodic orbit in such a system is merely a trajectory that returns to its starting point after some nonzero time T.

For practical purposes, hybrid systems are implemented by including a discrete state s which determines which chart the hybrid system currently belongs to. All

functions defining the hybrid system in that chart (vector field, event functions, etc.) take this state as an additional argument. Derivatives with respect to this state are never computed, enabling our automatic differentiation algorithms to operate only on smooth functions. For the algorithms discussed below, we assume the ODE or DAE on each chart is analytic and the event functions and transition functions are C^1 on their domain of definition. In most cases of interest, these functions are at least piecewise analytic/smooth, and the assumptions can usually be satisfied by adding new charts to the hybrid system.

2 Taylor Series Integration

Moore [375,380] and Barton, Willers, and Zahar [23,24] implemented general Taylor series methods for computing solutions to ODE initial-value problems in the 1960s and 1970s, followed by work of Corliss and Chang [135] and Griewank et al. [229]. Guckenheimer and Meloon [244] extended these methods to solve boundary value problems for locating periodic orbits of ODEs. At each step of a numerical integration, a degree d truncated Taylor polynomial solution $x(t) = \sum_{k=0}^{d} x_k t^k$ is generated using the Taylor polynomial mode of automatic differentiation [225, 450]. In [244], Taylor series coefficients were generated using the ADOL-C package [229]. Here, we generate Taylor coefficients using a new package ADMC++ described in Sect. 5. Typically, we set $d = 40$. Step sizes are estimated by examining the growth rates of Taylor coefficients.

Several authors have extended the Taylor series technique to computing numerical solutions to initial-value problems in DAEs. Chang and Corliss [101] describe computing Taylor series solutions to DAEs representing simple mechanical systems. Pryce [441] and Nedialkov and Pryce [407] show how to compute Taylor series coefficients for arbitrary DAEs using Pryce's structural analysis [442]. Here we assume the DAE has been converted to an ODE on a constraint manifold:

$$\begin{aligned} \dot{x} &= f(x), \\ F(x) &= 0\ . \end{aligned} \tag{1}$$

This can be done either explicitly by providing formulas for f and F, semi-automatically using Pryce's structural analysis [442], or implicitly using automatic differentiation and knowledge of the structure of the DAE as is done for the mechanical system example in Sect. 6. Given a consistent initial condition x_0 such that $F(x_0) = 0$, an approximate Taylor polynomial solution $p(t)$ to the ODE initial-value problem $\dot{x} = f(x)$, $x(0) = x_0$ can be generated in the standard way, and a step size h computed as described above. While $F(p(h))$ will be quite small because of the high order of Taylor series methods, it will not be zero, in general. Moreover, this constraint error typically grows quadratically in time [249]. This can be remedied by simply projecting each time step back onto the constraints $F = 0$.

What distinguishes our work is the connection of this method to computing Taylor series solutions to reduced ODEs written in terms of local parameterizations of the constraint manifold $\mathcal{M} = \{x : F(x) = 0\}$. In particular, let $f : \mathbf{R}^n \to \mathbf{R}^n$ and $F : \mathbf{R}^n \to \mathbf{R}^m$ be analytic, $F(x_0) = 0$, and assume $\ker D_x F(x_0)$ has dimension m. Define $p = n - m$, then by the Implicit Function Theorem there are neighborhoods $\mathcal{A} \subset \mathbf{R}^p$ of $0 \in \mathbf{R}^p$ and $\mathcal{B} \subset \mathcal{M}$ of x_0 such that the mapping $\psi : \mathcal{A} \to \mathcal{B}$ defined implicitly by

$$x = \psi(y) : \begin{cases} U_0^T(x - x_0) - y = 0 \\ F(x) = 0 \end{cases} \tag{2}$$

is well-defined and analytic on $\mathcal{A}$. Here the columns of $U_0 \in \mathbf{R}^{n \times p}$ form an orthonormal basis for $\ker D_x F(x_0)$. Clearly $\psi(0) = x_0$ and $F(\psi(y)) = 0$ for each $y \in \mathcal{A}$. Such a mapping is referred to as tangent space projection in the literature [435, 436, 568] and derives from locally projecting the manifold onto its tangent space. It can be shown that the DAE (1) yields the ODE

$$\dot{y} = U_0^T f(\psi(y)), \quad y(0) = 0, \quad y \in \mathcal{A}\,. \tag{3}$$

Clearly the Taylor series coefficients $\{x_i\}$ and $\{y_i\}$ to the solutions to $\dot{x} = f(x)$, $x(0) = x_0$ and (3) are related by $y_i = U_0^T x_i$ for $i > 0$, and therefore it can be shown that one step of a Taylor series method applied to (3) is equivalent to computing the truncated Taylor polynomial solution to $\dot{x} = f(x)$, $x(0) = x_0$ followed by the projection given by solving

$$\begin{aligned} U_0^T(x - x(h)) &= 0 \\ F(x) &= 0\,. \end{aligned} \tag{4}$$

for x, where $x(h)$ is the truncated Taylor polynomial solution evaluated at $t = h$.

These techniques are easily extended to hybrid systems by looking for sign changes in all of the event functions defined for a given chart. If an event function g changes sign over one step of the integration, the time of the event can easily be found by applying Newton's method to the scalar equation $g(x(t)) = 0$. There are simple formulas for computing derivatives of the Taylor polynomial solution with respect to the initial conditions and model parameters [225, 433] that are important for the periodic orbit and parameter estimation techniques discussed in the following sections.

3 Periodic Orbits

Periodic orbits of a hybrid system of DAEs are trajectories that return to their starting point after some time T. Computationally, periodic orbits are described by boundary value problems, and robust methods for solving these problems fall into two general categories [12]: multiple shooting methods that use numerical integration to approximate the flow map of the system [145, 148, 383] and global methods that project the system onto parameterized sets of discrete curves [12, 151, 152]. Root finding techniques such as Newton's method are employed to solve the resulting set of algebraic equations in both cases. Here we couple the Taylor series integration technique discussed in Sect. 2 to a multiple-shooting framework to compute periodic orbits in hybrid systems of DAEs where each patch has the same dimension n. This requires formulation of regular systems of equations whose roots represent the periodic orbit. The periodic orbit is discretized by selecting a set of points, times and discrete states

$$D = \{(x_i, t_i, s_i), 0 \le i \le N\}$$

on the periodic orbit that satisfy two properties:

1. All of the points of the orbit on an event surface are included in D, and
2. $x_0 = x_N$.

In addition, we usually fix $t_0 = 0$. We denote by E the set of indices of the x_i which lie on event surfaces and by e the number of elements in E. The equations that characterize D as a discrete closed orbit are then

$$\Phi(x_{i-1}, t_i - t_{i-1}) = x_i, \qquad i-1, i \notin E\ , \tag{5a}$$

$$\Phi(h(x_{i-1}), t_i - t_{i-1}) = x_i, \qquad i-1 \in E, i \notin E\ , \tag{5b}$$

$$\Phi(x_{i-1}, t_i - t_{i-1}) = x_i, \quad g(x_i) = 0, \quad i-1 \notin E, i \in E\ , \tag{5c}$$

$$\Phi(h(x_{i-1}), t_i - t_{i-1}) = x_i, \quad g(x_i) = 0, \qquad i-1, i \in E\ , \tag{5d}$$

where h is the transition function applied to x_{i-1} and g is the event function that vanishes at x_i. In writing these equations, we have suppressed the changes of discrete state that take place at transitions and use the same symbol Φ to denote the flows on the patches containing the trajectory segments. If the system is a DAE written as an ODE on a manifold, we also constrain the mesh points to lie on the constraint manifold and only enforce the shooting equations above in the tangent space to the manifold. For example, if $i-1, i \notin E$, then (5a) is replaced by

$$U_i^T(\Phi(x_{i-1}, t_i - t_{i-1}) - x_i) = 0, \quad F(x_i) = 0\ ,$$

where the columns of U_i form an orthonormal basis for $\ker D_x F(x_i)$.

These equations are underdetermined if there are indices that do not lie in E. The location of the corresponding points on their trajectories has one degree of freedom that is not fixed by the equation $\Phi(x_{i-1}, t_i - t_{i-1}) = x_i$ since, given (x_{i-1}, t_{i-1}), this consists of n equations for the $n+1$ variables (x_i, t_i). Altogether, with $x_0 = x_N$ and t_0 fixed, there are $nN + e$ equations in the $(n+1)N$ variables. For a hyperbolic periodic orbit, these equations are a regular system defining a smooth manifold P of dimension $N - e$ [433]. We accept this fact and use a version of Newton's method that is suitable for computing points on P, exploiting the fact that we "know" the tangent space to P. Moving the point (x_i, t_i) infinitesimally along its orbit yields a tangent vector to P that has components $f(x_i)$ and 1 in the appropriate locations of D. Insisting that the Newton updates be orthogonal to these vectors yields a regular system of equations to be solved for the updates. This strategy subsumes the definition of an explicit "phase condition" in the case of a system of ODEs that is not hybrid. The regular system of equations can be viewed as defining a residual function R whose roots, obtained via Newton's method, represent periodic orbits. Jacobian derivatives of R required by Newton's method are computed with the methods mentioned in the previous section.

This constitutes a "bare-bones" multiple shooting solver for periodic orbits of a hybrid system. The sequences of events along the periodic orbit to be calculated are specified in advance, and no attempt is made to modify these in the search for a periodic orbit. Similarly the number of mesh points is fixed, and there is no attempt to adapt the mesh to improve the condition number of the Jacobians for Newton's method.

4 Parameter Estimation

We now present an optimization method for estimating parameters for periodic orbit data. For simplicity, we restrict our attention to systems of autonomous ODEs $\dot{x} = f(x, \lambda)$. The extension to hybrid systems is straightforward. Here $\lambda \in \mathbf{R}^p$ is a set of free parameters we wish to vary in order to find a "best fit" of a periodic orbit to empirical data. The method is based upon an objective function that measures the distance between closed curves. We apply trust-region based optimization methods to compute a local minimum of the objective function over the space of free parameters λ. We assume the parameters are restricted to a region in which there is a family of periodic orbits that depend smoothly upon the parameters and the minimum occurs in the interior of this region (making this an unconstrained minimization problem).

Assume a time-series $D_r = [(x_0, t_0), \ldots, (x_N, t_N)]$ representing a reference periodic orbit is provided (e.g., from empirical data), with period $t_N = T_r$, and a discrete orbit $D_c = [(y_0, s_0), \ldots, (y_N, s_N)]$, with period $s_N = T_c$, representing an orbit in the model is computed via the periodic orbit algorithms discussed above. We assume the relative phase offset of the orbits is zero (i.e., $f(y_0)^T(x_0 - y_0) = 0$) and that the orbit mesh points have been computed at the same scaled times, $\bar{t}_i \equiv t_i/T_r = s_i/T_c$, $i = 0, \ldots, N$. We define the distance between the two discrete orbits as

$$d(\lambda) = \sum_{i=1}^{N-1} \|x_i(\bar{t}_i) - y_i(\bar{t}_i, \lambda)\|_2^2 \ (\bar{t}_i - \bar{t}_{i-1}) + \left(\log\left(\frac{T_r}{T_c(\lambda)}\right)\right)^2 , \tag{6}$$

where the dependence on the free parameters λ has been made explicit. The first term in this formula is a Riemann approximation to the L_2 distance between the orbits, and the second term takes into account the discrepancy between the periods. Even though the scaled times $\bar{t}_i$ are fixed by the data, the times of the mesh points for the computed orbit are allowed to vary during the periodic orbit computation. The value of the orbit at the fixed times is then re-computed by Taylor series integration, allowing the derivative of $d(\lambda)$ to be computed directly from the defining periodic orbit equations presented in Sect. 3 using AD (see [99] for further details). While it would also be possible to compute the second derivative $\nabla^2 d(\lambda)$ analytically using AD, we found a finite-difference approximation by differencing $\nabla d(\lambda)$ to be sufficient to investigate the feasibility of these algorithms.

With the objective function $d(\lambda)$ in hand, we applied trust-region minimization algorithms to find a best fit for the free parameters λ. Trust-region methods are a powerful class of Newton-like methods for solving unconstrained minimization problems, and use a quadratic model for the objective function, but constrain each iterate to stay in some local neighborhood of the previous iterate. We implemented a method called the hook step (or "optimal" step) method [260,381,484]. We followed the algorithms presented in [143] with minor adjustments to make the algorithm less likely to decrease the trust-region radius [99]. Convergence criteria were based on the relative gradient [143] of d at λ defined by

$$\text{relgrad}(\lambda)_i = \frac{\nabla d(\lambda)_i \lambda_i}{d(\lambda)} , \tag{7}$$

where the subscript i indicates the i^{th} component.

5 Software

Implementations of Taylor series integration, periodic orbit location, and parameter estimation algorithms rely heavily on automatic differentiation to compute derivatives of the underlying equations quickly and accurately. We required an AD package that provides forward, reverse, and Taylor polynomial mode derivative calculations of matrix-valued functions and chose MATLAB® as the framework for implementing these algorithms. We sought run times roughly equivalent to hand-coding the corresponding derivative calculations in MATLAB itself. For mechanics problems, we also required at least third-degree tensor derivatives of Taylor polynomial coefficients using both the forward and reverse mode. At the time of this work, none of the existing AD packages satisfied all of these requirements. In particular, none of the publicly available MATLAB-based AD tools including ADiMat [51, 53, 62] and MAD [181] implement Taylor series computations. We therefore created a custom AD package named ADMC++ to implement the required derivative computations.

ADMC++ is an operator overloading-based AD package for differentiating matrix-valued functions written in MATLAB. For efficiency reasons, all derivative computations are carried out externally to MATLAB in compiled object code originally written in C++ using the MATLAB MEX® interface. A MATLAB class **amatrix** is provided which overloads many of the matrix-level MATLAB intrinsic functions, and by evaluating a function on **amatrix** objects, a computational trace [225] is generated representing the expression graph of the function. The trace data structure is not stored in the MATLAB workspace, but rather is created in external memory through the MEX interface. Once a trace has been generated, derivatives are computed by looping through the trace in either the forward or reverse directions, and C++ classes are provided for tangent, adjoint, and Taylor polynomial derivatives. Tangent and adjoint computations may themselves be traced, allowing the computation of arbitrarily high-degree tensor derivatives in a manner similar to FADBAD/TADIFF [34].

By taking the MATLAB interpreter out of the forward and reverse sweeps of the trace, we are able to improve performance for these derivative computations, especially for the Taylor polynomial calculations that cannot be completely vectorized to eliminate MATLAB loops. This requires us to evaluate and differentiate each supported MATLAB intrinsic. Given the very large number of possibly differentiable MATLAB intrinsics, this is an arduous task indeed, and only a limited number of operations are currently supported.

A Taylor series integration package TSINT and multiple-shooting periodic orbit package TSPO have also been written that implement the algorithms discussed above for a wide variety of ODEs, DAEs and hybrid systems. These packages are written entirely in MATLAB and are dependent upon ADMC++ for all required derivative computations. Further details on these packages can be found elsewhere [433].

6 Applications

Our first application of these techniques is a periodic orbit location problem in a rigid-body mechanical system. The disc-foot passive walker [131, 194, 363] sketched

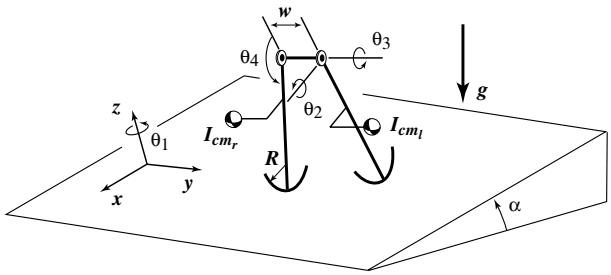


Fig. 1. Schematic diagram of the disc-foot passive walker. Drawing based on a diagram of the McGeer walker given in [194].

in Fig. 1 serves as a simple model of bipedal walking. It consists of two rigid body legs with unit mass and length connected at the hip by frictionless pin joints and separated by a distance w. The foot of each leg consists of a thin disc of radius R. The walker is placed on a flat plane inclined by an angle α from horizontal. It is assumed one leg is in contact with the ground at all times (the stance leg), while the other swings freely (the swing leg). The foot of the stance leg is in rolling contact with the ground at all times. The only external force on the system other than the contact forces at the stance foot contact point is gravity. Stable walking motions in these passive machines shed light on the ability of humans to walk in a stable manner.

Instantaneously, the system has four degrees of freedom: three rotation angles of the stance leg around the contact point and one rotation angle of the swing leg around the hip axis. Deriving ODE equations of motion of the system in terms of these four angles is straightforward, yet algebraically is quite complicated. Our goal was to see how much automatic differentiation could simplify the process of generating mechanically correct equations of motion, but still be able to compute periodic motions of the system to an equivalent level of accuracy as could be obtained from the original ODE system. As discussed in the introduction, the equations of motion of the system are most easily written as a set of differential-algebraic equations. While there are a wide variety of DAE formulations for constrained mechanical systems, we chose the Euler-Lagrange formulation [221]

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{x}} - \frac{\partial L}{\partial x} = -G^T(x)\lambda - D_x h(x)^T \mu \tag{8a}$$

$$G(x)\dot{x} + \tilde{g}(x) = 0 \tag{8b}$$

$$h(x) = 0 \tag{8c}$$

for its simplicity and wide-spread familiarity. Here L is the Lagrangian (kinetic minus potential energy) and x represents the set of generalized coordinates for the system consisting of three coordinates for the center of mass position of each leg and three Euler angles of rotation around the center of mass for each leg (a total of 12 coordinates). The rolling contact of the stance leg on the ramp gives two velocity constraints (8b) by requiring the instantaneous velocity of the stance leg contact point be zero. The pin joint at the hip and the requirement that the bottom of the stance foot lie on the ramp gives a total of six position constraints (8c). The

quantities λ and μ in (8a) are undetermined multipliers that must be computed along with the solution. Given algebraic formulas for the Lagrangian L, position constraint function h, and velocity constraint function $g(x, \dot{x}) = G(x)\dot{x} + \tilde{g}(x)$, automatic differentiation is used to compute the necessary derivatives appearing in the DAE, drastically simplifying the amount of programming effort required to compute them. AD is also used to differentiate the constraints to convert the system to an ODE on a manifold. The system is clearly hybrid with two states. Each chart has one event function given by the height of the swing foot above the ramp, and one transition function. The transition function is derived by considering angular momentum conservation around the new contact point [131, 194, 433].

We compared the DAE system with a MATLAB ODE model written by Garcia [193]. The DAE system consisted of approximately 60 lines of MATLAB code whereas the ODE model had approximately 240 lines and was much harder to derive and verify. To compare the accuracy of these methods, the periodic orbit algorithms using Taylor series integration discussed in Sect. 3 were applied using parameters and initial conditions found in [131]. At these parameter values, the system has a stable periodic orbit (Fig. 2). For both systems, three mesh points were used in the periodic orbit solver (one for each transition plus one mesh point not lying on an event surface). An error tolerance of 1.0e-16 was used in the Taylor series integration, along with an event solver tolerance of 1.0e-15. For the DAE calculation, the projection solver tolerance was set to 5.0e-16. For the DAE system, the initial residual of the periodic orbit equations was 6.9e-5 and two Newton iterations were required to reduce the residual to 1.3e-15. For the ODE system, the initial residual was 3.5e-5 and also took two Newton iterations to reduce the residual to 1.8e-15. The DAE calculation took approximately 12 times longer than the ODE calculations. The L_2 distance between these orbits was calculated as discussed in Sect. 4 and was found to be approximately 3.9e-14. The eigenvalues of the return map for both the ODE and DAE periodic orbits were also calculated. The largest difference between eigenvalues was found to be approximately 7.6e-14 with the magnitude of the largest eigenvalue equal to 0.70418256213669 (ODE). This agrees with the eigenvalue of $(0.8391560)^2$ given in [131] to 2.4e-7, very near the expected error provided in [131].

These results show the DAE Taylor series periodic orbit method has nearly the same accuracy as the ODE method. We reiterate that the goal of this project was to simplify the generation of the equations of motion yet still obtain numerical results with the same level of accuracy as could be obtained from an ODE model. One would expect the DAE method to be slower given the large amount of automatic differentiation that is used at each time step to evaluate the DAE equations of motion. Given that the DAE model took drastically less time to derive and verify, this increased computational cost when amortized across the process of deriving a model leads to a significantly more efficient framework for studying many mechanical systems. Also, Fig. 2(d) demonstrates the ability of the Taylor series integration technique presented in Sect. 2 to accurately satisfy the algebraic constraints on the system. Finally, these results provide strong independent verification of the results in [131] regarding the existence of a stable walking motion.

We next apply the techniques described in Sect. 4 to the Hodgkin-Huxley model [265]. The Hodgkin-Huxley (H-H) equations model electrical excitability of squid axon and are the archetype of conductance-based models of neural oscillations. They constitute a four-dimensional vector field with several parameters that produces periodic oscillations in some parameter regimes. We used the H-H

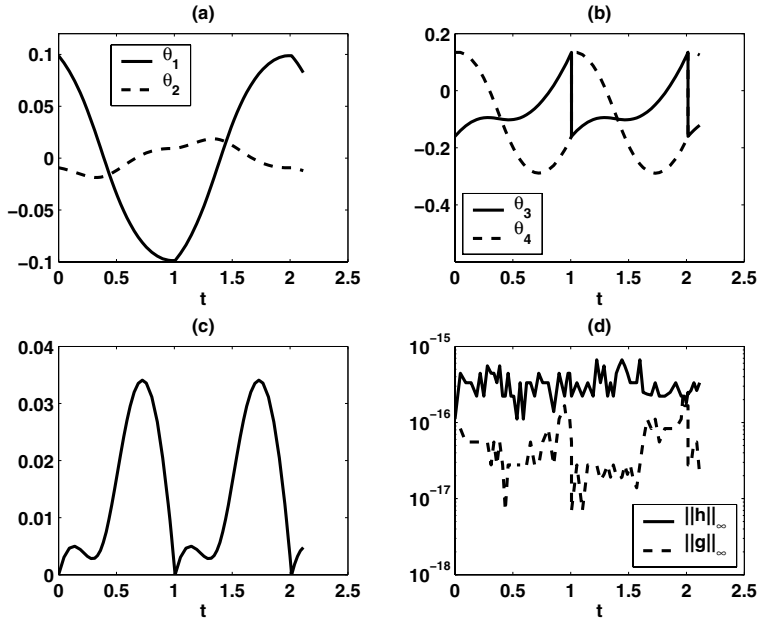


Fig. 2. Periodic trajectory of the disc foot walker for one complete step of the system (two strides of the walker). (**a**) Steer angle θ_1 and lean angle θ_2. (**b**) Stance leg angle θ_3 and swing leg angle θ_4 measured from the ramp normal. (**c**) Swing foot height. (**d**) Constraint error for DAE solution.

equations as a test-case for the parameter estimation algorithm described earlier, using the maximal sodium conductance $\bar{g}_{\mathrm{Na}}$, maximal potassium conductance $\bar{g}_K$, and injected current I_{ext} as active parameters. Their "standard values" are $(\bar{g}_{\mathrm{Na}}, \bar{g}_K, I_{\mathrm{ext}}) = (120, 36, -20)$. We began by fitting the Hodgkin-Huxley model to an ideal reference orbit generated using the H-H equations at perturbed parameters values $(140, 36, -20)$. This gave a reference orbit with period $T_r = 11.2082849$ ms. We took an approximation D_r to this reference orbit with $N = 30$ mesh points, and used the trust-region algorithm to look for an optimal fit of the Hodgkin-Huxley model to D_r, starting from the standard H-H parameter values. These parameter values give a starting D_c with $T_c = 14.574003$. The iteration converged in 8 steps, computing an optimal value of $d(\lambda_*) = 1.2483041\text{e-}12$ and $\|\mathrm{relgrad}(\lambda_*)\|_\infty = 7.9200748\text{e-}10$, with parameters $\lambda_* = (139.99986, 35.999964, -19.999989)$. and period $T_* = 11.2082847966$. These values are very close to the reference values, $(140, 36, -20)$.

To examine the effects of noise in the reference orbit on the convergence of the parameter estimation algorithm, we fixed the starting data at the standard Hodgkin-Huxley parameter values and added Gaussian noise to the reference data. For each run, we replaced the voltage time-series $\{V_i^r\}_{i=1}^N$ for the reference orbit with $\{N(V_i^r, \sigma)\}_{i=1}^N$, where $N(\mu, \sigma)$ is the normal distribution with mean μ and standard deviation σ.

Table 1. Results for noisy reference data with increasing variance. The computed minimum distance $d(\lambda_*)$, the norm of the relative gradient and optimal values found for the active parameters are shown at each variance σ^2.

σ^2	$d(\lambda_*)$	$\|\text{relgrad}(\lambda_*)\|_\infty$	$\bar{g}_{\text{Na}*}$	$\bar{g}_{K*}$	$I_{\text{ext}*}$
0	1.2483e-12	7.9201e-10	1.4000e+2	3.6000e+1	-2.0000e+1
0.1	7.6301e-3	3.3220e-9	1.3767e+2	3.5362e+1	-1.9793e+1
0.5	3.8202e-2	6.8566e-10	1.3505e+2	3.4650e+1	-1.9563e+1
2.0	1.5318e-1	1.4228e-5	1.3098e+2	3.3551e+1	-1.9213e+1
4.0	3.0699e-1	1.5196e-6	1.2814e+2	3.2792e+1	-1.8977e+1
16.0	1.2361	6.6402e-8	1.2110e+2	3.0973e+1	-1.8448e+1

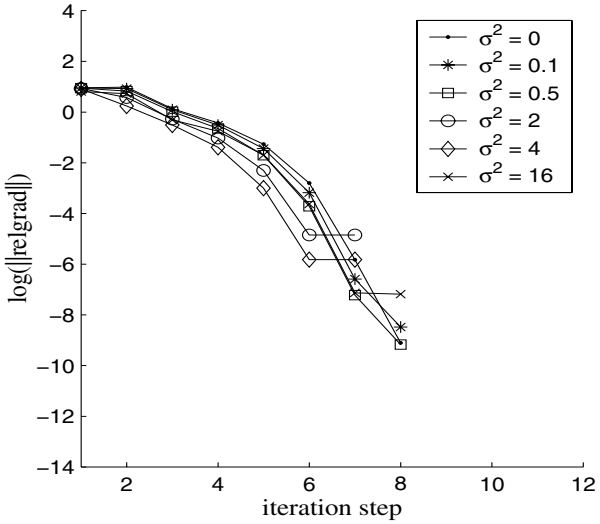


Fig. 3. Effects of noisy D_r on convergence to zeros of ∇d. The trust-region algorithm is seen to converge to local minima even for large variance noise in D_r. See the output in Table 1 for more details.

As the variance of the noise increased, the algorithm still converged to a minimum, but the minimum is increasingly farther away from the noise-free minimum, both in terms of optimal parameter values and the minimum value of $d(\lambda)$ achieved. The results of these computations are summarized in Table 1 and Fig. 3.

Even for large variance, the convergence of the trust-region algorithm is indicated by the small values of relgrad(λ_*). For example, with $\sigma = 4$ we found a local minimum with $d(\lambda_*) = 1.2361$, $\|\text{relgrad}(\lambda_*)\|_\infty = 6.6402\text{e-}8$ and optimal parameter values $\lambda_* = (121.10, 30.973, -18.448)$. These results indicate that the optimization algorithm is robust with respect to noise in the reference data: a local minimum of the objective function is still found with a small value for $\|\text{relgrad}\|_\infty$, indicating good convergence. Moreover, the parameters for the minima with noisy data approach those for the noise-free reference data as $\sigma \to 0$.

7 Conclusions

Taylor series integration and automatic differentiation provide a powerful set of tools for computing periodic orbits in hybrid systems of ODEs and DAEs, and for parameter estimation to fit these orbits to data. Taylor series integration allows numerical trajectories to be computed to high accuracy with large step sizes and provides dense output for accurate event location. Through the use of tangent space parameterization, the standard Taylor series algorithm for ODEs has a simple extension to DAEs formulated as an ODE on a constraint manifold. Furthermore, AD can be employed to simplify the conversion of the DAE to an ODE on a manifold, and as shown in the bipedal walking example, simplify the derivation of the governing DAE itself. Taylor series integration coupled with AD provides a simple mechanism for computing the derivative of the numerical flow of the ODE or DAE with respect to initial conditions and model parameters. These properties, coupled with a simple multiple-shooting framework allow the accurate computation of periodic orbits using very coarse discretizations in an efficient manner. Computing these orbits accurately is critical for further analysis such as parameter estimation, since loss in accuracy degrades the performance of Gauss-Newton optimization algorithms. Maintaining accuracy in the periodic orbit computation is necessary to ensure smoothness of the objective function and the amenability of Newton-based optimization methods to these parameter estimation problems.

A new AD library ADMC++ was presented, facilitating the derivative and Taylor polynomial calculations required to implement these algorithms. The library provides the forward, reverse, and Taylor polynomial automatic differentiation modes for functions written in MATLAB, but performs all derivative calculations in compiled object code for efficiency. All three modes can be combined to produce arbitrarily high-degree tensor derivatives of Taylor polynomial coefficients.

Implementation of Automatic Differentiation Tools for Multicriteria IMRT Optimization

Kyung-Wook Jee, Daniel L. McShan, and Benedick A. Fraass

Department of Radiation Oncology, The University of Michigan Medical School, Ann Arbor MI
kwjee@umich.edu

Summary. Automatic differentiation tools (ADOL-C) have been implemented for large-scale NLP optimization problems encountered in an advanced radiotherapy technique called Intensity Modulated Radiation Therapy (IMRT). Since IMRT treatments involve many tissue structures and their associated clinical objectives, the corresponding optimization problems are typically multi-objective. In this study, they are solved by a multi-criteria approach called Lexicographic Ordering. This approach allows clinical objectives to be categorized into several priorities or levels, and optimization is performed sequentially in order of priority while keeping the previously optimized results constrained. As a result, the feasible solution region is gradually reduced as the method progresses. For each level of optimization, the objective function and constraints are constructed interactively by a treatment planner and the corresponding Jacobian is provided by AD tools at a machine-precision level. Results indicate that a high degree of accuracy for Jacobian is essential to produce both feasible and optimal results for clinical IMRT optimization problems.

Key words: IMRT optimization, multicriteria lexicographic optimization, treatment planning

1 Introduction

Intensity Modulated Radiation Therapy (IMRT) is one of the most technically advanced treatment methods in external beam radiation therapy. Unlike conventional methods, IMRT delivers a sequence of radiation beams that are effectively broken into hundreds of pieces (i.e., beamlets) with each beamlet having different intensities. The ability to manipulate the intensities of individual beamlets permits a greatly increased degree of control over radiation fluence that enters the patient's body. As a result, IMRT can deliver more conformal and optimal radiation doses to the target volumes, leading to higher tumor control and decreased toxicity to healthy tissues [70, 373, 529].

The planning process for IMRT treatments, which is also referred to as the inverse planning, involves predetermination of a wide range of control parameters. Some parameters such as the number of beams and their corresponding energies and

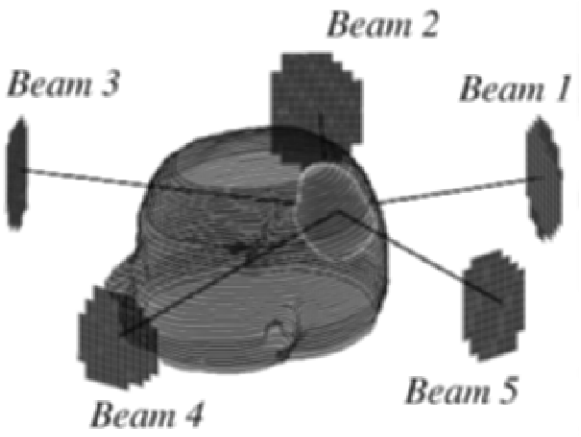


Fig. 1. Illustration of IMRT beam arrangement for a brain tumor case. The figure shows a total of 5 beams directed to the tumor volume and each beam is conceptually divided into many pieces called beamlets.

angles are manually chosen by a human planner based on prior experiences. Other parameters such as beamlet patterns are determined by an automated optimization system. Whether chosen manually or automatically, all of the control parameters are selected to satisfy clinical treatment goals associated with an individual patient. (This study is limited to the automatic determination of beamlet patterns.)

Factors that affect the optimal treatment result are quantified based on a radiation dose distribution and its radiobiological effects to tumor or surrounding healthy tissue structures. Deviations of such factors (i.e., planning criteria) from their desirable values typically constitute optimization problems and therefore are minimized during the inverse planning. The mathematical relations between the control parameters (optimization variables) and the objective function characterize the search space. The functional relations determine an appropriate choice for search algorithm. (In the case of beamlet optimization, beamlet intensities comprise the optimization variables.)

IMRT optimization problems are inherently multicriteria since they involve multiple planning objectives for many different tissue structures. For example, a good treatment planning solution can be characterized by minimal extents of both underdosing and over-dosing for tumor and radiation-sensitive structures, respectively. Our standard practice of solving these multicriteria problems has been a weighted-sum approach, in which a single function is formulated by a positively weighted sum of original objective functions and minimized by a conventional search algorithm [305, 529]. Although this approach provides a straightforward means of simplifying complex multicriteria problems, the task of choosing a set of weights that properly represents clinicians' preferences on individual objectives is not always trivial, therefore requiring multiple trial-and-errors before finding an acceptable solution. To improve this iterative process of preference articulation, a multicriteria

Table 1. Examples of planning criteria [305]. Dosimetric parameters are widely used in IMRT planning. Biological model-based parameters are gradually accepted as more clinical data supporting the models become available.

Direct Dose-Based Criteria

Min	Minimum dose to structure
Max	Maximum dose to structure
Mean	Mean dose to structure
StDev	Standard deviation of dose distribution in structure
DVHpoint	% Vol. of the structure above the specified dose D_t
Threshold	$\sum_{i \in \text{structure}} (\max(0, d_i - D_i))^p$, where d_i is point dose in the structure, and p is a user-specified parameter
DoseLSQ	$\sum_{i \in \text{structure}} (d_i - D_i)^p$

Biological Model-Based Criteria

NTCP	Normal tissue complication probability
TCP	Tumor control probability
EUD	Equivalent uniform dose
V_{eff}	Effective volume

strategy called lexicographic ordering [9, 367, 491] (also known as preemptive approach in goal programming) has been implemented.

A successful implementation of the lexicographic strategy for IMRT problems requires two important components: an efficient search algorithm that can handle nonlinear constraints and a flexible method of calculating a high-quality Jacobian for the search algorithm. This requirement becomes more stringent with an additional prerequisite of scalability since IMRT problems are generally large scale having thousands of input variables (beamlets). This study attempts to achieve both the numerical reliability and computational efficiency for the lexicographic method by implementing an industry-leading, large-scale NLP algorithm called Sequential Quadratic Programming (SQP) [214] and a highly versatile automatic differentiation tool called ADOL-C [229].

2 Methods and Materials

2.1 Lexicographic Approach

The term lexicographic refers to the process by which words are listed in alphabetical order as in a dictionary. In a similar fashion, lexicographic ordering in multicriteria optimization problems refers to a strategy that prioritizes objectives and solves them

in order of importance. The lexicographic method involves solving a sequence of optimization problems rather than solving a single, scalarized function at once [367]. The first step in this method is to categorize the multiple objectives into different levels of importance – i.e., the highest level being the most important objectives to be achieved. Then, a gradient-based search algorithm is used to solve one level of optimization problem after another, beginning from the top level. While the method progresses down, the preceding objective functions are converted into hard inequality constraints for subsequent levels:

$$\begin{aligned}&\text{Find } \mathbf{x} \in \mathbf{R}^n \text{ that minimizes } F_i(\mathbf{x})\\&\text{subject to } F_j(\mathbf{x}) \leq F_j\left(\mathbf{x}_j^*\right),\\&\text{where } j = 1, 2, \ldots, i-1;\ i > 1 \text{ and } i = 1, 2, \ldots, k\,,\end{aligned}$$

where the symbol $\mathbf{x}_j^*$ denotes the optimum of the jth objective function, found in the jth iteration. As the method proceeds, the number of constraints grows, causing the feasible solution space to reduce until the optimal solution is found.

2.2 Implementation

The lexicographic method generally reflects the mental process often used to make a decision in decision-making problems. Hence, this method tends to be more intuitive and interactive to a planner, particularly when choosing priorities for planning objectives. In this implementation (illustrated in Fig. 2), a graphical user interface (designed with Advanced Visual Systems libraries) is used to facilitate the interactive construction and prioritization of objective functions.

If planning metrics that are based on nonlinear functions are used, a non-linearly (NL) constrained optimization problem is presented for each level. This problem is solved efficiently by the SQP method. The general idea of SQP is to model (or approximate) NL-constrained problems by quadratic programming (QP) sub-problems

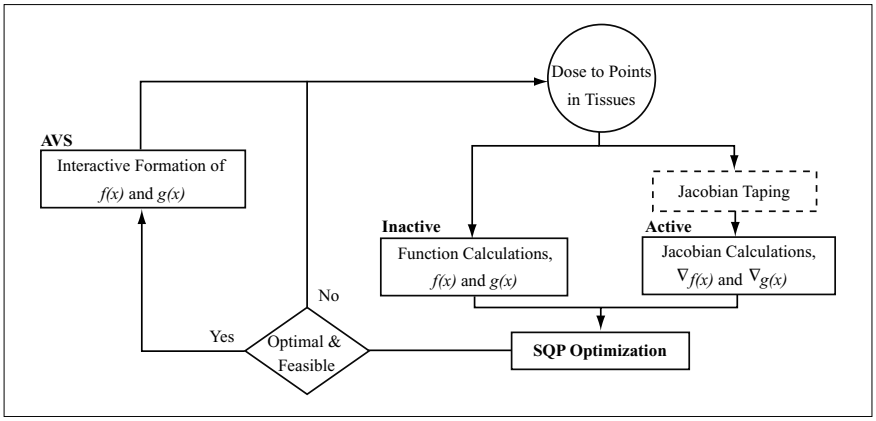


Fig. 2. Illustrating of the Lexicographic implementation for IMRT optimization.

and to define search directions based on the solutions of the sub-problems. The nonlinear constraints are also approximated by linearization. At each iteration, an augmented Lagrangian merit function is reduced along each search direction to ensure first-order convergence from any starting point. In the present implementation (SNOPT) [214], the QP sub-problems are solved using a reduced-Hessian algorithm. The Hessian of the Lagrangian is approximated by a limited-memory quasi-Newton method, and therefore calculations for only the corresponding Jacobians are explicitly required. The SQP method is expected to show a particularly good performance for problems having significant non-linearities and a large number of optimization variables - characteristics shown by IMRT optimization problems.

For each level of optimization, the Jacobian for the Lagrangian is calculated by an automatic differentiation tool, ADOL-C [229]. This tool keeps (or tapes) a record of the elementary computations made available during the evaluation of the objective and constraint functions at a given point and reviews the recorded information to produce the corresponding derivatives based on the chain rule. For the IMRT optimization problems, the reverse mode of gradient calculation is used due to the large number of the independent variables in comparison with the number of dependent variables. Since the reverse mode requires storing of the entire execution trace of the original functions, the amount of memory requirement for taping is large, which consequently deteriorates the overall performance of the gradient calculations. In this implementation, this storage burden is reduced by excluding the matrix multiplication process (used for the dose calculation) from gradient taping. The tape recording begins from the dose distribution (active independent variables) up to the objective and constraint values (active dependent variables). Therefore, ADOL-C computes the Jacobian with respect to the dose, and a separate chain rule is applied to the computed Jacobian to account for the matrix multiplication as in Fig. 3 This technique is generally described as *interface contraction* in [275].

Doses to m calculation points by n beamlet intensities:

$$\mathbf{D}_{m\times 1} = \mathbf{A}_{m\times n} \cdot \mathbf{X}_{n\times 1}$$

ADOL-C tape computes: $\nabla f(d), \nabla g_k(d)$

Chain rule is applied to obtain the Jacobian for SQP:

$$\nabla f(x) = \mathbf{A}^T_{m\times n} \cdot \nabla f(d)$$

$$\nabla g(x) = \mathbf{A}^T_{m\times n} \cdot \nabla g(d)$$

Fig. 3. Jacobian calculation using ADOL-C

Typically in IMRT problems, function evaluations are very expensive because time-consuming processes are required for the dose computation to multiple tissue structures. This excluded an option of using the Finite Difference method for the Jacobian calculations.

2.3 Clinical IMRT Case (Brain Tumor)

The performance of the lexicographic method for IMRT problems is demonstrated by using a brain case based on an in-house dose escalation protocol [504].

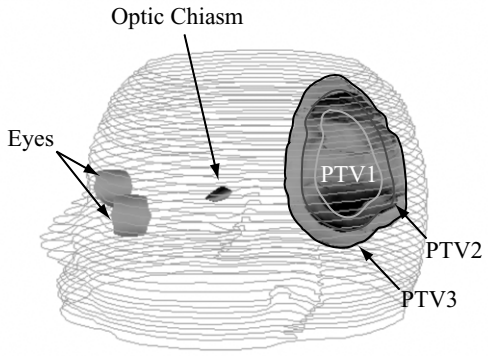


Fig. 4. Illustration of the tissue structures for a brain tumor patient.

A brain tumor is a group of abnormal cells that grows in or around the brain. Brain tumors are classified based on where the tumor is located, the type of tissue involved, whether the tumor is benign or malignant, other factors. When radiation therapy is recommended to treat a brain tumor patient, the planning procedure typically begins with anatomical segmentation of the actual tumor volume [referred to the gross tumor volume (GTV)] and other healthy tissue structures that are involved in the treatment. The segmentation is based on the three-dimensional patient dataset provided by the modern medical imaging modalities such as CT and MRI. In the brain dose escalation protocol, three concentric volumes are defined as the planning target volumes[1] (PTV1, PTV2 and PTV3 as illustrated in Fig. 4). The PTV1 represents the GTV with a uniform, isotropic expansion of 0.5 cm. This expansion accounts for a) the potential spread of cancer cells adjacent to the gross tumor and b) the random setup error associated with patient positioning to a treatment table. The PTV2 and PTV3 represent the GTV expanded by 1.5 cm and 2.5 cm, respectively. The dosimetric goals for the PTV1 are the highest in terms of the amount of the prescribed dose and its uniformity. These goals are gradually lowered for the larger expanded target volumes. As a result, a radiation dose distribution that conforms tightly and focuses increasingly to the core tumor body can be designed. In addition, the brain protocol prescribes that radiation sensitive critical organs (such as the optic chiasm) should be protected by minimizing the delivered dose under the maximum tolerable limits.

For this demonstration, an IMRT plan is prepared using a 5-axial beam arrangement having 500 beamlets and 6 MV photon beams. Approximately 53,000 dose-calculation points were randomly distributed throughout the structures. The specific

[1] The planning target volume (PTV) is a three-dimensional representation of the irradiation target volume on which the treatment goals are defined.

planning objectives and their relative priorities used for the lexicographic method are summarized in Table 2.

The planning goals for the three target volumes and for the chiasm are completely or best achieved by using the dose-volume histogram (DVH)-based objective functions. In addition, the objective of lowering the dose to the normal tissue is achieved by minimizing the mean dose objective function.

Table 2. IMRT planning objectives for brain case [504].

1^{st} Level (Highest Priority)	• Dose to the entire target volumes (PTV1, 2, and 3) should be greater than 85.5, 70, and 60 Gy. • Dose to the entire chiasm should be less than 10 Gy.
2^{nd} Level	• Make dose to the three target volumes as uniform as possible. Specifically, dose to the entire target volume (PTV1, 2, and 3) should be less than 94.5, 80, and 70 Gy.
3^{rd} Level (Lowest Priority)	• Minimize dose given to the normal tissue.

3 Results

3.1 Lexicographic Planning

The planning criteria used in this brain case involve evaluation metrics that are based on direct interpolations of the dose distributions delivered to the tissue structures. Hence, the results obtained from each level of optimization can be effectively summarized by cumulative histograms of the dose distributions (called dose-volume histogram). In addition, all of the objective functions used in this study were designed to be positive functions. Thus, the best possible numeric value after minimization is always zero - i.e., an objective function value of zero indicates a complete achievement of the associated planning goal.

Prior to each level of optimization, the input variables (i.e., the beamlet intensities) are initialized by random values chosen between the minimum and maximum boundaries. A typical result after the initialization is shown in Fig. 5. Five curves correspond to histograms for five tissue structures. Even with the random inputs, the figure show that the target volumes receive higher dose than the non-target structures since all the beams are shaped and arranged in such a way that they tightly conform to the geometries of the targets.

For the 1st level of optimization, a total of four objectives were identified to have the highest priority (as summarized in Table 2). Those are the ones that specify prevention of under-dosing and over-dosing to the three targets (PTVs) and the optic chiasm, respectively. The corresponding four objective functions were constructed and summed together each with unity weight, resulting in a single scalar function for the minimization process. For this level, a quasi-Newton method (instead of SQP) was used to minimize the function due to the absence of constraints. The final values for the four objective functions were found to be all zero, indicating that all goals

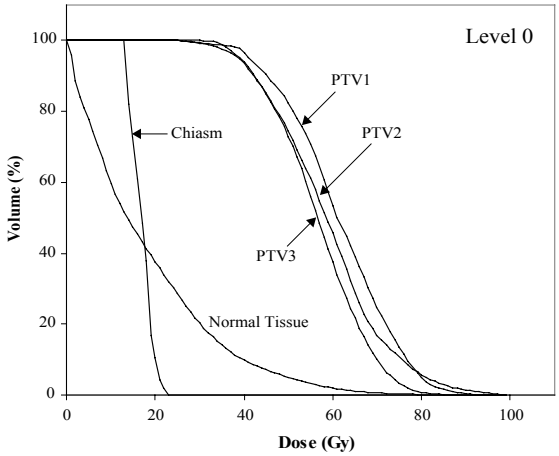


Fig. 5. DVHs prior to optimization where the initial intensity pattern of beamlets was chosen randomly between 0 to 25 in this example.

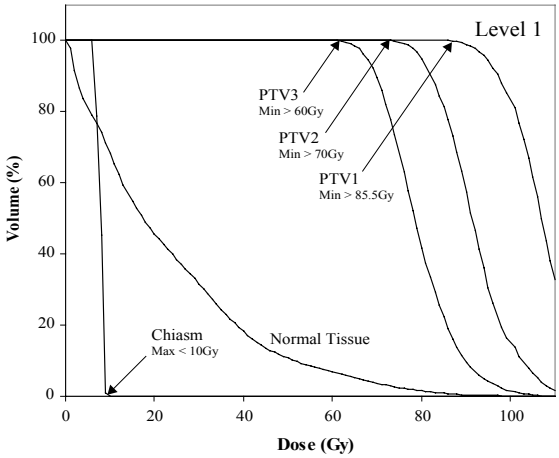


Fig. 6. DVH after the 1st level optimization.

were achieved as shown in Fig. 6. Then, these four functions were turned into four individual inequality constraints and the next set of objectives was identified for the following level.

The 2nd level objectives concern the uniformity of the target dose distributions. Desirable goal levels were set to be 94.5, 80, and 70 Gy of maximum dose for PTV1, PTV2, and PTV3, respectively. Optimization began with random initialization for input beamlets, and this nonlinearly constrained problem was solved by SQP. Figure 6 clearly shows that the previously optimized results are kept tightly constrained while minimizing the object function in the 2nd level. The final optimal values were found to be small but non-zero, indicating that the original goal levels were set too

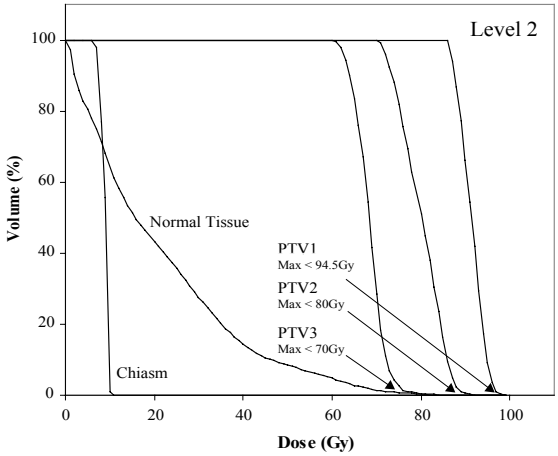


Fig. 7. DVH after the 2nd level optimization.

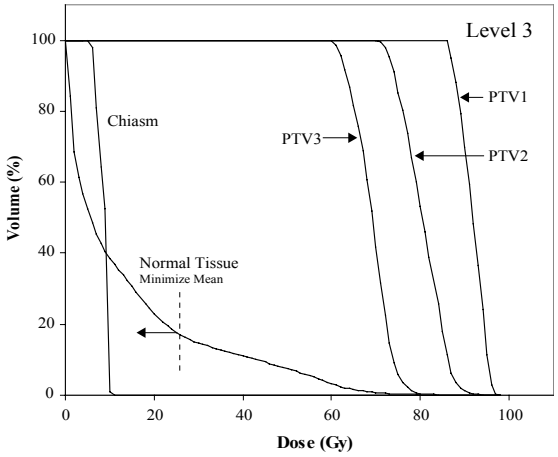


Fig. 8. DVH after the 3rd level optimization. (Final level)

high. However, it was observed that these best achieved values for the maximum doses (for PTV 1, 2, and 3) were all acceptable to the treatment planner.

The final level of optimization generally attempts to decrease radiation dose to the healthy normal tissue surrounding the target volumes. This was achieved by minimizing its mean dose. After the optimization, the mean dose was found to decrease from 20.9 Gy to 14.9 Gy. The resulting beamlet intensity patterns represented the final optimal IMRT solution for this brain case.

3.2 Algorithmic Performance

Algorithmic performance of the lexicographic method for the demonstrated brain case is summarized in Table 3. As described in Sect. 2.2, this method is known to

Table 3. All timing results are based on an Alpha workstation having a 666 MHz CPU speed and ∼500 Mbytes of physical memory.

	1st Level	2nd Level	3rd Level
# of objective functions	1	1	1
# of constraints	0	4	7
# of independent variables	53,000	53,000	53,000
# of dependent variables	1	5	8
# of major iterations	21	43	171
# of function calls	23	83	468
Function calc. time (ms)	≈60	≈60	≈360
Jacobian calc. time (ms)	≈200	≈450	≈820
Total optimization time (s)	**6**	**52**	**363**

be a reduced feasible region method since the feasible space is gradually reduced with the increased number of constraints as the method progresses. Accordingly, the SQP algorithm exhibits a general trend of using more iterations and function-calls before finding a feasible and optimal solution. The function calculation time generally increases with the decreasing level since the low levels involve more tissue structures and therefore more calculations for the increased dose points. Potentially, one could expect a further improvement of algorithmic performance if the planning criteria used at each level of optimization exhibit a convex shape so that a warm start can be used based on the solution obtained from the prior level, instead of a randomly initialized start.

The ADOL-C gradient taping required only 90 msec. For the brain case, the tape stores up to 350,000 arithmetic operations. If the matrix-multiplication were included in the gradient tape (i.e., without the interface contraction), the taping would have taken 28,000 msec. Moreover, CPU times for the single function and gradient evaluations would be significantly longer, taking 9,000 and 33,000 msec, respectively.

4 Conclusions

The use of AD tools significantly reduced the development time and efforts for derivative calculations involving a comprehensive set of planning criteria used in IMRT optimization problems. Particularly, for the lexicographic method, ADOL-C has shown multiple benefits over other tools attributed to its use of operator overloading and function pointer features since this method requires interactive constructions of objective and constraint functions for difference levels of optimization. Early results indicate that highly accurate derivatives are essential to produced both feasible and optimal solutions at a good convergence rate. It was also found that the lexicographic method provides an intuitive and effective way of prioritizing planning objectives and articulating their priorities into a final IMRT solution.

Application of Targeted Automatic Differentiation to Large-Scale Dynamic Optimization*

Derya B. Özyurt and Paul I. Barton

Department of Chemical Engineering, Massachusetts Institute of Technology
Cambridge, MA
{derya,pib}@mit.edu

Summary. A targeted AD approach is presented to calculate directional second order derivatives of ODE/DAE embedded functionals accurately and efficiently. This advance enables us to tackle the solution of large scale dynamic optimization problems using a truncated-Newton method where the Newton equation is solved approximately to update the direction for the next optimization step. The proposed directional second order adjoint method (dSOA) provides accurate Hessian-vector products for this algorithm. The implementation of the "dSOA powered" truncated-Newton method for the solution of large scale dynamic optimization problems is showcased with an example.

Key words: Directional second order adjoint, truncated-Newton method, Hessian-vector product, ODE, temperature profile matching, TAMC

1 Introduction

Automatic Differentiation (AD) has an increasingly important and enabling role to play in the large-scale computations of interest to the chemical and biological processing industries. The solution of large-scale dynamic optimization problems modeled by ODEs or DAEs and involving many optimization parameters is one of these computations. The efficient numerical solution of these problems is important in many applications where optimization of time-dependent performance is required. In this paper, we will describe an efficient method employing a targeted AD approach to solve the dynamic optimization problem:

$$\min_{p} \quad J(p) = h(x(t_f, p), p) + \int_{t_0}^{t_f} g(t, x(t, p), p)\, dt$$

* This material is based upon work supported by the National Science Foundation under Grant No. OCE-0205590.

subject to

$$\begin{aligned} f(t, x, \dot{x}, p) &= 0 \\ f_0(t_0, x(t_0), \dot{x}(t_0), p) &= 0 \\ p^L \leq p \leq p^U \ , & \end{aligned} \tag{1}$$

where the states $x(t,p) \in \mathbb{R}^{n_x}$, the optimization parameters $p \in \mathbb{R}^{n_p}$, $t \in [t_0, t_f]$, and f_0 is a function prescribing the initial conditions for a general (implicit) DAE.

Inherently, dynamic optimization problems consist of an optimization problem along with integration of the dynamic system for the purpose of obtaining the objective function value and its derivatives. An approach which decouples solution of the optimization problem from solution of the embedded dynamic system can enable exploitation of the full advantages of state-of-the-art integration (e.g., a multistep BDF method with efficient sensitivity/adjoint calculation capabilities) and large-scale nonlinear programming (NLP) tools. The decoupling results in a more tractable optimization problem with less variables and constraints than its alternative collocation method, i.e., full discretization, generates. However, the increase in the cost of function and derivative evaluations and the possibilities of generating non-physical and/or non computable intermediate iterates have to be considered. There are two well-established methods that have proven to be effective for the solution of optimal control problems via this decoupling, namely the single- and multiple-shooting methods. Both approaches will be reinforced by improvements in the efficient and accurate computation of specific derivative information. This is because both require estimation of the first and second order derivatives of the objective function. These derivatives can be obtained by solving for the first and second order sensitivities of the state variables. However, this procedure is computationally expensive unless the number of optimization parameters is relatively small. By solving the adjoint system for a given direction in the parameter space the dependence on the number of the parameters can be reduced significantly.

Until now, first order derivatives have been calculated efficiently with either the adjoint approach [92] or the forward sensitivities approach [170]. Optimization procedures for dynamic systems embedded can be improved by utilizing accurate second order derivatives, therefore some attempts have been made to include second order information by computing directional second order sensitivities of the state variables [90]. However, this approach obviously cannot eliminate the dependence of the derivative evaluation cost on the number of optimization parameters. The standard way of estimating directional second order derivatives is to use directional finite differences based on a first order adjoint code:

$$\left.\frac{\partial^2 J}{\partial p^2}\right|_{p=p^*} u \approx \frac{\nabla J(p^* + \varepsilon u) - \nabla J(p^*)}{\varepsilon} \ , \tag{2}$$

which requires two state and adjoint integrations, one at p^* and one at $p^* + \varepsilon u$. The cheap gradient result of AD [225] states that vector-matrix products can be evaluated for less than the cost of four function evaluations (in our case a state integration). Similarly, according to the cheap Hessian result [225], a directional second order derivative can be calculated for less than the cost of 10 function evaluations. Thus, one would not expect a directional second order adjoint method to be competitive computationally with finite differences, as confirmed by the computational results in [326, 528]. However, we will show that the method presented in this paper is capable of computing accurate directional second order derivatives noticeably

cheaper than finite differences. Furthermore, the choice of ε in (2) is constrained by the need for accurate derivative approximations on the one hand, and the integration tolerance of the first order adjoint code on the other hand. Often it is difficult or impossible to find a suitable choice of ε for a particular problem, especially in the dynamic optimization context. In contrast, the directional second order adjoint method described in this paper computes derivatives to the accuracy of the integration tolerance, a numerical parameter that is easy to set and adjust. This imparts robustness to the NLP solution procedure among other advantages.

To achieve this cheap second order derivative result requires an approach which we call "targeted AD," since the approach uses AD to construct the relevant systems of equations required by the integration subroutines rather than discretizing the state system before applying AD directly to obtain the adjoint codes. In this way, we avoid the differentiation of a large and complex integration code with corrector iterations and error control, and we exploit efficient integration procedures for the sensitivity system. Our targeted AD approach constructs from the state equations in (1), the directional Second Order Adjoint (dSOA) system consisting of the directional first order sensitivity equations, first order adjoint of the sensitivities and directional second order adjoint system [424]. Solution of the constructed dSOA system yields the directional second order derivative information efficiently and accurately. The usage of AD provides a subtle way to obtain the directional first order sensitivity system and the first and second order adjoint systems, in addition to some other matrix-vector, vector-matrix and vector-matrix-vector products required, efficiently and without round-off errors. Moreover, for stiff ODEs and DAEs this targeted AD approach enables a better use of state-of-the-art integration algorithms [170].

In this paper, we only consider stiff ODE systems. However, the theory and application can be extended to DAE embedded systems by subtle consideration of initial conditions and stability of the adjoint systems.

In the following section, we introduce the adjoint method to calculate directional second order derivatives. This is followed by the description of a general dynamic optimization method and its implementation using Nash's truncated-Newton method [391] with dSOA as a directional second order derivative evaluator. A case study demonstrates the promise of the approach presented. Then, we conclude with some final remarks.

2 Directional Second Order Adjoint Method and AD

We are interested in computing accurate second order derivatives in a given direction, i.e., $\frac{\partial^2 J}{\partial p^2} u$. Considering

$$J(p) = h(x(t_f, p), p) + \int_{t_0}^{t_f} g(t, x(t, p), p)\, dt \ ,$$

the integral form functional can be computed by appending an extra state variable to the embedded dynamic system, converting the first term on the right hand side above to another point-form functional, i.e.,

$$J(p) = h(x(t_f, p), p) + x_{n_x+1}(t_f, p) \equiv \tilde{h}(x_{n_x+1}(t_f, p), x(t_f, p), p) \ .$$

Therefore, deriving the dSOA method for only point-form functionals suffices. With $f(t,x,\dot{x},p) = \dot{x} + F(t,x,p)$ and $f_0(t_0,x(t_0),\dot{x}(t_0),p) = x(t_0) - x_0(p)$, the state equations can be written as

$$\dot{x} + F(t,x,p) = 0, \quad x(t_0) = x_0(p)\ . \tag{3}$$

The formula for the first order derivative $\frac{\partial J}{\partial p}$, is obtained by introducing a Lagrange multiplier λ and constructing the following augmented objective function [92]

$$K(\lambda,p) = J(p) - \int_{t_0}^{t_f} \lambda^T(\dot{x} + F(t,x,p))\,dt\ .$$

Then

$$\begin{aligned}\frac{\partial J}{\partial p} = \frac{\partial K}{\partial p} &= \tilde{h}_p(t_f) + \tilde{h}_x(t_f)x_p \\ &\quad - \int_{t_0}^{t_f} \lambda^T(\dot{x}_p + F_x x_p + F_p)\,dt - \int_{t_0}^{t_f} \lambda_p^T(\dot{x} + F(t,x,p))\,dt\ ,\end{aligned}$$

which after some arrangement and noting that the second integral term is identically zero becomes

$$\frac{\partial J}{\partial p} = \tilde{h}_p(t_f) + \tilde{h}_x(t_f)x_p - \int_{t_0}^{t_f} (\lambda^T F_p + \lambda^T F_x x_p - \dot{\lambda}^T x_p)\,dt - (\lambda^T x_p)|_{t_0}^{t_f}\ .$$

Here subscripts p and x denote partial derivatives with respect to the parameters and state variables, respectively, e.g., $F_p = \frac{\partial F}{\partial p}$. By defining the first order adjoint equations as

$$\begin{aligned}\dot{\lambda}^T - \lambda^T F_x &= 0 \\ \lambda^T(t_f) &= \tilde{h}_x(t_f)\ ,\end{aligned} \tag{4}$$

we obtain

$$\frac{\partial J}{\partial p} = \int_{t_0}^{t_f} -\lambda^T F_p\,dt + \tilde{h}_p(t_f) + (\lambda^T x_p)|_{t_0}\ .$$

Now we obtain the second order derivative in the direction u as

$$\begin{aligned}\frac{\partial^2 J}{\partial p^2}u &= \frac{\partial}{\partial p}\left(\frac{\partial J}{\partial p}\right)u \\ &= \frac{\partial}{\partial p}\left(\int_{t_0}^{t_f} -\lambda^T F_p\,dt + \tilde{h}_p(t_f) + (\lambda^T x_p)|_{t_0}\right)u\ .\end{aligned} \tag{5}$$

Therefore, the directional second order derivative of a point-form functional is obtained from[1]

$$\begin{aligned}\frac{\partial^2 J}{\partial p^2}\ u = &\int_{t_0}^{t_f} -\{F_p^T\mu + (\lambda^T \otimes I_{n_p})(F_{pp}\ u + F_{px}\ s)\}\,dt \\ &+ \tilde{h}_{pp}(t_f)\ u + \tilde{h}_{px}(t_f)\ s(t_f) + \Big[(\lambda^T \otimes I_{n_p})x_{pp}\ u + x_p^T\mu\Big]_{t=t_0}\ .\end{aligned} \tag{6}$$

[1] Here I_{n_x} is a n_x by n_x identity matrix. $\otimes$ denotes a Kronecker product. Double subscripts denote second order partial derivatives, e.g., $F_{xp} \equiv \frac{\partial^2 F}{\partial p \partial x}$.

The variables $s \equiv x_p u$, λ, and $\mu \equiv \lambda_p u$, i.e., directional first order sensitivities, the first order adjoints and directional second order adjoints, respectively, need to be calculated. Directional first order sensitivity equations for the state system (3) are

$$\dot{s} + F_x s + F_p u = 0, \quad s(t_0) = x_{0_p} u \ , \tag{7}$$

and the directional second order adjoint equations are

$$\begin{aligned} \dot{\mu} - F_x^T \mu &= (\lambda^T \otimes I_{n_x})(F_{xp} u + F_{xx} s) \\ \mu(t_f) &= \tilde{h}_{xx}(t_f) s(t_f) + \tilde{h}_{xp}(t_f) u \ . \end{aligned} \tag{8}$$

The main issue with these equation systems is their efficient evaluation in a numerical integration scheme. They contain several vector-matrix, matrix-vector products (Table 1). Unlike the application of direct AD to a code which discretizes the state system, we leave the state system intact without any discretization and construct the necessary code by applying targeted AD to assemble pieces required to evaluate the above equation systems. Fortran codes that become subject to AD to obtain the first order derivatives involve a set of subroutines describing the ODE system and the point-form functional. The former can have many subroutines to describe the physical system, whereas the latter is usually simpler in shape and complexity. The first order derivative codes generated by AD are used to obtain the second order codes.

Table 1. A list of derivatives required by dSOA.

First Order:	$\tilde{h}_x$, $\tilde{h}_p$, $\mu^T F_p$ $(\lambda^T F_x)$, $F_x s$, $F_p u$, $\lambda^T F_x$ $(\mu^T F_x)$
Second Order:	$\tilde{h}_{xx} s + \tilde{h}_{xp} u$, $\tilde{h}_{px} s + \tilde{h}_{pp} u$, $(\lambda^T \otimes I_{n_p})(F_{pp}\ u + F_{px}\ s)$, $(\lambda^T \otimes I_{n_x})(F_{xp}\ u + F_{xx}\ s)$

The adjoint equations are obtained by applying the reverse mode of AD to the code evaluating the residuals of the state equations, i.e., $\dot{x} + F(t, x, p) = \Delta$ with the independent variables $(\dot{x}, x)$ in the seed direction $(\dot{\lambda}, \lambda)$. The directional first order sensitivity equations are obtained by the forward mode of AD applied to the same residual code with the independent variables $(\dot{x}, x, p)$ in the seed direction $(\dot{s}, s, u)$. Provided that the residual evaluator is a composition of twice differentiable elementary functions, second order derivatives can be obtained by applying four different combinations of the forward and reverse modes (forward over reverse, reverse over reverse, forward over forward, reverse over forward). However, the cost of forward propagation of the tangents increases linearly with the number of domain directions, proving the forward over forward method inefficient. On the other hand, reverse over forward mode requires applying the reverse mode to a more complicated code generated by the forward mode. Moreover, it has been shown [225] that the adjoints of adjoints can be represented as tangents of adjoints, leaving the forward over reverse mode the most reasonable alternative for our problem. Therefore, the forcing term in the directional second order adjoint equations is constructed by applying the forward mode of AD to the code generated by the reverse mode with the independent variables (x, p) in the seed direction (s, u). The homogeneous part of the

ODE can be evaluated by the same code as the first order adjoint equations with a different seed direction, namely $(\dot{\mu}, \mu)$. Finally, the integrand in the quadrature equations (Eqn. (6)) is constructed by applying the forward over reverse mode of AD to $\dot{x} + F(t, x, p) = \Delta$. Similar to the construction of the directional second order adjoint equations, the reverse mode of AD with independent variable (p) in the seed direction (λ) yields the vector-matrix product $(\lambda^T F_p)$ in Eqn. (5) which is reused with the seed direction (μ) to compute the first term of the integrand. The second term of the integrand is computed by applying the forward mode of AD to the $(\lambda^T F_p)$- code with independent variables (x, p) in the seed direction (s, u).

After construction of the dSOA system, evaluation of the directional second order derivatives requires a numerical procedure involving a forward and then a backward integration pass. Intuitively, backward over forward integration is as efficient as forward over reverse differentiation. The forward integration is implemented efficiently using the staggered corrector method [170] to obtain the states and directional first order sensitivities in one or a small number of directions. Numerical integration methods for stiff ODEs typically employ a corrector iteration at each time step. The use of the staggered corrector method of [170] exploits the similarity between the corrector iteration for the state equations and those for the directional first order sensitivities. At each integration step first the discretization equations determining the state variables are solved. The factored Jacobian matrix used in the Newton iteration for the state variables is reused in the following Newton iterations to solve the linear systems of equations corresponding to the discretization of the sensitivity systems. The incremental computational cost of calculating the sensitivity equations for a single direction by employing the staggered corrector method can be less than 30% of that for calculating the state equations.

During the forward integration the state and first order sensitivity trajectories are stored. Subsequently, a backward integration is performed of the first order adjoint and second order directional adjoint equations in one or a small number of directions, once again, with the staggered corrector method. This is because the second order adjoints can be interpreted as sensitivities of the first order adjoints, therefore they can also be calculated efficiently employing a staggered corrector method similar to the sensitivities of the state variables. Finally, $\frac{\partial^2 J}{\partial p^2} u$ is evaluated at time t_0 from the quadrature calculation.

This procedure summarized in Table 2 is implemented within a modified version of DASPKADJOINT [92,335], an adjoint sensitivity solver. Specifically, DASPKADJOINT is modified to accommodate sensitivity calculations during backward

Table 2. dSOA procedure.

1. Input u, t_0, t_f
2. Forward Integration:
   state equations (3), first order sensitivities in direction u (7).
3. at t_f: Calculation of initial values for adjoints
4. Backward Integration:
   first order (4) and directional second order (8) adjoints,
   quadrature in equation (6)
5. at t_0: Calculation of $\frac{\partial^2 J}{\partial p^2} u$

integration, to save and retrieve forward sensitivities, to calculate the second order derivatives at the final step, and to perform these calculations for several directions. These modifications are carried out only for the integration methods using direct linear solvers.

The input information necessary to set up a given problem is similar to that of DASPKADJOINT. However, in addition to the state and first order adjoint equations, directional first order sensitivity and directional second order adjoint equations along with the integrands of the quadrature are needed.

Before applying the dSOA method results in the dynamic optimization context, let us examine how well dSOA performs compared to the finite difference approximation with the help of an example.

2.1 Example

Our aim is to see whether dSOA is advantageous compared to the finite difference method (FDM) using two gradient evaluations via the first order adjoint method (2). Also, we want to see whether there is an advantage of calculating two directional second order derivatives simultaneously with dSOA. To this end, we consider an integral-form functional as our objective function, namely

$$J(p) = \int_{t_0}^{t_f} \sum_{i=1}^{n_x} z_i \, dt \ ,$$

where $t_0 = 0$ and $t_f = 0.16$. The dynamic system is described by a system of PDEs

$$z_t = p_1 z_{xx} + p_2 z_{yy} \ ,$$

posed on a two dimensional unit square with zero Dirichlet boundary conditions [91]. Spatial derivatives (those with respect to x and y) are approximated by centered finite difference approximations on a uniform grid of size M, where $M \times M = n_x$. The boundary conditions are included in the discretized PDE, reducing the system to an ODE. The initial conditions are posed as

$$z(0, x, y) = 16x(1 - x)y(1 - y) \ .$$

The nominal values of p_1 and p_2 are equal to 1.0. The optimization parameters are the boundary conditions, initial conditions, and two parameters from the original PDE. The direction is chosen to be

$$u_1 = (1, 0, \cdots, 0) \ .$$

The relative tolerance (RTOL) for the integration is 10^{-8}, whereas the absolute tolerance (ATOL) is 10^{-10}.

Three important observations are apparent (See Fig. 1) for the dSOA method:

1. the cost ratio of dSOA to a simulation, $\left(\frac{\text{cost}(dSOA)}{\text{cost}(SIM)}\right)$ is in the range 2-4,
2. the cost of calculating a directional second order derivative with dSOA is more efficient than FDM, especially for larger problems,
3. additional directional derivatives can be obtained even "cheaper."

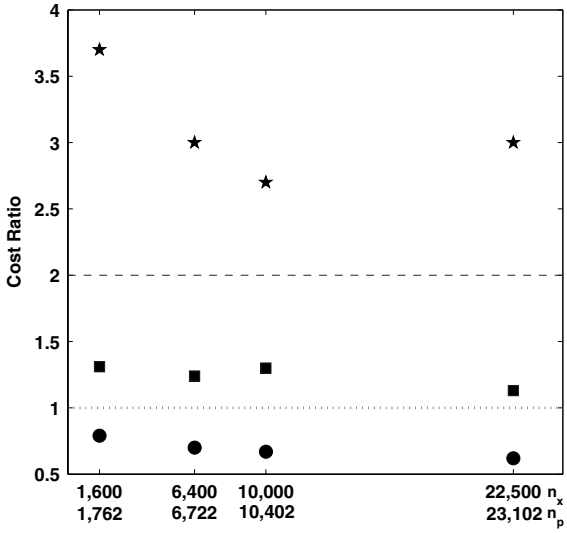


Fig. 1. Cost ratios for 2-D heat equation example. ($\bigstar$: $\frac{\text{cost}_{(dSOA)}}{\text{cost}_{(SIM)}}$, $\bullet$: $\frac{\text{cost}_{(dSOA)}}{\text{cost}_{(FDM)}}$, $\blacksquare$: $\frac{\text{cost}_{(dSOA_2)}}{\text{cost}_{(dSOA_1)}}$)

The final observation will prove very promising in some applications, such as biconjugate methods, where two directional derivatives can be desired at the same time. The incremental cost of computing the second and subsequent directions in the staggered corrector method is typically much less than that for computing the first direction. Therefore, when we compare the computational cost of the second order derivatives in two directions, i.e.,

$$u_1 = (1, 0, \cdots, 0)$$
$$u_2 = \left(\frac{1}{n_p}, \cdots, \frac{1}{n_p}\right),$$

at the same time with the cost of calculating the first column of the Hessian matrix, i.e., in the direction u_1 only, we see a 15-30% increase in the computational cost, as presented by the ratio $\frac{\text{cost}_{(dSOA_2)}}{\text{cost}_{(dSOA_1)}}$.

3 Dynamic Optimization and dSOA

A solution procedure for the dynamic optimization problem (1) based on decoupling requires an NLP solver and a method to provide the objective function, first and second order derivative information for a given set of parameter values. Second order optimization methods require the second order derivatives of the objective function with respect to the optimization parameters. Although this second order information can be approximated using a finite difference scheme based on gradient evaluations, exact second order information can impart robustness and reduce the number of iterations for convergence to a local minimum.

The directional second order derivatives provided by dSOA fit perfectly into an NLP solution procedure in which a descent direction is obtained by solving the Newton equation:

$$\nabla^2 J(p^{(k)})u = -\nabla J(p^{(k)}) \; . \tag{9}$$

By solving the Newton equation approximately, a "Newton-like" search direction can be obtained which approaches the Newton direction in the limit as the minimum is approached, resulting in a superlinear convergence [393]. Therefore, by calculating an approximate Newton direction, a compromise between fast convergence and computational cost per iteration can be attained [391,392]. This truncated-Newton method allows incorporation of "exact" directional second order information. Therefore they may be promising NLP solvers for large-scale dynamic optimization problems, provided that a technique to calculate the requisite second order information efficiently and accurately is available.

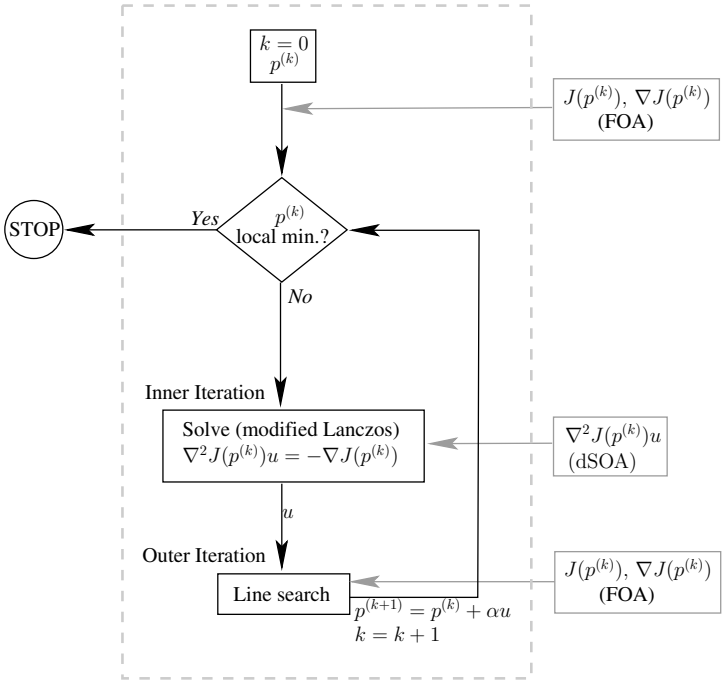


Fig. 2. Truncated-Newton Algorithm.

The truncated-Newton (TN) algorithm adapted to solve the dynamic optimization problem (1), depicted in Fig. 2, consists of an outer iteration for the nonlinear optimization problem and an inner iteration for approximate solution of the Newton equation. The directional second order adjoint method introduced above along with its first order counterpart computes the directional second order derivatives and gradients of the objective function efficiently and accurately, improving both the outer and inner iterations of the truncated Newton methods.

This algorithm is implemented using Nash's truncated-Newton code for optimization problems with bounds on variables [391]. The original code is modified to employ the dSOA method within the inner iteration.

3.1 Example

Our main aim now is to show that, for large-scale dynamic optimization problems, obtaining the second order information via dSOA is advantageous to the overall optimization procedure when compared to approximating it by directional finite differences.

Numerical experiments are performed on a Pentium IV/3.20 GHz Shuttle X machine with 1 GB memory and running Linux kernel 2.4. All automatic differentiation tasks are performed by the AD tool TAMC [205].

Temperature Profile Matching

We consider a two dimensional boundary control heating problem adopted from [431]. A rectangular domain (See Fig. 3) is heated by controlling the temperature on its boundaries Ω_1 and Ω_2. Within a specified interior subdomain ($\Omega_c = \{(x,y)|x_c \leq x_{max}, y_c \leq y_{max}\}$) the temperature profile obeying a nonlinear parabolic PDE has to follow approximately a prespecified temperature-time trajectory. The objective function is

$$\min_v J(v) = \int_{t_0}^{t_f} \int_{y_c}^{y_{max}} \int_{x_c}^{x_{max}} \omega(x,y,t)[T(x,y,t) - \tau(t)]\, dx\, dy\, dt \ ,$$

where $\omega(x,y,t) = 0$ for $t \in [0, 0.2]$, and $\omega(x,y,t) = 1$ for $t \in [0.2, 2]$ is chosen. The nonlinear parabolic PDE described as

$$\begin{aligned} \alpha(T)\,[T_{xx} + T_{yy}] + S(T) &= T_t, \quad (x,y,t) \in \Omega \times [t_0, t_f] \\ T(x,0,t) - \lambda T_y &= v_1(x,t), \quad x \in \partial\Omega_1 \\ T(0,y,t) - \lambda T_x &= v_2(y,t), \quad y \in \partial\Omega_2 \\ T_x(x_{max},y,t) &= 0 \\ T_y(x,y_{max},t) &= 0 \\ 0 \leq v_1, v_2 &\leq v_{max} \end{aligned}$$

is reduced into an ODE using the numerical method of lines [431] with $\alpha(T) = 1.0$. The internal heat generation is represented as

$$S(T) = S_{max} \exp\left(\frac{-\beta_1}{\beta_2 + T}\right) \ ,$$

where $\beta_1 = 0.2$, $\beta_2 = 0.05$, and $S_{max} = 0.0$ unless specified otherwise. For the instance considered $t_0 = 0$, $t_f = 2.0$, $x_{max} = 0.8$, $y_{max} = 1.6$, $x_c = 0.6$, $y_c = 1.2$, and $v_{max} = 1.1$. The control functions on the boundaries $\partial\Omega_1 = (x,y)|y = 0$ and $\partial\Omega_2 = (x,y)|x = 0$ are defined as

$$v_1(x,t) = \begin{cases} v(t) & 0 \leq x \leq 0.2 \\ \left(1 - \frac{x-0.2}{1.2}\right) v(t) & 0.2 \leq x \leq x_{max} \end{cases}$$

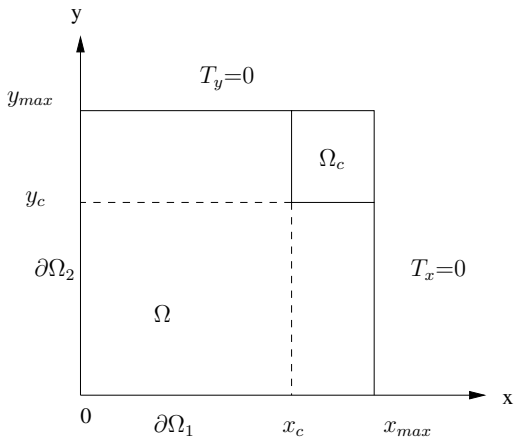


Fig. 3. Two dimensional domain for the temperature profile matching example.

$$v_2(y,t) = \begin{cases} v(t) & 0 \le y \le 0.4 \\ \left(1 - \frac{y-0.4}{2.4}\right) v(t) & 0.4 \le y \le y_{max} \end{cases}.$$

The spatial integrations in the objective function are approximated using the rectangular integration rule.

For an integration tolerance of 10^{-10} and the optimization accuracy value of 10^{-8}, we tested several cases with a gradually increasing number of parameters. The number of parameters is increased both by finer control parameterization and addition of initial conditions as degrees of freedom to the optimization. These latter additions are noted by a (+) sign in Column 2 of Table 3, i.e., $10 + 450$ denotes 10 control parameters and 450 initial conditions are considered as optimization parameters. Nonlinearity is introduced by setting $S_{max} = 0.5$ and noted by (nl) in Column 2. We can conclude that

1. The computational cost per iteration of a "dSOA powered TN" method stays (nearly) constant when more parameters are added to the optimization problem (Table 4). The only increase is caused by integration restarts at each control vector parameterization time point. Obviously, the total cost of optimization increases because the number of iterations increases.
2. The finite difference approximation to the directional second order derivatives does not result in a robust performance (Table 3). Especially, for cases with larger number of parameters or nonlinearity, the dSOA based method outperforms this approach in terms of attained objective function value at a comparable overall computational cost.
3. For large numbers of parameters average time per iteration values for dSOA tend to stay within 10% range of the TPI values for FDM (Fig. 4). In contrast to the example in Sect. 2, the cost per iteration of FDM is less, because the TN algorithm only requires a single gradient evaluation at each inner iteration to approximate the directional second order derivative. A gradient evaluation at the nominal parameter values is already available from the previous outer iteration.

Table 3. Computational results for the temperature matching example using dSOA and finite difference approximation of the Hessian-vector product ((n_x: number of equations, n_p: number of parameters, NIT: number of iterations, NF: number of outer iterations, CG: number of inner iterations, J: objective function value, CPU: overall CPU time; * marks the runs where the optimization procedure terminated because no significant improvement in the objective function value is achieved).

		dSOA					FDM				
n_x	n_p	NIT	NF	CG	$J(\times 10^5)$	$CPU(s)$	NIT	NF	CG	$J(\times 10^5)$	$CPU(s)$
45	40	33	34	105	4.295	69.3	33	33	121	4.330	61.1
	40+5	30	30	100	4.325	65.4	30	31	94	4.358	49.0
	40+20	34	35	101	4.582	67.4	24	25	54	11.95*	31.5
	160	78	79	206	4.812	464.1	83	84	222	4.600	412.8
	320	128	129	269	7.779	1108.8	112	113	224	10.80*	883.4
	320(nl)	115	120	229	75.99	983.4	110	114	219	155.55*	899.7
861	10+5	12	13	36	6.922	1049.6	13	14	39	6.922	988.1
	10+450	32	33	183	6.258	5079.1	7	8	16	16.86*	444.9
	20	22	23	80	4.736	4019.5	24	25	80	4.735	3506.1
	50	48	49	171	4.193	17660	27	28	83	4.606*	8116.8

Table 4. Average CPU time per iteration (TPI) for FDM and dSOA.

		$TPI(s)$	
n_x	n_p	FDM	dSOA
45	40	0.4	0.5
	40+5	0.4	0.5
	40+20	0.4	0.5
	160	1.4	1.6
	320	2.6	2.8
	320(nl)	2.7	2.8
861	10+5	18.6	21.4
	10+450	18.5	23.5
	20	33.4	39.0
	50	73.1	80.3

4 Conclusions

We have incorporated an efficient method to calculate accurate directional second order derivatives for stiff ODE embedded functionals into the truncated-Newton method to solve large-scale dynamic optimization problems. Our methodology relies heavily on automatic differentiation, since AD is the only reliable and efficient technology for evaluation of the equation systems required by dSOA. Both the forward and reverse modes of AD are intensively used to evaluate several matrix-vector, vector-matrix and vector-matrix-vector products. The resulting dSOA method has only "weak" n_p dependence [424], therefore as the number of parameters is increased, the average cost of derivative evaluations for large dynamic systems with many parameters does not grow.

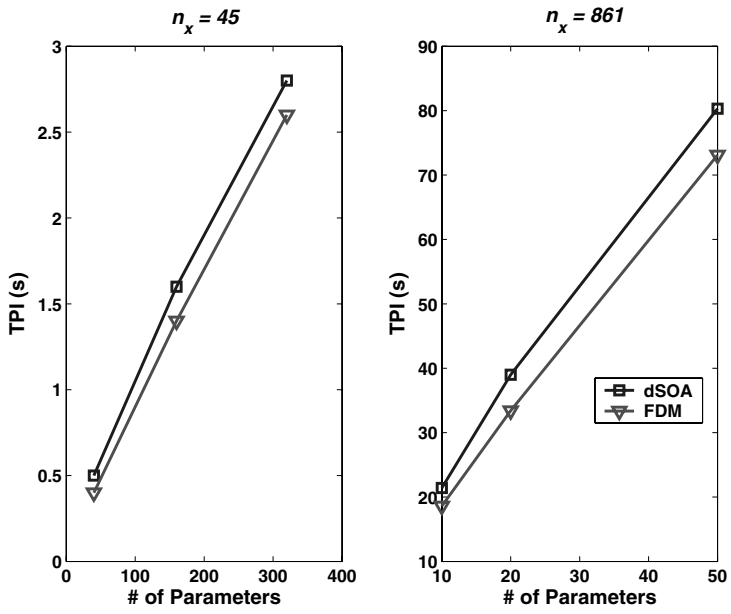


Fig. 4. Comparison of TPI for FDM and dSOA.

Although the computational costs of evaluating a directional second order derivative via finite difference approximation and with dSOA are comparable for a large number of parameters, obtaining the latter accurately improves the computation time by reducing the total number of iterations and increases the robustness of the TN method.

Improvements to the existing implementation are foreseen in more efficient code construction with AD, in using two directional methods for the inner iterations of truncated-Newton method, in incorporating (in)equality constraints within the truncated-Newton method and in the usage of iterative linear solvers to reduce the cost of integration in general.

Automatic Differentiation: A Tool for Variational Data Assimilation and Adjoint Sensitivity Analysis for Flood Modeling

W. Castaings[1], D. Dartus[2], M. Honnorat[3], F.-X. Le Dimet[4], Y. Loukili[5], and J. Monnier[6]

[1] INRIA/IDOPT, France
william.castaings@imag.fr*
[2] INPT – IMFT/HYDRE, France
dartus@imft.fr
[3] INPG – INRIA/IDOPT, France
marc.honnorat@imag.fr**
[4] UJF – INRIA/IDOPT, France
francois-xavier.le-dimet@imag.fr
[5] INRIA/IDOPT, France
youssef.loukili@imag.fr***
[6] INPG – INRIA/IDOPT, France
jerome.monnier@imag.fr

Summary. This paper illustrates the potential of automatic differentiation (AD) for very challenging problems related to the modeling of complex environmental systems prone to floods. Numerical models are driven by inputs (initial conditions, boundary conditions and parameters) which cannot be directly inferred from measurements. For that reason, robust and efficient methods are required to assess the effects of inputs variations on computed results and estimate the key inputs to fit available observations. We thus consider variational data assimilation to solve the parameter estimation problem for a river hydraulics model, and adjoint sensitivity analysis for a rainfall-runoff model, two essential components involved in the generation and propagation of floods. Both applications require the computation of the gradient of a functional, which can be simply derived from the solution of an adjoint model. The adjoint method, which was successfully applied in meteorology and oceanography, is described from its mathematical formulation to its practical implementation using the automatic differentiation tool Tapenade.

Key words: Adjoint model, data assimilation, sensitivity analysis, river hydraulics, catchment hydrology, flood modeling, Tapenade

* Work supported by the Pactes program funded by CNES.
** Work supported by a CNRS doctoral grant (BDI) co-funded by CNES.
*** Post-doctoral work funded by Région Rhône-Alpes ("Prévention numérique de crues" project).

1 Introduction

Flooding is the result of complex interactions between the components of water cycle, and the forecast of such catastrophic events requires an integrated approach (models and data) for the hydro-meteorological prediction chain. The modeling of flood generation and propagation involves catchment scale hydrology and river hydraulics. Actually, every model component only leads to an approximation of the geophysical reality, since the underlying physics formulation and the model inputs are all sources of uncertainty. Understanding, analysis and reduction of this uncertainty induce the following issues:

- Empirical parameters associated with model formulation, as well as initial and boundary conditions, which are essential to mathematical closure and drive the considered system, remain very difficult to estimate.
- Quantitative measures of the effect of input variations on model prognostic variables provide physical insight into the model dynamics, and are useful guidelines for the choice of calibration parameters and formulation of calibration criteria.

A deterministic approach dealing with the aforementioned estimation and sensitivity analysis problems requires computing the derivatives of a function of model output variables with respect to input variables. Modern automatic differentiation (AD) tools such as Tapenade [255, 258, 279] provide a very helpful and efficient assistance for the related practical implementation issues.

Two applications are presented in this paper: variational data assimilation [68, 327] and adjoint sensitivity analysis [86].

2 The Adjoint Method

The evolution of the state of many time-dependent physical systems can be described by a system of differential equations. For a given model, the value of the state variable y is driven by the control variables which are potentially all model inputs. For the general presentation of the adjoint method, we will consider the initial condition u and a model parameter v as control variables:

$$\left\{ \begin{array}{lll} \frac{\partial y}{\partial t}\,(t) + A\big(y(t), v\big) & = & 0 \qquad \forall\, t \in]0, T] \\ y(0) & = & u \end{array} \right. , \tag{1}$$

where A is a (possibly nonlinear) partial differential operator. Let ϕ be a general objective function which depends on the control variables through the state variable:

$$\phi(u, v) \;=\; \int_0^T \varphi\big(y(u, v; t)\big)\, dt \;, \tag{2}$$

where φ is a sufficiently smooth functional. With the adjoint method, it is possible to compute efficiently the partial derivatives of a function of the model state variable with respect to control variables [341]. If p is defined as the solution of the adjoint model:

$$\left\{ \begin{array}{lll} \frac{\partial p}{\partial t}\,(t) - \Big[\frac{\partial A}{\partial y}\big(y(t), v\big)\Big]^* \cdot p(t) & = & \frac{\partial \varphi}{\partial y}(t) \qquad \forall\, t \in [0, T[\\ p(T) & = & 0\;, \end{array} \right. \tag{3}$$

where $[\,\cdot\,]^*$ denotes the adjoint operator, then we obtain a simple expression of the partial derivatives of the functional:

$$\frac{\partial \phi}{\partial u}(u,v) = -p(0) \quad \text{and} \quad \frac{\partial \phi}{\partial v}(u,v) = \int_0^T \left[\frac{\partial A}{\partial v}\big(y(t),v\big)\right]^* \cdot p(t)\,dt\ .$$

All partial derivatives are calculated with a single forward integration of the direct model (1) followed by a single backward integration of the adjoint model (3). Another advantage of this method is that the homogeneous part of the adjoint equations is independent of the functional φ. In other words, the same model can be used to calculate the derivatives of several functionals without major modifications.

The calculation of partial derivatives of a functional is useful in several domains of flood simulation. Here, we are especially interested in two essential applications: variational data assimilation and adjoint sensitivity analysis. Variational data assimilation finds the control variables that minimize a cost function measuring the discrepancy between the state variable of the model and data obtained from the observation of the physical system. An efficient minimization of the cost function is carried out by using a descent algorithm requiring the computation of its gradient.

The adjoint sensitivity analysis determines the contribution of all model inputs to the variation of a response function. Instead of performing finite difference approximation of the gradient, requiring extensive direct model computations (brute force method), a single run of the adjoint method provides all sensitivities. Various applications of the adjoint method were investigated for environmental problems in the framework of the INRIA/IDOPT project [359, 413, 522, 554, 564].

In practice, the numerical computation of the gradient of the functional ϕ is performed by an implementation of the adjoint method which requires the construction of an adjoint code. From a numerical point of view, the best representation of the functional is the associated computer code. Instead of manually coding the adjoint, which would require significant training and time investment, we chose to use the AD tool TAPENADE [279] to create the adjoint of the implementation of the direct model. This code-based approach makes it possible to compute the numerical value of the gradient exact up to roundoff [156]. This is crucial for variational data assimilation problems, where the minimization algorithm may fail to converge if the gradient is not accurately computed.

3 River Hydraulics

Flood forecasting requires an accurate modeling of river flows. The most commonly used mathematical models for operational purposes in river hydraulics rely on the Shallow Water Equations (SWE). The two-dimensional SWE are derived from the three-dimensional Navier-Stokes equations by a vertical integration under the hydrostatic assumption. In the conservative formulation, the state variables are the water depth h and the discharge $\mathbf{q} = h\mathbf{u}$, where $\mathbf{u}$ is the depth-averaged velocity vector. If we consider a computational domain Ω with a boundary Γ, the two-dimensional SWE can be written as follows:

$$\begin{cases} \partial_t h \;+\; \mathrm{div}(\mathbf{q}) \;=\; 0 & \text{in } \Omega\times]0,T] \\ \partial_t \mathbf{q} \;+\; \mathrm{div}\big(\frac{1}{h}\mathbf{q}\otimes\mathbf{q}\big) \;+\; \frac{1}{2}g\nabla h^2 & \\ \qquad\quad +\; gh\nabla z_b \;+\; g\frac{n^2\mathbf{q}\|\mathbf{q}\|_2}{h^{7/3}} \;=\; 0 & \text{in } \Omega\times]0,T] \\ h(0)=h_0\,, \quad \mathbf{q}(0)=\mathbf{q}_0 & \text{in } \Omega\,, \end{cases} \tag{4}$$

where g is the magnitude of the gravity vector, z_b is the bed elevation, n is the Manning roughness coefficient, h_0 and $\mathbf{q}_0$ are the initial conditions. Moreover, we must add boundary conditions. For an inlet Γ_{in}, a discharge $\mathbf{q}_{\text{in}}$ is imposed:

$$\mathbf{q}_{|\Gamma_{\text{in}}}(t) \;=\; \mathbf{q}_{\text{in}}(t) \qquad \forall\, t\in]0,T]\,. \tag{5}$$

At an outlet Γ_{out}, we can either prescribe a water depth h_{out} or impose Neumann conditions:

$$h_{|\Gamma_{\text{out}}}(t) \;=\; h_{\text{out}}(t)\,, \quad \tfrac{\partial \mathbf{u}}{\partial \mathbf{n}}|_{\Gamma_{\text{out}}}(t) \;=\; 0 \qquad \forall\, t\in]0,T] \tag{6}$$

$$\text{or} \qquad \tfrac{\partial h}{\partial \mathbf{n}}|_{\Gamma_{\text{out}}}(t) \;=\; 0\,, \;\; \tfrac{\partial \mathbf{q}}{\partial \mathbf{n}}|_{\Gamma_{\text{out}}}(t) \;=\; 0 \qquad \forall\, t\in]0,T]\,. \tag{7}$$

The state of the flow is determined by the initial condition (a Cauchy problem) but also by the model parameters (z_b and n) and the boundary conditions. Actually, in order to carry out a simulation of a real flow, it is necessary to have a good knowledge of these model inputs. However, they are incompletely known in practice, and when an approximation is available, it is often subject to large uncertainties. For example, the flood plain elevation is measured using remote sensing techniques [27], and bed elevation data is made up of ground surveyed cross sections. The collected data consist in a set of scattered points used to assign an elevation value to each computational grid point using interpolation techniques. Unfortunately, the raw data is usually approximate, incomplete or sometimes simply missing. Moreover, the interpolation induces additional numerical approximation. Other model inputs cannot even be measured directly and should be defined by the modelers' expertise. For example, the estimation of roughness is generally based on land use classifications and empirical tables where a roughness coefficient is assigned to each land cover type [11]. A model is never perfect since it cannot take into account all the physics of the system, and its implementation induces numerical approximations. Therefore, a simulation can never reflect exactly the physical reality. However, it is possible to represent some parts of the model errors by formulating an additional term in the equations [522], introducing new control variables.

Furthermore, some observations of the flow state may be available, such as water depth, water level or velocity measurements. These should be in accordance with the simulation results. Therefore, the problem to be addressed consists of identifying a set of control variables consistent with both the simulation results and the hydraulic reality represented by observation data. Hence, we use variational data assimilation for the identification of initial and boundary conditions, model parameters, bed elevation and for the evaluation of a systematic model error. This method consists in minimizing a cost function measuring the discrepancy between the state of the simulated flow and the available observations of the real flow. The minimization is performed by a limited memory quasi-Newton algorithm [212] which requires the computation of the partial derivatives of the cost function with respect to the control variables of the model. The derivatives are computed by the adjoint method requiring the use of an adjoint code.

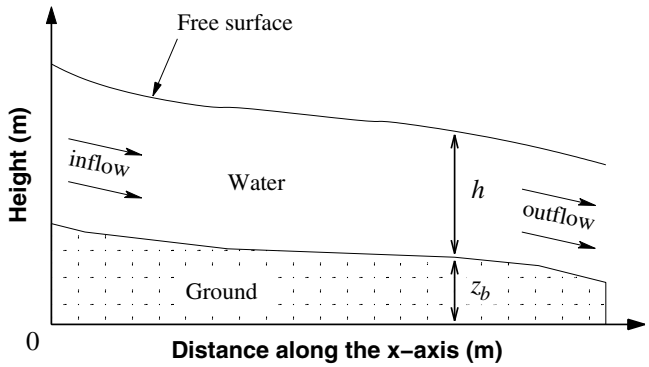


Fig. 1. One-dimensional vertical cross section of the computational domain.

The SWE (4) are solved using the finite volume method and the HLLC approximate Riemann solver [344, 510]. The direct program is written in Fortran 90 and is made up of about 1600 lines of code. The adjoint code was obtained thanks to the AD tool TAPENADE [255].

Some parts of the raw code produced by TAPENADE had to be manually modified to make it work properly. One can distinguish three main modifications. The first one consists in the correction of the adjoint code for statements that are locally non-differentiable, such as a square root. Since the appropriate treatment of the case where the argument is zero might depend on the context, this task was not automated, and manual intervention is needed to remove the singularity. The second modification is related to the computation of the cost function. In the direct code, the observations are read from a file every time step before the call to the subroutine that computes the cost increment. Since they are also needed in the reverse sweep, the observations should either be stored during the forward sweep or be reread before the call to the adjoint subroutine. However, TAPENADE did not perform either of these two options, and the latter had to be manually implemented in the adjoint code. Finally, the raw adjoint code produced by TAPENADE stored too many variables unnecessarily. For example, the state variable, which consists in a large array of dimension n, was stored $2n$ times per time step, whereas only once per time step was sufficient. Without modification, the adjoint could not run on a simple test case because the required memory was too large (more than 2 gigabytes). After the removal of the unneeded storage statements, the memory footprint is only 16 megabytes, and the ratio between the execution time of the adjoint code and that of the direct code is about 3.5.

Two numerical experiments of data assimilation are presented. They actually consist of twin experiments, where observation data are computed by the direct model with a set of known parameters. Then, a perturbation is applied to the latter, modifying the simulation results. Afterward, the variational data assimilation method is used to retrieve the original value of the parameters. The same adjoint code is used in both experiments.

The reduction of the uncertainty in the topographic data is crucial since the latter can have an important effect on the flow behavior during a flooding event. Thus, the first experiment concerns the identification of the bed elevation z_b in a

rectangular channel. The channel is 20.5×2 meters, the reference bed elevation is defined by $z_b(x,y) = 0.2 - \frac{1}{20}(x-10)^2$ if $x \in [8,10]$ and $z_b(x,y) = 0$ otherwise for all $y \in [0,2]$. A constant discharge $\mathbf{q}_{\text{in}}$ of 2 $m^3 s^{-1}$ is imposed at the inlet, and a constant water depth h_{out} of 0.6 meters is prescribed at the outlet. This configuration leads to a steady flow featuring an hydraulic jump after the bump. A vertical cross section of the computational domain is plotted in Fig. 2. For the simulation, we use a rectangular mesh made up of 82×8 finite volumes. Hence, the discrete control vector z_b consists of 656 degrees of freedom. The observations of the water depth h^{obs} and the velocity $\mathbf{u}^{\text{obs}}$ are created from this reference configuration: they are defined as equal to the state variables of the steady flow for each computational point. In this case, the observation data is time-independent. However, nothing prevents us from using observations varying in time.

A simulation is carried out with a modified configuration over a period of $T = 3$ seconds: a flat bed defined by $z_b \equiv 0$ is used with the reference steady state as an initial condition. As a result, the water flow is disrupted and becomes unsteady. In order to use the variational data assimilation method to retrieve the reference bed elevation, we introduce the following cost function to be minimized:

$$j_1(z_b) \;=\; \frac{1}{2}\int_0^T \left(\left\| h(t) - h^{\text{obs}}(t) \right\|_{\Omega}^2 + \left\| \mathbf{u}(t) - \mathbf{u}^{\text{obs}}(t) \right\|_{\Omega}^2 \right) dt \;, \tag{8}$$

where $\|\cdot\|_{\Omega}$ denotes the L^2 norm on Ω. The gradient of this cost function is computed with the adjoint code and is used as an input for the minimization algorithm. In Fig. 3 (a), the cost function and the norm of its gradient, both normalized by their initial value, are plotted against the number of iterations of the minimization process. Figure 3 (b) shows the bed elevation for several steps of the minimization process. We can see that convergence has been achieved, and the original bump on the bed is retrieved. However, even if the shape of the bump is correctly identified, one can notice a constant offset between the retrieved bed elevation and the original one because only the gradient of z_b is present in the equations. Therefore, the bed elevation can be identified only up to a constant bias.

The second experiment concerns the identification of the upstream boundary condition $\mathbf{q}_{\text{in}}$ during a flooding event. In practice, an accurate identification would be very valuable, since a flood hydrograph is usually extrapolated from water level

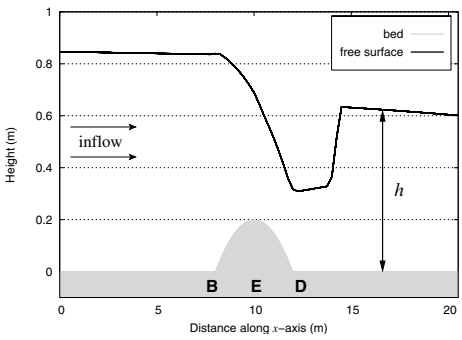
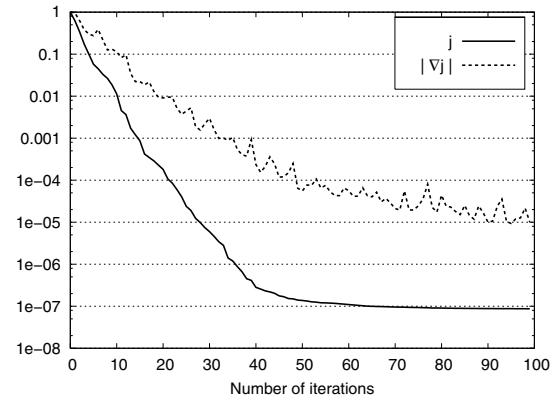
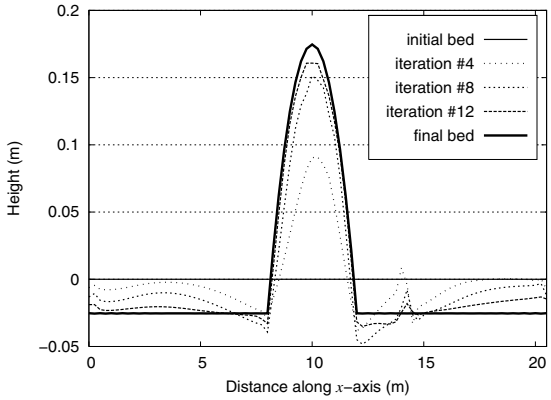


Fig. 2. Rectangular channel with a bump: vertical cross section of the computational domain.



(a) Convergence of the cost function

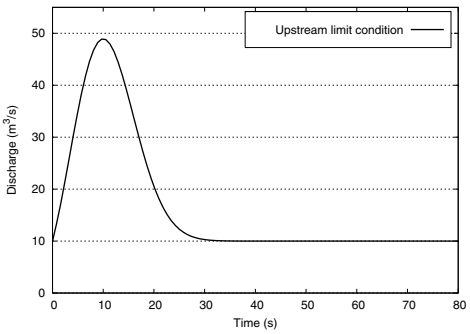


(b) Convergence of the bed elevation

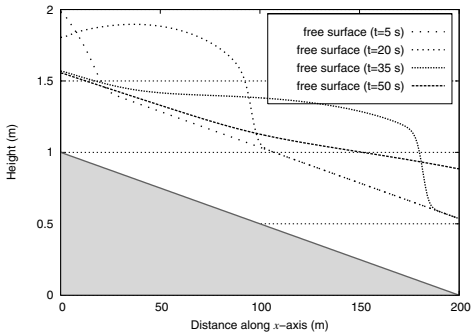
Fig. 3. Rectangular channel with a bump: convergence of the minimization.

measurements, leading to a high degree of uncertainty. We consider a 200 meters long rectangular channel with a constant slope of 0.5 %. The initial conditions consist in a steady flow initiated by the prescription of a constant discharge of 10 $m^3 s^{-1}$ at the inlet. Flooding is created by the modification of the upstream boundary condition: for a period of $T = 80$ seconds, it is defined by $\mathbf{q}_{\text{in}}(t) = 10 + 5t \exp\left(-\frac{(t-5)^2}{100}\right)$. The corresponding hydrograph is plotted in Fig. 4 (a). One can see the propagation of the wave in Fig. 4 (b), where the water surface profile is displayed at several time steps. The water depth is recorded continuously in time at a measurement point located at a given distance x_m from the upstream boundary of the domain. This measurement is used as an observation $h^{\text{obs}}(t)$ of the water depth during the flooding event. It makes it possible to define a cost function measuring the discrepancy between the water depth h and the observations h^{obs} at the point x_m:

$$j_2(\mathbf{q}_{\text{in}}) \;=\; \frac{1}{2}\int_0^T \left| h(x_m,t) - h^{\text{obs}}(t) \right|^2 dt \,. \tag{9}$$



(a) Upstream boundary condition



(b) Free surface evolution

Fig. 4. Rectangular channel with a constant slope: configuration.

From the initial hypothesis of a steady flow where the discharge is 10 $m^3 s^{-1}$, the experiment identifies the hydrograph that is at the origin of the wave. For that purpose, we use the variational data assimilation method. If the point of measurement is situated near the inlet ($x_m \leq 40$ meters), the identification of the boundary condition is very good. In Table 1, ε, the relative L^2 error between the identified discharge and the reference discharge is given for several values of the distance x_m. We conclude that the efficiency of the identification decreases when the distance of the point of measurement to the inlet increases.

The potential of variational data assimilation applied to river hydraulics was illustrated through two experiments. However, the tests were limited to the case of twin experiments where uncertainty on the data and assumptions on the model inputs are perfectly controlled. Assimilation of real observation data should be further

Table 1. Relative L^2 error on the identified boundary condition for several positions of the water depth measurement.

x_m	1 m	20 m	40 m	80 m	120 m	180 m	195 m
ε	0.45 %	0.50 %	0.53 %	2.16 %	5.21 %	8.95 %	10.1 %

investigated. The identification of a systematic model error would be valuable for an operational use of an hydraulic model. Moreover, the assimilation of observations of a different nature, such as trajectories or flood marks could bring additional information for the identification of control variables. A big asset of the adjoint method is that the same adjoint code can be used for all problems. Only the few lines of the adjoint code that correspond to the computation of the cost function are to be modified, which is very easy with an AD tool.

4 Catchment Hydrology

Recently, distributed hydrological models became an attractive approach for the modeling of watershed hydrology. Nevertheless, limited knowledge of model inputs (initial and boundary conditions, parameters) and observations of the hydrological response make the underlying problems of parametrization, calibration, sensitivity analysis and uncertainty analysis very challenging. However, sensitivity analysis is very often carried out using restricted, inaccurate and subjective techniques such as the brute force method. More sophisticated methods based on Monte Carlo simulations [26, 46, 47, 268] are now at the forefront in catchment scale hydrology. Nevertheless, they are based on sampling strategies of the parameter space and require many model runs. Therefore, even if analysis is easier to set up and the statistical framework is better suited for global sensitivity analysis, the computational cost is prohibitive if one wants to take into account the spatial variability of model parameters and its influence on the hydrological response. However, the rainfall-runoff relation is a typical case where the dimension of the system response to be analyzed is small compared to the number of input parameters to be prescribed. In this case, the adjoint model is very efficient in computing the gradient of a response function w.r.t. all parameters (see Cacuci [87] for a recent theoretical basis).

The underlying physics of MARINE, a model developed by Estupina et al. [160] is adapted to events for which infiltration excess dominates the generation of the flood. Rainfall abstraction by infiltration is evaluated using the Green Ampt model, and the resulting surface runoff (hillslope flow) is transferred using the Kinematic Wave Approximation (KWA). Lastly, river flow is routed with the full Saint-Venant equations, $1D$ or $2D$ depending on the valley configuration. The coupling with the river hydraulics component will not be discussed below. The simplification of mass and momentum conservation equations representing overland flow (KWA) is given by:

$$\frac{\partial h}{\partial t} + \frac{s^{1/2}}{n}\frac{\partial h^{5/3}}{\partial x} = r - i \,, \tag{10}$$

where h is the flow depth, n is the Manning roughness coefficient, s is the slope in the steepest direction, r is the rainfall rate, and i is the infiltration rate. A preliminary analysis of the digital elevation model computes a single steepest descent flow direction from four available directions for each cell. Then, (10) is solved using a simple explicit Euler scheme on the hillslope represented by a cascade of planes. Since the time step is not adjusted during the simulation, an *a priori* value u_m of the maximum velocity occurring during the simulation is used as a cutoff value in order to ensure convergence of the numerical scheme. In the right hand side of (10) representing the excess rainfall, the infiltration rate $i(t)$ is evaluated using the following procedure:

$$\begin{aligned} r < i \qquad & i = r \\ r \geq i \qquad & i = \frac{dI}{dt} = K\left(\frac{\psi\eta\theta}{I} + 1\right) , \end{aligned} \tag{11}$$

where I is the cumulative infiltration, K the hydraulic conductivity, ψ the suction force, η the porosity and θ the relative initial moisture deficit. Equation (11) is solved using an implicit Euler scheme, and the resulting fixed point equation using the Newton method.

MARINE, like most hydrological models, is far from being fully comprehensive and really simplifies the complex hydrological reality. In fact, improving physical understanding would increase the number of parameters to be calibrated. Since observation data is usually only an integrated flood hydrograph at the catchment outlet, appropriate parametrization, consistent choice of the degrees of freedom to be estimated and formulation of calibration criteria is mandatory. However, this requires an extensive knowledge of the effect of parameters variations on functions of the model state variables. Therefore, the potential of the adjoint method described in Sect. 2 and demonstrated by Margulis [352] and Li [336] for sensitivity analysis in hydrometeorology should be investigated for this specific application.

However, mathematical representations of catchment hydrology are very often strongly non-linear and involve multiple thresholds or switches due to the intrinsic nature of related conceptualization of the physical processes (rainfall, infiltration regimes, maximum infiltration capacity, etc.). Since introducing smoothing functions may lead to important inconsistencies between direct and adjoint models, AD seems to be an efficient alternative to obtain the required derivatives (i.e. sub-gradients). Hence, the adjoint of the MARINE model was developed using TAPENADE. The overall objective of the study was to conduct adjoint sensitivity and variational data assimilation experiments. Therefore, the flexible and efficient computer code structure proposed by Chavent [108] was adopted. Initial efforts were also dedicated to direct model analysis and source code modifications in order to identify the potential problems related to non-differentiable statements or conditional iterations in the computational approach. The necessary modifications to the code produced by TAPENADE are very similar to those described in the previous section and related to the very cautious storing and re-calculations which are not always necessary for the calculation of the adjoint variables. Before optimization, the adjoint code was respectively 10 and 100 times more expensive than the direct code in terms of computational cost and required memory. After a limited optimization, the memory footprint is divided by a factor of 2, and the code is 3 times faster.

Since the objective targeted by MARINE is an accurate representation of the rising limb of the flood, only the global response of the watershed (outlet hydrograph) will be analyzed. For flash flood events, the runoff coefficient and the maximum discharge are probably the most relevant quantities to be estimated. Thus, let us define

$$g_1 = \frac{\displaystyle\int_0^T q(t)\,dt}{\displaystyle\int_0^T\int_0^\Omega r(t)\;d\Omega\,dt} \qquad g_2 = \frac{\max\limits_{t\in[0,T]} q(t)}{q_{ref}} , \tag{12}$$

where $q(t)$ is the outlet discharge, and q_{ref} is the maximum discharge obtained when all rainfall is transformed into runoff (no infiltration). Sensitivities to g_2 are only defined during the rising limb and vanish during recession. The maximum discharge

q_{max} is calculated during the temporal integration. When $q(t)$ is greater than the current q_{max}, temporal increments in sensitivities correspond to sensitivities of $q(t)$ to model parameters. In addition, from the previously mentioned quantities, we can define g_3, a non-dimensionalized and normalized synthesis of the hydrograph,

$$g_3 = \frac{g_1}{\sqrt{{g_1}^2 + {g_2}^2}} + \frac{g_2}{\sqrt{{g_1}^2 + {g_2}^2}} \,. \tag{13}$$

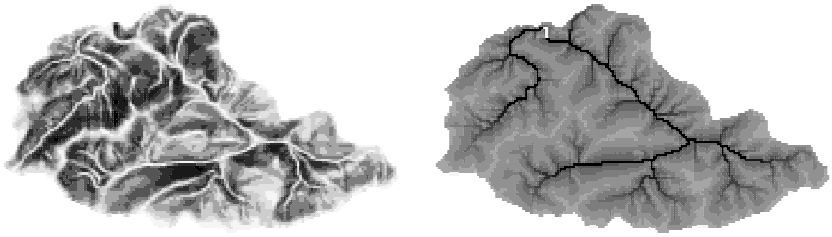
The examination of the sensitivities will allow us to investigate their hydrological meaning and analyze model behavior.

The chosen watershed is a very small catchment area (25 km^2) from the upper part of the Thoré basin in southern France. Given the basin features, no river flow routing was considered, and uniform land use and soil type is assumed to facilitate the analysis. The Manning roughness coefficient n and Green Ampt model parameters (K, ψ and η) are derived a priori from published tables using information on land use and soil type. Given the available information for the re-analysis of such catastrophic events, in order to assign a time step for the simulation, it was assumed that $u_m = 1\, ms^{-1}$ would be a reasonable value for the maximum velocity occurring during the simulation. Concerning the rainfall forcing, real radar data (HYDRAM from METEO FRANCE) was lumped over the area. Since the flow directions are computed before the model integration, accounting for the slope s in the sensitivity analysis would lead to systematic underestimation of its influence. Therefore, an adjoint sensitivity analysis was carried out w.r.t. model parameters K, ψ, η, θ, and n. Moreover, the response function can also be differentiated w.r.t. numerical or algorithmic artifacts such as u_m which appear neither in continuous nor discretized model equations. This is definitely an advantage of AD over the "equation-based" approach. However, ranking the sources of uncertainty (i.e. the sensitivity of model response to parameters) requires a normalization of the adjoint variables. The scaled sensitivities are

$$s_k = \frac{\partial g}{\partial \alpha_k} \cdot \frac{\alpha_k}{g} \,, \tag{14}$$

with g the response function, α_k the model parameter and s_k the normalized sensitivity. Since parameters are fully distributed, the L_2 norms were computed in order to rank parameters' influence on model response. A summary of the results obtained is given in Table 2. One can see that u_m has a greater impact on g_2 and g_3 than the real calibration parameters. In other words, the maximum discharge during the event is mainly driven by the assigned cutoff value. However, it was confirmed using the adjoint sensitivity analysis that its effect gradually alleviates and vanishes for larger values, the maximum discharge being driven mainly by the Manning roughness coefficient. This is a key issue in distributed rainfall-runoff modeling where given the simple conceptualization of the complex hydrological reality adopted for the formulation of mathematical models, internal variables and therefore estimated parameter values cannot be directly related to physical quantities. However, for this model, dedicated parametrization and modeling approach for the drainage network should lead to consistent internal variables. On the other hand, the results obtained with response function g_1 show that the effect of this threshold on the runoff coefficient can be neglected and that the partition of rainfall into runoff and infiltration is mainly driven by hydraulic conductivity K. The influence of the other infiltration

parameters (η, θ, ψ) cannot be distinguished given (11), and the importance of friction in favoring or limiting infiltration is greater than expected. The same analysis was carried out along a segment of the parameter space ($\theta \in [0, 1]$) and showed that the wetter the soil is, the shorter is the decay of i to K and the larger is the influence of parameter K over other infiltration parameters. In addition, a detailed analysis of the spatial and temporal patterns of the obtained sensitivities really provide physical insight into the model dynamics. In fact, all the cells of the watershed are solicited for infiltration from both direct rainfall and excess rainfall coming from upstream in the basin (*run-on*). The latter seems to be critical since the spatial pattern of sensitivity to all Green Ampt infiltration parameters is driven by the drainage network. For example, one can see in Fig. 5 the correspondence between slopes and sensitivity to K. Moreover, by varying rainfall duration and intensity, it was found that the influence of the parameters variability (variance of sensitivity matrix) is lower for short and intense storms. Furthermore, Fig. 6 exhibits the



(a) DTM slopes (b) Sensitivity to K

Fig. 5. Sensitivity to infiltration parameters and catchment slopes.

temporal patterns of the sensitivities to model parameters. The thresholds related to rainfall dynamics and different infiltration regimes lead to similar thresholds on temporal increments of adjoint variables. One can see in Fig. 6 that the event was divided into four periods, and again the results are in agreement with the infiltration excess overland flow mechanism. In fact, during period 1, rainfall totally infiltrates to the unsaturated zone without intervention of model parameters. Then, once rainfall intensity becomes important (beginning of period 2) the infiltration from direct rainfall is immediate and *run-on* develops. This can be noticed by the rising of the sensitivity to n. During period 2, rainfall duration and intensity remain limited and do not produce rising of the hydrograph. On the contrary during period 3, rainfall is so intense and its duration so important that a large amount of runoff is produced.

Table 2. Contributions (in %) of model parameters to the hydrological response.

	η, θ, ψ	K	n	u_m
g_1	17.55	34.09	13.23	3.75E-06
g_2	1.42	2.65	21.39	71.68
g_3	8.48	16.42	14.25	43.87

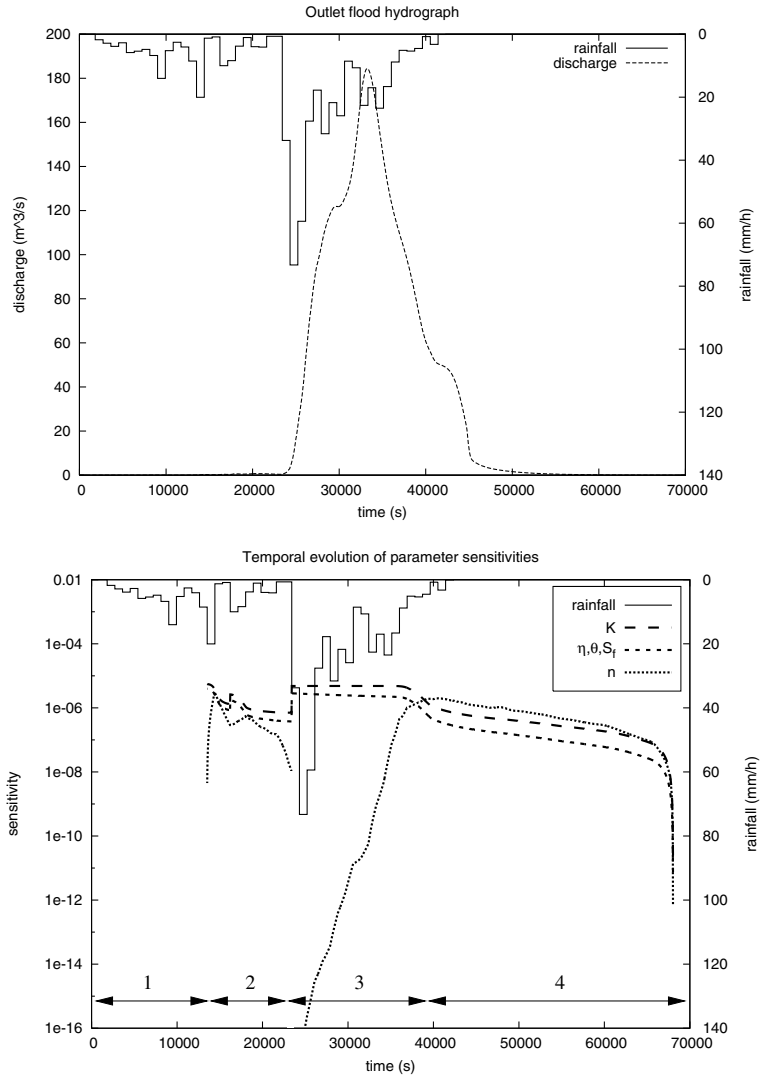


Fig. 6. Sensitivity to parameters and flood hydrograph.

The runoff coefficient and associated statistical moments were computed (mean and variance) and showed that during this period the runoff coefficient is very close to unity on the whole watershed. Therefore, the global influence of K remains constant and the sensitivity of other infiltration parameters decreases as the cumulative infiltration increases. At last, once intense rainfall stops, *run-on* produces infiltration mainly in the drainage network and progressively decreases.

For the test case we considered, the potential of the adjoint sensitivity procedure was demonstrated for model diagnostic and sensitivity analysis. However, the

behavior of the hydrosystem is analyzed only locally in the parameter space around effective values of model parameters and lead to sensitivities for a single point of the surface response. The influence of the point chosen for this local sensitivity analysis should be further investigated to draw more general conclusions. Indeed, results may depend on the type of soil, on the ratio between rainfall intensity (also duration) and hydraulic conductivity. The contribution of some form of global sensitivity analysis should be examined. The development of automatic calibration methods received much attention from the hydrological community, but problems of differentiability, parameter insensitivity, parameters interactions and multiple local optima make the use of gradient-based methods very difficult. However, compared to global conceptual models, the mathematical formulation of distributed and physically based models contains fewer thresholds and switches. Moreover, the spatial integration over the watershed as well as the temporal integration over the event prevent many non-differentiable regions in the hypersurface when a scalar response at the catchment outlet is analyzed. Systematic preliminary sensitivity analysis should always be carried out to identify the key parameters which affect the chosen hydrological response. This should avoid formation of flat portions of surface response (i.e. low sensitivities for one direction of the parameter space). From the observed trends, a multi-criteria calibration strategy could be developed to adjust some of the parameters for a given aspect of the response. Such investigations are in progress and require appropriate regularization approaches and strategies for the reduction of the control space to be developed. Then, the well-posed inverse problem can be solved using standard unconstrained optimization methods.

5 Conclusion

The adjoint method is a very efficient and flexible mathematical tool to calculate derivatives of a function of model state variable w.r.t. control variables. Two applications for flood modeling were presented in this prospective study. Variational data assimilation allows the identification of model parameters, initial and boundary conditions. Adjoint sensitivity analysis provides knowledge of the contribution of model inputs to the variations of some features of the solution, as well as physical insight into the model dynamics.

The practical implementation of the adjoint method is significantly facilitated by the use of efficient AD tools such as Tapenade. The development time is considerably reduced, and many human errors are avoided. However, with the current version of the tool, it is still necessary to manually modify the adjoint code. Unnecessary storage of variables can make the code require too much memory, and the adjoint of some non-differentiable statements need to be fixed manually.

Nevertheless, AD remains a very powerful technique used for the achievement of the adjoint method. It opens new trends for the construction of an hydrometeorological prediction chain and for future contributions concerning flood hazard forecast and mitigation.

Development of an Adjoint for a Complex Atmospheric Model, the ARPS, using TAF*

Ying Xiao[1,3], Ming Xue[1,2], William Martin[1], and Jidong Gao[1]

[1] Center for Analysis and Prediction of Storms, University of Oklahoma, Norman, OK, USA
{ying_xiao,mxue,wjmartin,jdgao}@ou.edu
[2] School of Meteorology, University of Oklahoma, Norman, OK, USA
[3] School of Computer Science, University of Oklahoma, Norman, OK, USA

Summary. Large-scale scientific computer models, such as operational weather predictions models, pose challenges for the applicability of AD tools. We report the application of TAF to the development of the adjoint model and tangent linear model of a complex atmospheric model, ARPS. Strategies to overcome the problems encountered during the development process are discussed. A rigorous verification procedure of the adjoint model is presented. Simple experiments are carried out for sensitivity study, and the results confirm the correctness of the generated models.

Key words: Adjoint model, tangent linear model, sensitivity analysis, atmospheric model, supercell thunderstorm simulation, ARPS, TAF, testing AD-generated code

1 Introduction

Adjoint models have been used in meteorology for applications such as four-dimensional variational data assimilation (4DVAR) and sensitivity studies for over two decades. However, operational implementations of 4DVAR did not start until recent years. Compared with the models used in other fields, operational weather prediction models are much more complex, and the problem sizes tend to be much larger. Thus the application of the associated adjoint models is often hindered by overwhelming programming tasks and high computational cost. The European Center for Medium-Range Weather Forecast (ECMWF) is the first center to implement an operational 4DVAR system [447], and the development of the adjoint code took many staff-years. The adjoint codes for a few mesoscale research models have also been developed in recent years, mostly by hand, and used for data assimilation and sensitivity studies [159, 192, 466, 527, 571]. Most existing adjoint models in meteorology contain only limited and often simplified physics processes. However, for the

* This work was supported by NSF ATM-0129892.

Advanced Regional Prediction System (ARPS) [560–562], a comprehensive regional atmospheric prediction model, an adjoint model with limited physics was developed by hand in the mid to late 1990's [192, 527]. Since then, the ARPS model has undergone significant changes. In this paper, we report our recent work on developing an adjoint code of the ARPS with full physics with the help of automatic differentiation tools, initially TAMC and more recently TAF.

In the rest of the paper, we denote the ARPS Nonlinear Model as ARPS NLM, the Tangent Linear Model as ARPS TLM, and the Adjoint Model as ARPS ADM. We assume that the reader knows the basic concepts of Fortran 90 grammar and Automatic Differentiation (AD). For more coverage of AD, please refer to the book by Griewank [225]. A good knowledge of meteorology is not necessary for the reader.

2 The Advanced Regional Prediction System

ARPS is a comprehensive nonhydrostatic regional to storm-scale atmospheric modeling and prediction system initially developed at the Center for Analysis and Prediction of Storms (CAPS) at the University of Oklahoma, under the support of the National Science Foundation Science and Technology Center (STC) program. The goal of ARPS is to serve as a system for mesoscale and storm-scale numerical weather prediction as well as a wide range of idealized studies in numerical weather prediction. It includes a real time data analysis and assimilation system, a forward prediction model, and a post-analysis package. The dynamic core of ARPS is based on the compressible Navier-Stokes equations that describes the atmospheric flow and uses a generalized terrain-following coordinate system. A variety of physical processes are taken into account in the model system.

The ARPS model equations are solved using second-order and fourth-order finite difference methods. The staggered Arakawa C-grid is used [10]. The split-explicit time integration method [310] is used, in which different time step sizes are used for integrating the fast acoustic modes and other slow modes. In the vertical direction, the acoustic mode is treated implicitly, as is the vertical turbulence mixing. The ARPS NLM includes a set of physical parameterizations, i.e., subgrid-scale and planetary boundary layer turbulence parameterizations, cloud microphysics, convective parameterizations, surface layer flux parameterizations, soil model, and longwave and shortwave radiation. Most of these physics parameterizations contain nonlinearities and on-off switches that are known to be sources of problems for adjoint codes [386, 559, 570]. Nonlinearities also exist with the model dynamics, such as in the advection process. For additional details on the ARPS model, the reader is referred to [560–562].

The forward prediction model, i.e., the ARPS NLM, on which the TLM, and ADM are based, contains about 40,000 lines of Fortran 90 source code excluding comments, blank lines, and I/O subroutines. Except for the convective parameterizations and radiation packages, and some rarely used subcomponents, the ARPS ADM includes all of the major components of the ARPS NLM. The adjoint code of ARPS is believed to be one of the most complicated in meteorology.

3 Transformation of Algorithms in Fortran (TAF)

TAF is a source-to-source AD tool for Fortran 90/95 codes developed by FastOpt (`fastopt.com`). It is the commercial successor to the Tangent Linear and Adjoint Model Compiler (TAMC) by Giering and Kaminski [207], which was used to develop the MM5-based 4DVAR system [466]. Compared with TAMC, TAF is much more robust, much faster and better maintained. At the earliest stage of our development, we used TAMC, but it was not able to generate the tangent linear or adjoint model from the top level driver of ARPS. The model had to be decomposed into sublayers, and the data dependencies across the layers had to be analyzed manually. This limitation greatly degraded the benefit of using AD tools. Similar problems also existed when we first applied TAF in mid-2002. By working closely with the TAF developers, we are now able to generate the TLM and ADM of the entire ARPS model with TAF, although all directly generated codes are not necessarily correct.

The AD tools are not completely reliable resulting from incorrect handling of some Fortran constructs due to limitations of or bugs in the tools. The exchanges with FastOpt often result in bug fixes to TAF or work-arounds. The nonlinear model code often had to be modified in many places to avoid mis-handling by TAF. In order to ease the maintenance of the adjoint codes, necessary changes are made to the NLM code instead of the generated TLM or ADM code whenever possible. Direct modification to the ARPS TLM and ADM codes is discouraged and is used as the last resort to address mainly performance issues. Most of the changes to the source code will be incorporated into the official version of ARPS NLM so that TAF can be applied to future versions of the NLM with much reduced effort.

4 Code Generation and Testing

We modified the NLM source code such that the AD tool (TAF) can perform the code transformation efficiently and correctly. We need to investigate the pitfalls in the NLM codes that may result in TAF failure.

4.1 Code Generation Issues

The ARPS NLM was first written in Fortran 77 and was converted to Fortran 90 by an automatic conversion tool. Some older features such as SAVE and GO TO statements remained. Even though TAF can handle some of these structures, codes generated with these structures often have problems. For example, we replace the SAVE statements with common block variables since the SAVE statement causes incorrect recomputation of intermediate NLM quantities and complicates the testing of the generated codes.

The overall performance of TAF is very impressive, and its use greatly sped up the development of the ARPS TLM and ADM. The robustness of TAF also improved with versions during the period we used it. However, there remain some problems with the generated codes. As with its predecessor TAMC, most of the TAF problems are related to the recomputation of the intermediate NLM values and flow dependency analysis of arrays. For example, TAF cannot distinguish the data dependency of individual elements and those of entire arrays. Consider the case

shown in Fig. 1 (line numbers are included for convenient reference). If line 1 in the code is removed, TAF assumes no dependency between the values f and x because of the assignment in line 2. TAF treats it as if the entire array x is set to a constant, i.e., to 1.0. Adding the statement in line 1 avoids this error.

```
  SUBROUTINE test (f, x)
     REAL :: f(100), x(100)
1    x(1) = x(1)
2    x(1) = 1.0  ! This assignment fools TAF
3    DO i = 1, 100
4      f(i) = x(i)**2
5    ENDDO
  END SUBROUTINE test
```

Fig. 1. Example code for which incorrect data flow analysis occurs with TAF.

Some large NLM subroutines had to be divided into smaller subroutines. Usually, when the subroutine contains nonlinear calculations, the size of the corresponding adjoint subroutine is increased, sometimes significantly. Large subroutines often cause compilation to fail at high optimization levels.

The main time integration loop of the NLM is divided into two levels to incorporate the two-level checkpointing scheme [224] supported by TAF. The scheme is designed to reduce the number of time levels that have to be stored in the main memory while at the same time avoiding excessive disk input and output. The NLM state is saved in checkpoint files every certain number of time steps, and the model states between the checkpoints are reconstructed by additional NLM integrations starting from the checkpointing times. Compared to saving every time step of the NLM base state in memory, the cost of implementing the checkpointing scheme is about one extra NLM integration time.

Floating point exception problems have been encountered with the TLM and ADM even though the NLM model works properly. This is because the valid domains of some functions are different from their derivatives. For example, the derivative of function $x^{1/2}$ ($sqrt$) is $1/2x^{-1/2}$, and there is a floating pointer exception if the derivative is executed with $x = 0$. Since the $sqrt$ function is widely used in the ARPS NLM, we replace it by the following pseudo-code $safesqrt$:

$safesqrt(x)$
  if $(x < a - small - value)$ then return $sqrt(a - small - value)$
    else return $sqrt(x)$

In summary, the procedure below generates the TLM and ADM models.

1. Apply TAF to generate TLM and ADM models
2. Test generated code
3. Identify the sources causing TAF errors in the ARPS NLM code
4. Modify the problematic codes in ARPS NLM
5. Go to step 1

4.2 Code Testing

Since the model integration can be seen logically as matrix multiplication, we denote the NLM model as matrix A, the TLM as matrix A', and thus the ADM as A'^T. The followings tests can be performed on ARPS TLM and ARPS ADM codes.

a. Linearity test of the TLM

$$A'(\lambda x)/(\lambda A'x) = 1, \lambda \in R\ . \tag{1}$$

b. Comparison with finite difference of NLM

$$(A(x+\delta x) - A(x))/(\delta x A'(\delta x)) \to 1, \text{when } \delta x \to 0\ . \tag{2}$$

c. Consistency between TLM and ADM

$$(y, A'x) = (A'x)^T y = x^T A'^T y = (x, A'^T y)\ , \tag{3}$$

where $(\cdot, \cdot)$ defines an inter product. Thus,

$$(y, A'x)/(x, A'^T y) \to 1\ . \tag{4}$$

Since AD tools apply the chain rule to transform the NLM source code [61, 207], for each active subroutine in the NLM, its tangent linear subroutine and adjoint subroutine are generated in the TLM and ADM, respectively. Therefore, the above test methods are also applicable to lower level subroutines in the NLM. Following the TAF convention, the adjoint (tangent linear) subroutines are named by adding the prefix *ad* (*g_*) to the corresponding subroutines of the NLM model. Furthermore, we may also test a block of codes if the TLM or ADM counterparts are available.

If the top level subroutine of the generated code fails the test, testing is performed on subroutines at successively lower levels until the source of the error is identified. The testing may also be applied to blocks of codes within a subroutine if this subroutine is identified as the source of problem.

After the initial TAF bug fixes and nonlinear model code changes, most of the remaining problems encountered are related to the recomputations of the NLM variables which are necessary for the calculations of the derivative code (these variables are called "required variables"). We add to the above three standard tests a recomputation test that verifies the correctness of recomputation in ADM. A variable is required if it is used in one of the following situations [168]:

a. Evaluation of the local Jacobian: E.g., for the function $f(x) = x^3$, the local Jacobian is $f'(x) = 3x^2$, and the value of x is required to evaluate $f'(x)$.
b. Evaluation of control flow information: The most common examples are the variables which determine loop steps, e.g., the variable n in do $i = 1, n$, and the variables in conditional statements, e.g., the variable x in if $(x > 1.0)$.
c. Evaluation of index expressions: An example is i in array $a(i)$.

Not all NLM variables are computed in the ADM, which may only compute variables indispensable for the ADM computation. The recomputation test is very important because it directly examines the consistency between the NLM and ADM.

To perform the test, we need to add code right before the invocation of the target subroutines to record and compare the values of arguments passed to these

subroutines. The testing driver first invokes TLM and records all NLM quantities passed to the subroutine being tested. After the execution of TLM, the driver program runs the ADM and compares the NLM values passed to the corresponding adjoint subroutine with those recorded in the first phase of the test. In the example illustrated by Fig. 2, subroutine *checkTF_f* is inserted into the NLM to record the value of x which is required in the computation of the adjoint subroutine *adf* in ADM. Subroutine *checkTF_f* stores x in a global stack data structure in memory. In the corresponding ADM, *checkTA_adf* is called to compare the current value of x computed in the ADM with the value stored in the stack. These two values should be identical.

```
subroutine NLM                  subroutine ADM
 . . .
! x is the required value          . . .
                                 call checkTA_adf(x, adx, ady)
call checkTF_f(x, y)
call f(x, y)                     call adf(x, adx, ady)
end subroutine NLM               end subroutine ADM
```

Fig. 2. Illustration of code modifications for the recomputation test.

To carry out the test, we need a test driver to do the following:

1) Provide the test data. For a scientific computation model, randomly generated input values may end up with a floating point exception because of physically unrealistic values. In our test framework, the driver invokes the NLM to generate the test data.
2) In most cases, the NLM, TLM, and ADM need to be invoked in a single test. The initial base state (including all arguments and global variables, i.e., variables declared in common block and variables with SAVE attribute) for all the models should be identical. For example, in the recomputation test, the driver first invokes the NLM and then invokes the ADM. The state of the model may be changed after the execution of the NLM because the value of some arguments and global variables may be overwritten by the NLM. Therefore, the driver must recover all base state variables prior to the invocation of the ADM.

4.3 Tangent and Adjoint Test Code Generator

Because a weather forecast model involves a large number of variables (the number of arguments of the main subroutine of ARPS NLM is more than 150), it is very tedious and error prone to write test codes and drivers by hand. We developed an automatic test code generator, called TATCG (Tangent and Adjoint Test Code Generator), to facilitate the testing. TATCG can generate test code and a driver for all four tests described above. TATCG is able to analyze the NLM, TLM, and ADM codes to detect all global variables whose values may be altered in the execution of the corresponding models so that it can generate code to restore their values. It can also perform code transformation if the test is to be applied to a block of codes instead of a subroutine. New subroutines that wrap the target code blocks are generated in addition to the test codes.

5 Testing Results

The correctness and efficiency of the ARPS ADM are tested with the data from a supercell thunderstorm simulation similar to the one documented in [561], except that it used externally supplied boundary conditions and included full physics (except for convective parameterizations and full radiation). We also carried out some simple adjoint sensitivity experiments for which we have a good idea of the correct solution. The tests were performed on an IBM Regatta P690 computer using a single Power4 1.1 GHz CPU. The experiments in Sect. 5.1 are performed at double precision, although single precision generally works as well. The experiments in Sect. 5.2 were performed in single precision, which is the default setting for APRS NLM.

5.1 Correctness Test

In this set of tests, we implemented the three standard testing schemes given in Sect. 4.2. We perturbed all independent variables of ARPS NLM by 1% of the base state values. If the test output is close to 1.0, the ARPS ADM is considered correct.

The physics components tested include the Kessler warm rain microphysics, surface layer flux parameterizations, subgrid-scale turbulence, planetary boundary parameterizations, and the soil-vegetation model. The forward model was first run to generate the nonlinear base state, and the tests were run from 6600 seconds through 7620 seconds of model time. The convective storm evolves on a time scale of about one hour. The number of integration time steps was 20 for the period. Given below are the computational times used by the TLM and ADM in terms of the NLM model time, and the results of the three standard tests, which are very close to 1.

TLM Model Computation time: 2 times NLM model
ADM Model Computation time: 11 times NLM model
Output for test a): 1.00000000000000000
Output for test b): 1.00000196449945489
Output for test c): 0.99999999999999412

5.2 Validation Test Using a Supercell Simulation

The validation test described next is used to check for the consistency of both the TLM and ADM with a single, small perturbation of the NLM. The TLM can always be compared with the difference of two slightly perturbed forward runs of the NLM. Because the ADM is linear, physical processes are temporally symmetric, so that, in some circumstances, results from the ADM can be directly compared with those from the TLM, and, therefore, also with the NLM with small perturbations.

To provide data for this test, we use a low-resolution supercell simulation. This simulation uses $35 \times 35 \times 19$ grid cells in the X-, Y-, and Z-directions, respectively, at a horizontal resolution of 2 km and a vertical resolution of 1 km. The time step size was 12 seconds. Full model physics were used, including ice microphysics. The model was initialized with a thermal bubble, and, after 1800 seconds of integration, a complex storm structure developed. Figure 3 shows the low-level wind field at this time. The NLM is run 100 time steps beyond the 1800 second point, and the model state is saved for each of these 100 steps for use in the TLM and ADM. Additionally,

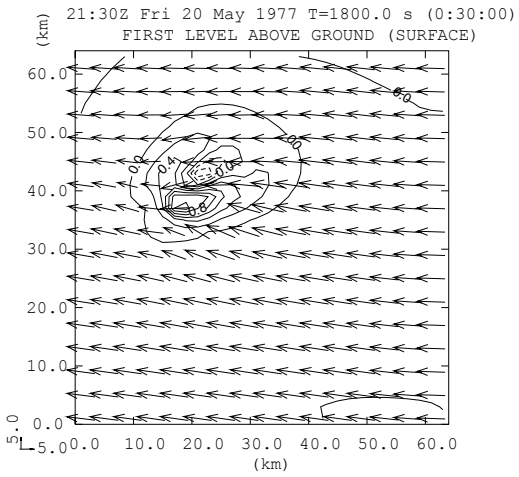


Fig. 3. Wind field 500 meters above the surface after 1800 seconds of integration of a supercell storm simulation. Contours are drawn every 0.2 m/s for updraft/downdraft amplitude. Wind vectors represent the horizontal wind component, with the length of the 5 m/s wind vector shown in the lower left corner.

the NLM is integrated twice from the 1800 second point with microphysics turned-off: Once with no perturbation and once with a perturbation in water vapor of 0.1 g/kg at a single grid cell. Microphysics were turned-off for these two runs because the microphysical modules for the TLM and ADM have not been fully debugged for long integration time, and we wish to compare as closely as possible a perturbation of the NLM with the TLM and ADM. The size and location of the perturbation is shown by a box drawn in Fig. 4–6. The perturbation was of a single 2 km by 2 km by 1 km grid cell centered at the first scalar variable level above ground.

Figure 4 shows the resulting forward sensitivity, or response, of the water vapor field at the end of 100 time steps of the NLM integration to a perturbation at the beginning of the period. The forward sensitivity is calculated by taking the difference between the perturbed and unperturbed forecasts and dividing by the magnitude of the initial perturbation. The result is nearly identical to that of the TLM integrated over the same 100 time steps. A similar pattern is obtained by a backward integration of the ADM for 100 time steps from the end of the total NLM run (1800 seconds plus 100 time steps). For this test, we specify the value of water vapor in the same box used for the forward sensitivity test, as the variable for which the backward sensitivity is sought. The backward sensitivity is then the value of differential changes in the forecast water vapor in the box divided by differential perturbations at the initial time. Figure 6 shows the resulting backward sensitivity pattern, a pattern which is nearly identical to those of NLM and TLM runs, except for a rotation of 180 degrees. This implies that the forward sensitivity of water vapor at a point is symmetric in this location with the backward sensitivity of water vapor from the same point. This is as expected since in the region in which the box for perturbation was chosen, horizontal wind gradients are small, and advection and

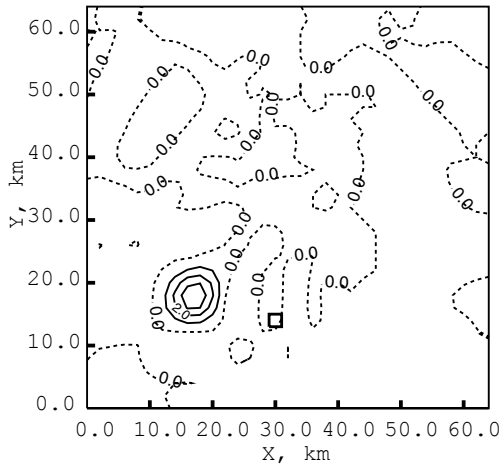


Fig. 4. Forward sensitivity of water vapor at 500 m above the ground to a water vapor perturbation 1200 seconds earlier, calculated from two NLM runs. Contours are drawn every 1%. Box drawn near (x = 30 km, y = 13 km) shows the location and size of the initial 1 km deep 0.1 g/kg perturbation. The maximum value is 3.79%.

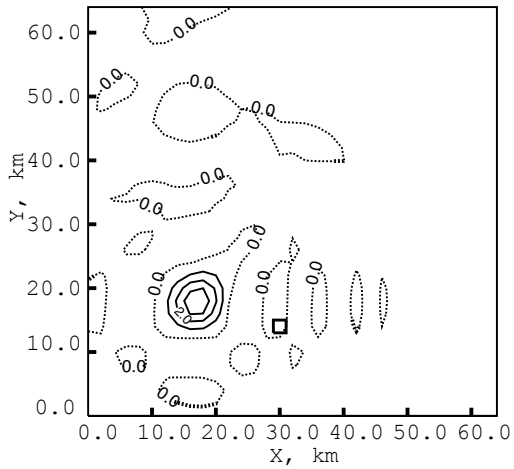


Fig. 5. Same as Fig. 4, except that the sensitivity values are from the TLM model. The maximum value is 3.80%.

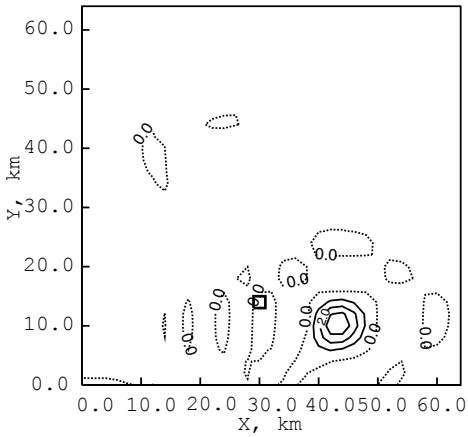


Fig. 6. Backward sensitivity of water vapor in the box drawn to the water vapor field 100 time steps earlier, calculated by the ADM. Contours are drawn every 1 %. The maximum value is 3.83 %.

diffusion are the primary physical processes at work, and these processes are quasi-linear for small perturbations. This test tends to confirm that the ADM and TLM are working correctly for the processes of advection and diffusion.

Sensitivities calculated from the ADM are exact for the model. For a data assimilation system, it is necessary to use the second order derivative as described by LeDimet et al. [326] if sensitivities with respect to the data are desired. This is facilitated by the ability of TAF to produce code for the second derivatives of a model.

6 Conclusion

In this paper, we reported the development of the tangent linear and adjoint codes for a complex, full physics mesoscale atmospheric model, the ARPS, with the help of an automatic differentiation tool, TAF. The issues and problems encountered during the development process are discussed, together with simple examples that illustrate the problems and solutions. A test procedure is presented, and a description of an automatic test code generation tool that assists the testing is given. We also presented experimental results from a supercell thunderstorm simulation that to a certain degree shows the validity of our tangent linear and adjoint model.

We will continue to debug the adjoint codes for the warm rain and ice microphysics packages which are causing instability for long time integrations even though they passed standard tests for a limited number of time steps.

The current ARPS ADM model is far from being efficient. This issue can be addressed by optimizing the ADM codes and through parallelization. The former can be accomplished by making compromises between recomputation and storage of the

intermediate NLM quantities, by removing redundant storage and recomputation, and by directly modifying the generated codes. We plan to apply the adjoint code to sensitivity studies of more realistic cases and develop a 4DVAR system based on the adjoint model.

Tangent Linear and Adjoint Versions of NASA/GMAO's Fortran 90 Global Weather Forecast Model

Ralf Giering[1], Thomas Kaminski[1], Ricardo Todling[2], Ronald Errico[2], Ronald Gelaro[2], and Nathan Winslow[2]

[1] FastOpt, Hamburg, Germany
www.FastOpt.com
[2] Global Modeling and Assimilation Office, NASA/GSFC, Maryland, USA
gmao.gsfc.nasa.gov

Summary. The NASA finite-volume General Circulation Model (fvGCM) is a three-dimensional Navier-Stokes solver being used for quasi-operational weather forecasting at NASA/GMAO. We use the automatic differentiation tool TAF to generate efficient tangent linear and adjoint versions from the Fortran 90 source code of fvGCM's dynamical core. fvGCM's parallelisation capabilities based on OpenMP and MPI have been transferred to the tangent linear and adjoint codes. For OpenMP, TAF automatically inserts corresponding OpenMP directives in the derivative code. For MPI, TAF generates interfaces to hand-written tangent linear and adjoint wrapper routines. TAF also generates a scheme that allows the tangent linear and adjoint models to linearise around an external trajectory of the model state. The generation procedure is set up in an automated way, allowing quick updates of the derivative codes after modifications of fvGCM.

Key words: Tangent linear model, adjoint model, source-to-source translation, Fortran-90, TAF, global weather model, parallelism

1 Introduction

Many applications in dynamic meteorology rely on derivative information. Talagrand [502] and Errico [158] describe the use of tangent linear (TLM) and adjoint (ADM) models for sensitivity analysis, stability (singular vector) analysis, variational data assimilation, and observation targeting. The use of second order derivative information for sensitivity analysis in the presence of observations, uncertainty analysis, and (Hessian) singular vector analysis is reviewed by Le Dimet et. al. [326].

Despite recent progress in automatic differentiation (AD) and early successful applications of AD tools in the support of TLM and ADM coding [64, 105, 106, 408, 563, 569, 572], hand-coding of TLMs and ADMs is still common in dynamic

meteorology. This is in contrast to oceanography, where fully automated generation of TLMs and ADMs of GCMs is becoming standard [191, 262, 290, 354, 516]. The purpose of the present study is to demonstrate the (after initial code preparations) automated generation of the TLM and ADM of a state of the art GCM using an AD tool.

The next section of this paper introduces the GCM, followed by a section describing the TLM and ADM generation. Sections 4 and 5 each address a particular challenge in the AD process: preserving the GCM's parallelisation capabilities and linearising around an externally provided trajectory. Section 6 discusses the TLM and ADM performance, and Sect. 7 shows an application example. Conclusions are drawn in the final section.

2 Finite-volume General Circulation Model

The NASA finite-volume General Circulation Model (fvGCM) [337–339] is a three-dimensional Navier-Stokes solver. The GCM was developed at NASA's Data Assimilation Office (DAO, now Global Modeling and Assimilation Office, GMAO) for quasi-operational weather forecasting. The model has various configurations. For the current study, we use two resolutions: the b55 production configuration, which runs on a regular horizontal grid of about 2 by 2.5 degree resolution (144×91 grid cells) and 55 vertical layers as well as the coarse a18 development configuration, with roughly 4 by 5 degree horizontal resolution (72×46 grid cells) and 18 vertical layers. The time step is 30 minutes, and typical integration periods vary between 6 and 48 hours depending on applications. The state of the model comprises three-dimensional fields of 5 prognostic variables, namely two horizontal wind components, pressure difference, potential temperature, and moisture.

3 Applying TAF to fvGCM

For GMAO's retrospective Data Assimilation System (GEOS-DAS, [569]), TLM and ADM versions of fvGCM's dynamical core are needed. Both the TLM and the ADM refer to the Jacobian of the mapping of the initial state onto the final state. While the TLM evaluates the product of the Jacobian times a vector of initial state perturbations in forward mode, the ADM evaluates the product of a (transposed) final state perturbation vector with the Jacobian in reverse mode. Further applications at GMAO such as sensitivity analysis, stability (singular vector) analysis, or chemical data assimilation also require TLMs and ADMs.

Transformation of Algorithms in Fortran (TAF) [207, 210] is a source-to-source transformation AD tool for programs written in Fortran 77-95, i.e., TAF generates a TLM or ADM from the source code of a given model. As the above applications differ in their sets of dependent and independent variables, TAF generates a TLM/ADM pair for each of the applications and two pairs for finite difference tests (rough and detailed). fvGCM is implemented in Fortran 90 and contains about 87000 lines of source code excluding comments. It makes use of Fortran-90 features such as *free source form*, *modules*, *derived types* and *allocatable arrays*.

Our standard approach to render a given code TAF-compliant consists of combining modifications to the model code with TAF enhancements. For instance, at two

places fvGCM's control flow structure has been simplified. Generation of an efficient store/read scheme for providing required values [207] (often denoted by *trajectory*) has been triggered by 41 TAF init directives and 75 TAF store directives. The ADM can be generated with and without a checkpointing scheme [210]. To support TAF's data dependence analysis, 11 TAF loop directives have been used to mark parallel loops. In total 204 TAF flow directives have been inserted to trigger generation of specified calling sequences [210]. For instance, TAF flow directives allow one to use the Fast Fourier Transformation (FFT) and its inverse in the TLM and ADM, respectively, which is more efficient than using a generated FFT derivative code [81, 210, 502]. In some subroutines, variables were allocated and/or initialised during the first call. This introduces a data flow dependence between the first and later calls which forces TAF to generate proper but inefficient recomputations. In order to avoid these recomputations we have moved the allocations and initialisations into extra module procedures which are not differentiated.

As a result of these initial modifications, the generation procedure for the derivative code is now fully automated. This is important to allow quick updates of the derivative code to future changes in the underlying model code. After each code change, the updated TLM and ADM have to be verified. Depending on the nature and the extent of the change, additional modifications may be necessary to keep the generated derivative code correct and efficient. Unfortunately, there is an error in the SGI-compiler (version 7.4.0) on the production machine (Origin 2000) which requires switching off the compiler optimisation for two files and reducing the optimisation level (-O2 instead of -O3) for six more files in the ADM. It turns out that the wrong compiler optimisation causes an inaccuracy of only a few percent, which is acceptable for many applications.

4 Parallelisation

For parallelisation, fvGCM can run on both shared and distributed memory architectures. On shared memory machines the model parallelises over vertical levels using OpenMP [418, 419] directives, and on distributed memory machines it parallelises over latitude bands using calls to the MPI library (MPI-1 [240] or MPI-2). There are even architectures that allow one to combine OpenMP and MPI, e.g., non-uniform memory access (NUMA) systems, to which our production machine, an SGI Origin 2000, belongs.

The challenge for AD is to transfer these parallelisation capabilities to the TLM and ADM. This task is not to be confused with AD applications based on sequential function codes, where parallelisation is restricted to the derivative code (see, e.g., [49, 79]).

For OpenMP, the model arranges all its parallelisation by repeated use of the `parallel do` directive. Analysis of such parallel loops is discussed in [207, 254]. The loop analyses in TAF have been extended to evaluate the `parallel do` directive. For each parallelised loop of the model, i.e. each loop furnished with an OpenMP directive, TAF can automatically generate the proper parallelisable TLM and ADM versions, including their OpenMP directives. By specifying either **-omp** or **-omp2** as command line options, the user selects the OpenMP standard to which the generated code conforms. Without either command line option, the code generation ignores OpenMP directives in the model code altogether, i.e. TAF produces sequential code.

```
      subroutine mp_send3d_ns(im, jm, jfirst, jlast, kfirst, klast, &
                              ng_s, ng_n, q, iq)
          implicit none
          integer im, jm
          integer jfirst, jlast
          integer kfirst, klast
          integer ng_s      ! southern zones to ghost
          integer ng_n      ! northern zones to ghost
          real q(im,jfirst-ng_s:jlast+ng_n,kfirst:klast)
          integer iq
    ! Local:
          integer i,j,k
          integer src, dest
          integer qsize
          integer recv_tag, send_tag
          ncall_s = ncall_s + 1
    ! Send to south
          if ( jfirst > 1 ) then
            src = gid - 1
            recv_tag = src
            qsize = im*ng_s*(klast-kfirst+1)
            nrecv = nrecv + 1
            tdisp = igonorth*idimsize + (ncall_s-1)*idimsize*nbuf
            call mpi_irecv(buff_r(tdisp+1), qsize, MPI_DOUBLE_PRECISION, &
                           src, recv_tag, commglobal, rqest(nrecv), ierror)
            dest = gid - 1
            qsize = im*ng_n*(klast-kfirst+1)
            tdisp = igosouth*idimsize + (ncall_s-1)*idimsize*nbuf
            call BufferPack3d(q, 1, im, jfirst-ng_s, jlast+ng_n, kfirst, klast, &
                                 1, im, jfirst, jfirst+ng_n-1,   kfirst, klast, &
                                 buff_s(tdisp+1))
            send_tag = gid
            nsend = nsend + 1
            call mpi_isend(buff_s(tdisp+1), qsize, MPI_DOUBLE_PRECISION, &
                           dest, send_tag, commglobal, sqest(nsend), ierror)
          endif
    ! Send to north
          if ( jlast < jm ) then
            ...
          endif
          end subroutine mp_send3d_ns
```

Fig. 1. Example of a wrapper routine for MPI-communication. To save space the kernel of the lower `if-then-endif` construct (indicated by the dots) is not displayed. It works analogously to the kernel of the upper `if-then-endif` construct.

For MPI-communication, rather than including calls to the library routines directly into the main code of the GCM, there is an additional layer of routines in between. These wrapper routines are called from the main code of the model and arrange all MPI-communication internally. All wrappers plus a few utility routines form a Fortran-90 module (named `mod_comm`). As an example, Figure 1 shows the MPI-1 version of a wrapper routine that exchanges a three-dimensional field across boundaries of latitude bands. It does all the necessary bookkeeping for indices, and the packing of the relevant section of the field `q` using the utility routine `BufferPack3d`, which is also part of `mod_comm`.

For a subset of MPI-1, Faure and Dutto [166, 167] address handling in forward and reverse mode AD, respectively. Carle and Fagan [97] as well as Bischof and Hovland [60] address handling of MPI-1 in forward mode AD. In forward mode, TAF handles most relevant MPI calls. Among the MPI library routines used by fvGCM,

```
  subroutine mp_send3d_ns_ad(im, jm, jfirst, jlast, kfirst, klast, &
                             ng_s, ng_n, q, iq)
      implicit none
      integer im, jm
      integer jfirst, jlast
      integer kfirst, klast
      integer ng_s      ! southern zones to ghost
      integer ng_n      ! northern zones to ghost
      integer iq        ! Counter
      real    q(im,jfirst-ng_s:jlast+ng_n,kfirst:klast)
! Local:
      integer i,j,k
      integer src
      integer recv_tag
      ncall_r = ncall_r + 1
! Receive from south
      if ( jfirst > 1 ) then
        nread = nread + 1
        call mpi_wait(rqest(nread), Status, ierror)
        tdisp = igonorth*idimsize + (ncall_r-1)*idimsize*nbuf
        call BufferUnPack3dx(q, 1, im, jfirst-ng_s, jlast +ng_n  , kfirst, klast, &
                                1, im, jfirst     , jfirst+ng_n-1, kfirst, klast, &
                                buff_r(tdisp+1))
      endif
! Receive from north
      if ( jlast < jm ) then
       ...
      endif
      if (ncall_r == ncall_s) then
        call mpi_waitall(nsend, sqest, Stats, ierror)
        nrecv = 0
        nread = 0
        nsend = 0
        ncall_s = 0
        ncall_r = 0
      endif
  end subroutine mp_send3d_ns_ad
```

Fig. 2. ADM version of the wrapper in Fig. 1. Again the kernel of the second `if-then-endif` block is not displayed (indicated by the dots) to save space.

the only one missing is *MPI_Allreduce* with *MPI_MAX* as a reduction operation. As MPI versions of both the TLM and ADM are needed, we chose to handle MPI via a different approach (see also [262]): Adjoints of all wrappers have been hand-coded. In the TLM, most of the model wrappers can be reused; only a single TLM wrapper had to be hand-coded. As the actual MPI library calls are carried out within the wrappers, this approach works independently of the MPI standard (MPI-1 or MPI-2). The ADM version of the wrapper in Fig. 1 is shown in Fig. 2.

Inclusion of the proper calling sequences for TLM and ADM versions of the wrappers into the generated TLM and ADM is triggered by TAF flow directives. Specifying TAF flow directives for a routine makes TAF analyses ignore the code of the routine and instead use the information provided by the directives. The flow directives for the wrapper in Fig. 1 are shown in Fig. 3. The first word, `!$taf`, is a keyword indicating a directive to TAF. The leading "`!`" makes the Fortran compiler ignore the directive. The next words, `module mod_comm subroutine mp_send3d_ns`, indicate the module and the routine to which the flow directives refer. The first two directives indicate the input and output arguments of the subroutine. The numbers refer to the position of an argument in the argument list, i.e., arguments 1 to 10

```
!*********************************************************************
! mp_send3d_ns
!*********************************************************************
!$taf module mod_comm subroutine mp_send3d_ns   input = 1,2,3,4,5,6,7,8,9,10
!$taf module mod_comm subroutine mp_send3d_ns  output =
!$taf module mod_comm subroutine mp_send3d_ns  active = 9
!$taf module mod_comm subroutine mp_send3d_ns  depend = 1,2,3,4,5,6,7,8  ,10
!$taf module mod_comm subroutine mp_send3d_ns  adname = mod_comm::mp_send3d_ns_ad
!$taf module mod_comm subroutine mp_send3d_ns ftlname = mod_comm::mp_send3d_ns
!$taf module mod_comm subroutine mp_send3d_ns common mp_3d_ns output = 1
!$taf module mod_comm subroutine mp_send3d_ns common mp_3d_ns active = 1
```

Fig. 3. TAF flow directives triggering generation of calls to TLM and ADM versions of Fig. 1.

(in fact all arguments) are input, and none is output. For the current routine the output directive may also be omitted, as the empty set is the default for this type of directive. The next two directives indicate active and required arguments [207]. The next two directives indicate the names of the TLM and the ADM versions of the routine. Since the name of the TLM version corresponds to the name of the original routine, TAF recognises that the routine is linear. The last two directives refer to elements of the common block `mp_3d_ns`; their syntax is similar to that of the directives for the argument list.

With the flow directives, the generated TLM and ADM can be linked to the hand-written wrappers. Hence, the generation procedure can be fully automated and produces four TLM/ADM versions, one for each of the combinations OpenMP on/off and MPI on/off.

5 Linearising around an External Trajectory

In dynamic meteorology it is typical to run the TLM/ADM on a coarser resolution than the forecast model. Also a set of processes called physics (e.g. parametrisations of clouds and rain, surface drag, or vertical diffusion) is usually not included in the derivative code. This reduces the computational demand and avoids potential problems arising from numerical instability. To compensate for this approximation, one linearises along an external trajectory of the state computed by the complete high resolution model. Technically this is achieved by making the (coarse grid) TLM/ADM integration periodically read in a regridded version of the external state and overwrite the internal state. However, to an AD tool, overwriting makes the final model state appear independent from the initial model state, as the data flow is interrupted. Straightforward use of AD would result in an erroneous TLM/ADM. To solve this problem, we exploit TAF's flexibility in setting up a store/read scheme for providing required variables. A combination of TAF init, store, and flow directives essentially hides the overwriting from TAF analyses and includes proper calls to the routines that provide the external trajectory. The resulting trajectory versions of the TLM and ADM are no longer proper linearisations of the coarse resolution model, unless they are run with an external trajectory provided by the model itself.

Table 1. TLM and ADM run time in multiples of model run time.

Platform/Setup	resolution	TLM	ADM	ADM-noopt
Linux Intel 4	a18	1.5	7.0	-
SGI-OpenMP-1/8 threads	b55	1.5	10.8	20.6
SGI-MPI-1/8 threads	b55	1.5	3.9	12.6

6 Performance of Generated Code

The performance of the sequential TLM and ADM versions has been tested on a Linux PC (P4 3GHz Processor, 2 GByte memory, Intel Fortran Compiler 8.0) and on a SGI Origin 2000 (Compiler version 7.4.0). On the Linux PC we could only run the coarser a18 configuration (see Sect. 2), because of memory limitations. On the SGI we have done separate tests for OpenMP-1 (8 threads) and for MPI-1 (8 processors). We ran the ADM in both the inaccurate version with full compiler optimisation and the accurate version with reduced compiler optimisation. The integration period was six hours. We ran the configuration without check-pointing and without reading an external trajectory, i.e., both the TLM and ADM integration include a model integration. All required variables were stored in memory.

The performance numbers are listed in Table 1. It is common to quantify the CPU time of the derivative code in multiples of the CPU time of a function evaluation (model integration). For the ADM performance, it is striking that the OpenMP version performs so much worse than the MPI version. We think that this is due to many critical sections, which have to be generated, because the OpenMP 1.1 standard does not support array reductions. As the OpenMP 2.0 standard [419] allows array reduction, TAF generated code conforming to OpenMP 2.0 avoids critical sections. Unfortunately, we were not able to run that version of the generated code on SGI, due to a problem in the compiler's handling of OpenMP 2.0. It is also remarkable that the ADM value for MPI on SGI is much better than on Linux. We attribute this to the difference in grid resolution between the two configurations, which affects the ratio of memory accesses to computations. While both resolutions need to access the same number of arrays in memory (albeit of different sizes), the coarse a18 resolution on Linux does fewer operations. This conjecture is supported by initial ADM tests of the MPI-1 version in a18 resolution on SGI, which show a performance ratio close to the one for Linux. Finally, SGI values for the ADM with reduced optimisation are considerably slower than those with full optimisation for both OpenMP and MPI.

Figure 4 shows the speedup for OpenMP-1, for the model itself, the TLM and the ADM for 2, 4, 6, and 8 threads. The speedup for n threads is defined by the quotient of the run times for 1 and n threads. The ideal speedup ignoring communication overhead is also indicated. While the TLM speedup is almost as good as that of the GCM, the ADM speedup lags behind. As mentioned above, this is presumably due to the critical sections, and OpenMP-2 is expected to yield a better speedup.

In the MPI case (Fig. 5), the TLM and ADM speedup is similar to that of the GCM code, with the ADM speedup being slightly better.

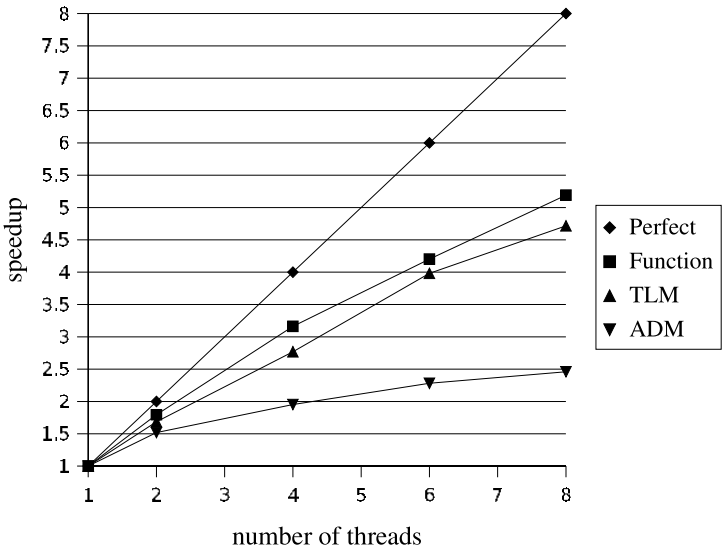


Fig. 4. Speedup for OpenMP-1.1 configuration.

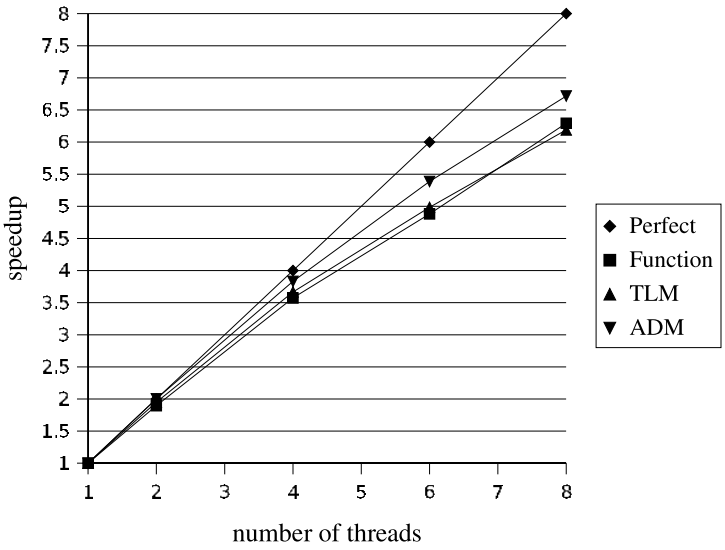


Fig. 5. Speedup for MPI-1 configuration.

7 Application Example

As an example for an application of both the TLM and ADM, we compute the leading singular vectors of a 24 hour integration. A singular vector is an eigenvalue of A^*A, where A is the Jacobian of the function mapping the initial state onto the

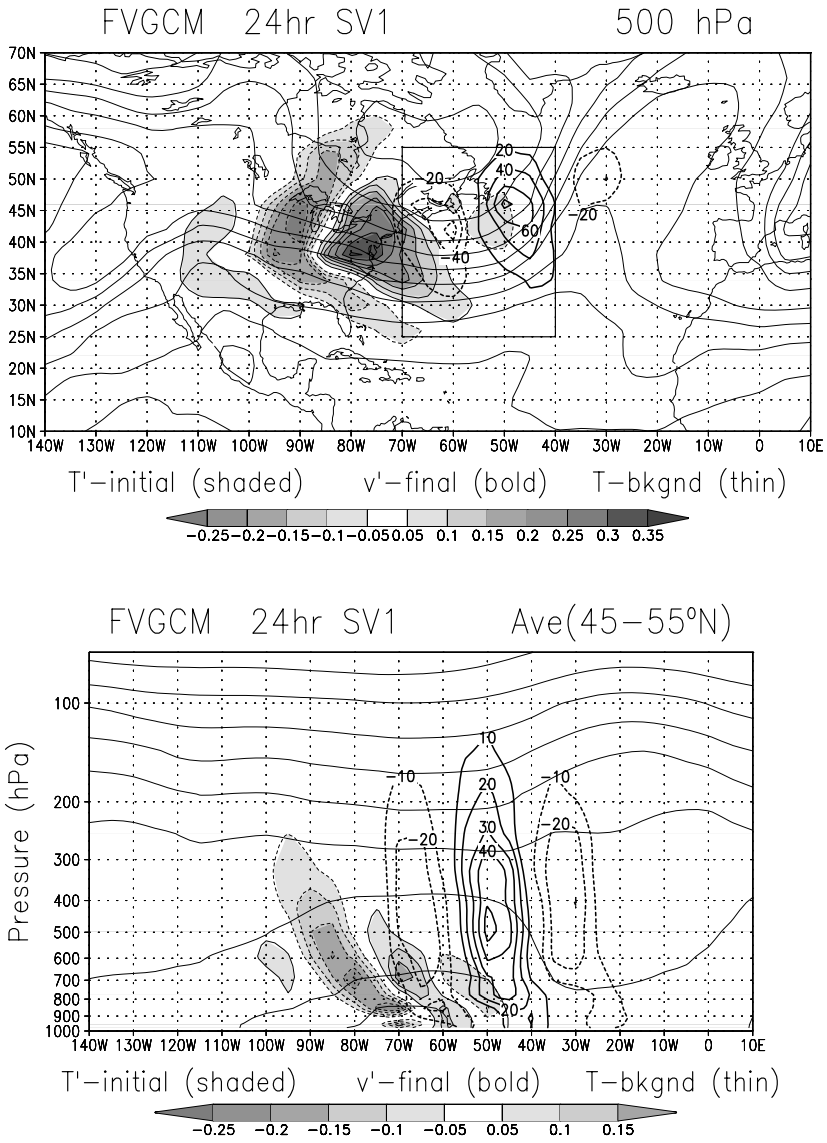


Fig. 6. Leading singular vector for 24 hour integration. See text for details.

final state, and A^* is its adjoint. Via the definition of the adjoint, the singular vector depends on the norms in state space at initial and final times. The leading singular vector is the initial time perturbation that amplifies the most (in the sense defined by the pair of norms). For the singular vector computation the automatically generated TLM and ADM have been extended by hand-coded TLM and ADM versions of

formulations for simple drag and vertical mixing (important damping processes). The coarse a18 resolution in the OpenMP version with reduced optimisation for the ADM has been used. ARPACK [332] is used to solve the eigenvalue problem.

Figure 6 shows the dominant singular vector for total energy norms at initial and final times. Grid points outside the target area indicated by the light black rectangle on the upper panel and in the top five vertical layers do not contribute to the norm at final time. The upper panel shows a horizontal view on the 500 hPa pressure level and the lower panel a vertical cross section averaged over the latitude band from 45 to 55 North. The shaded areas show the initial temperature component of the singular vector (temperature perturbation in K), and the thin black contours the background temperature (with 0.2 degree contour interval in the top panel and 0.1 in the bottom panel). The bold black contours show the evolved perturbation in the northward wind speed in m/s. As expected for a midlatitude perturbation, it evolves from a small scale, tilted structure at initial time to a larger scale structure at final time.

8 Conclusions

We have presented the generation of TLM and ADM versions of the dynamical core of fvGCM by means of TAF. After initial preparations, the generation process is fully automated. This automation is important, as it simplifies adaptation of the TLM and ADM to future changes of and extensions to the GCM code. A TLM integration takes the run time of about 1.5 model integrations. For the ADM that number varies with the configuration of the GCM in terms of resolution and parallelisation strategy. In the most favourable case (fine resolution, MPI-1, no check-pointing, problems with the Fortran compiler ignored) a factor of 3.9 is achieved.

Challenges such as transferring the model's parallelisation capabilities to the TLM and ADM, or linearising around an external trajectory have been overcome. We cannot think of any fundamental obstacle that could seriously hamper automatic generation of TLMs, ADMs, and even Hessian codes of models in dynamic meteorology.

Efficient Sensitivities for the Spin-Up Phase*

Thomas Kaminski, Ralf Giering, and Michael Voßbeck

FastOpt, Hamburg, Germany
www.FastOpt.com

Summary. In geosciences, it is common to spin up models by integrating with annually repeated boundary conditions. AD-generated code for evaluating sensitivities of the final cyclo-stationary state with respect to model parameters or boundary conditions usually includes a similar iteration for the derivative statements, possibly with a reduced number of iterations. We evaluate an alternative strategy that first carries out the spin-up, then evaluates the full Jacobian for the final iteration and then applies the implicit function theorem to solve for the sensitivities of the cyclo-stationary state. We demonstrate the benefit of the strategy for the spin-up of a simple box-model of the atmospheric transport. We derive a heuristic inequality for this benefit, which increases with the number of iterations and decreases with the size of the state space.

Key words: Spin-up, sensitivities, source-to-source transformation, TAF, implicit function, atmospheric transport

1 Introduction

In geosciences, it is common to spin up models to a cyclo-stationary state with periodic boundary conditions (forcing) as is illustrated by Fig. 1. For instance, to simulate the global carbon dioxide distribution in the atmosphere, one runs an atmospheric transport model with a repeated seasonal cycle of carbon dioxide fluxes at the Earth's surface [323]. The spin-up is completed once the simulated seasonal cycle of atmospheric carbon dioxide no longer changes from one year to the next. Other examples are simulations of the global thermo-haline ocean circulation [390] or of the terrestrial biosphere [481]. Especially for coupled model integrations, required spin-up times are often prohibitively long. Sophisticated techniques have been devised to reduce them (see, e.g, [287, 350, 470]).

* Part of this work has been carried out in the project CAMELS, supported by the EU under contract no. EVK2-CT-2002-00151 within the 5th Framework Programme for Research and Technological Development.

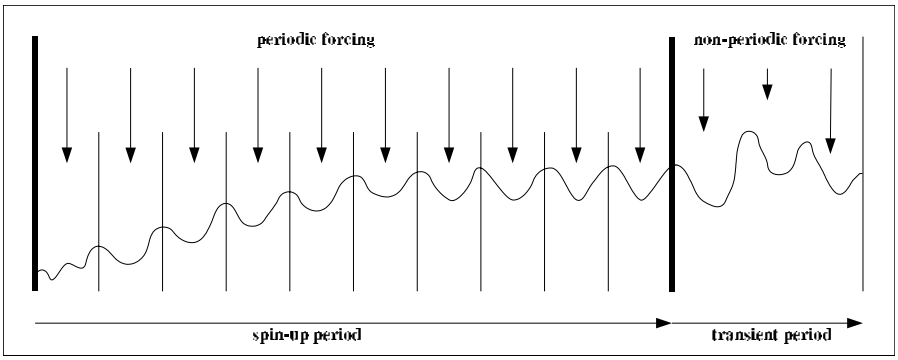


Fig. 1. Schematic representation of spin-up phase.

Often, the sensitivity of the equilibrium state (spin-up sensitivity) with respect to the forcing or internal parameters of the model is required. This sensitivity might be interesting per se, or it might be a part of an extended sensitivity computation of other quantities that depend on the equilibrium state through a transient integration (see Fig. 1).

Spin-ups are also carried out in fields other than geosciences. For example, the simulation of a steady aerodynamic flow around an airfoil constitutes a special case of a spin-up integration with constant forcing (reflecting the airfoil's shape and the far field) and a period of one time step. Spin-up sensitivities provide important information for design optimisation.

Accurate spin-up sensitivities can be provided by automatic differentiation (AD [225]). However, applying AD in a straightforward way results in derivative propagation through the entire spin-up process, which is even more costly than the spin-up itself. Often, the computation is only rendered feasible at the cost of approximations in the model formulation. The terrestrial biosphere model BETHY [311, 312], which forms the core of the Carbon Cycle Data Assimilation System (CCDAS, see `http://CCDAS.org`), provides an example of such an approximation. Spinning up a pool for slowly decomposing organic carbon in soil is avoided by introducing a so-called β factor to parameterise the combined effect of both equilibrium pool size and turn-over time on the release of carbon dioxide [457, 471]. Using this approximation, only a fast decomposing soil carbon pool has to be spun up, which reduces the spin-up time from thousands of years [481] to five years.

The present paper evaluates an alternative approach to sensitivity calculation for the spin-up phase, the *full Jacobian* approach, which requires the Jacobian for only a single year integration in equilibrium. The layout for the remainder of the paper is as follows: Sect. 2 formalises the spin-up process, discusses various ways of computing spin-up sensitivities, and presents the *full Jacobian* approach. Section 3 describes its implementation, and Sect. 4 demonstrates its application to the spin-up of a simple atmospheric transport model. Section 5 analyses the computational efficiency of the *full Jacobian* approach. A summary and an outlook are given in Sect. 6.

2 Spin-up Sensitivities

Formally, integration of a model over a forcing-period of, say, one year can be represented by a function $f : R^p \times R^n \to R^n$ mapping the state x at January 1, 0 am, to next year's state at the same time, y:

$$y = f(b, x) \ , \tag{1}$$

where b denotes input quantities other than the initial state such as boundary conditions or internal parameters of the model. The spin-up is the iteration of (1), with y taking the role of x for the subsequent iteration. A necessary condition for terminating the spin-up is convergence of y to a fixed point x_e (equilibrium state), i.e. the equation

$$x_e = f(b, x_e) \tag{2}$$

must hold within a specified accuracy.

The *spin-up sensitivity* is the derivative of the equilibrium state x_e with respect to b. AD of the source code for (1) can provide derivative code to compute this sensitivity. The most obvious approach (*standard*/black-box AD) is to differentiate the code of the entire spin-up phase, say in forward mode [225] of AD. Using, for instance, our AD-tool Transformation of Algorithms in Fortran (TAF [207, 210]), one would specify b as an independent/input variable and x_e as a dependent/output variable. TAF then generates code that iterates

$$\frac{dy}{db} = \frac{\partial f}{\partial b} + \frac{\partial f}{\partial x} \cdot \frac{dx}{db} \tag{3}$$

along with (1). In this derivative code each relevant function code statement is preceded by the corresponding derivative statement. For a given combination of b and initial value of x, the spin-up sensitivity is evaluated by running the derivative code with $\frac{dx}{db}$ initialised to zero. In case there is only one stable equilibrium, the results of both the spin-up iteration and the derivative iteration are independent of the initial value of the state, x. Griewank [225] provides a formal analysis of the convergence criteria for (3).

However, there are more efficient strategies of providing spin-up sensitivities, based on the Implicit Function Theorem for (2): Presuming that f is sufficiently regular and $\frac{\partial f}{\partial x}(b, x_e(b)) - Id$ is non-zero for a given b, then there exists a regular function $\tilde{b} \mapsto x_e(\tilde{b})$ around b, and its derivative satisfies

$$0 = \frac{\partial f}{\partial b}\,(b, x_e(b)) + \left(\frac{\partial f}{\partial x}(b, x_e(b)) - Id\right) \frac{dx_e}{db}(b) \ , \tag{4}$$

where Id denotes the identity in R^n, and $\frac{dx_e}{db}$ is determined by local properties of f around b and the equilibrium state x_e. [179, 225] suggest a *delayed derivative propagation* strategy (two-phase-AD), which does not turn on derivative propagation until the iteration of (1) converges well. Bücker et al. [84] apply this strategy to a CFD code.

Christianson [111,112] analyses reverse mode (adjoint) AD of the iteration of (1) and suggests an efficient alternative adjoint, which only uses the required values (trajectory) [207] from the last iteration of (1) and thus considerably reduces the necessary resources for storing/recomputing required values. TAF implements automatic generation of the Christianson scheme, triggered by a TAF-loop directive [210].

```
subroutine f( p, b, n, x, y)
integer p, n
real b(p), x(n), y(n)
```

Fig. 2. Header of subroutine (file `f.f`) implementing (1).

All above strategies (*standard*, *delayed derivative propagation*, and *Christianson*) are matrix-free, i.e. they employ products of the Jacobian $\frac{\partial f}{\partial x}$ with a vector to solve (3) for $\frac{dx_e}{db}$. The present study explores the alternative *full Jacobian* strategy of first running the spin-up, then computing the full Jacobian, i.e. $\frac{\partial f}{\partial b}(b, x_e(b))$ and $\frac{\partial f}{\partial x}(b, x_e(b))$, and finally solving (4) for $\frac{dx_e}{db}$.

3 Implementation

We sketch a Fortran implementation of the *full Jacobian* approach based on our AD tool TAF [207, 210]. It is instructive to consider first the simpler case of a sensitivity calculation that is restricted to the spin-up phase.

We start from the code of a single year integration, more precisely an implementation of (1), a subroutine form of f. The header of the subroutine is shown in Fig. 2. A single code for evaluating the two Jacobians required by (4), i.e. $\frac{\partial f}{\partial b}(b, x_e(b))$ and $\frac{\partial f}{\partial x}(b, x_e(b))$, is generated by applying TAF with command line options **-forward -pure -toplevel f -input b,x -output y -jacobian m -ftlmark _dbx f.f**, where $m = p + n$ (the sum of the dimensions of b and x) and `f.f` contains the source code of the subroutine `f`. The option **-pure** invokes TAF's pure mode, i.e. the derivative code does not include a function evaluation, and function code statements are only included where necessary to provide required values. TAF generates a subroutine `f_dbx` that evaluates the Jacobian.

A simple driver program that runs the code for the spin-up and its derivative is shown in Fig. 3. Subroutine `spinup` performs the spin-up, including the iterative call of the subroutine `f` and a termination condition. The field `x0` contains the initial value of the state, and the field `xe` is its equilibrium value. The matrices $\frac{dx}{db}$ and $\frac{db}{dx}$ are initialised to zero, and $\frac{dx}{dx}$ and $\frac{db}{db}$ to the identities in R^n and R^p, respectively. The Jacobian is evaluated for $x = x_e$, i.e. at the equilibrium, then the result is split into the two Jacobians and passed to a solver routine which finally returns $\frac{dx_e}{db}$.

For cases in which the Jacobian evaluation in reverse mode is preferable, the TAF command line and the driver need to be modified. The command line arguments **-forward -ftlmark** have to be replaced by **-reverse -admark**. In the driver, it is now the field `y_dbx` that must be initialised, and the Jacobian is returned in the fields `b_dbx` and `x_dbx`.

We now address the case in which the spin-up is part of a larger computation, and some sensitivity involving the entire computation is needed. We apply TAF to the source code of the entire computation. To handle the spin-up, we use the TAF flow directives for the subroutine `spinup` (see Fig. 4) which trigger inclusion of a calling sequence for an externally provided derivative routine of `spinup`. The forward mode then calls a routine `spinup_tl`, while the reverse mode calls a routine `spinup_ad`. For details on TAF flow directives see [211].

```
      real b(p), x0(n), xe(n), x(n), y(n)
      real x_dbx(p+n,n), b_dbx(p+n,p)
      real y_dbx(p+n,n), y_dx(n,n), y_db(p,n), x_db(p,n)
      b = ...
! spin-up
      x0 = 1.
      call spinup( p, b, n, x0, xe)
! initialisation of derivative objects
      do j =1,n
         do i=1,p+n
             x_dbx(i,j)=0.
         enddo
         x_dbx(p+j,j)=1.
      enddo
      do j =1,p
         do i=1,p+n
             b_dbx(i,j)=0.
         enddo
         x_dbx(j,j)=1.
      enddo
! Jacobian evaluation
      x=xe
      call f_dbx( p, b, b_dbx, n, x, x_dbx, y, y_dbx)
! separating the Jacobians
      do i=1,p
         y_db(i,:) = y_dbx(i,:)
      enddo
      do i=1,n
         y_dx(i,:) = y_dbx(p+i,:)
      enddo
! solve for spin-up sensitivity
      call solve (n, p, y_dx, y_db, x_db)
```

Fig. 3. A driver for solving first (2) and then (4) using the *full Jacobian* approach.

```
!$taf subroutine spinup   input = 1,2,3,4
!$taf subroutine spinup  output = 5
!$taf subroutine spinup  active = 2,5
!$taf subroutine spinup  depend = 1,2,3,4
!$taf subroutine spinup  adname = spinup_ad
!$taf subroutine spinup ftlname = spinup_tl
```

Fig. 4. TAF flow directives for the subroutine `spinup`.

The two routines `spinup_tl` and `spinup_ad` are hand-written wrappers. They first compute $\frac{dx_e}{db}$ as shown in Fig. 3 and then carry out a matrix multiplication to propagate the derivative through the spin-up. The form of this matrix multiplication depends on the mode in which the entire code is differentiated. In forward mode, `spinup_tl` multiplies $\frac{dx_e}{db}$ from the right with the derivative of b with respect to the independents, and in reverse mode from the left with the derivative of the dependents with respect to x_e.

4 Numerical Example

As a test code, we employ "boxmod," a simple model of the atmospheric transport, which uses an Euler scheme to integrate the continuity equation for a passive trace gas. In this context, passive means that the concentration does not influence the atmospheric transport. The model is described in [456, 503], and its forward and reverse mode derivatives are described in [456]. There is one box for each hemisphere; their tracer concentrations take the role of x in (1). The inter-hemispheric mixing rate [456] of 1/year is its single parameter and corresponds to b in (1). We use boxmod in its methyl chloroform setup described in [456], with a uniform sink term corresponding to an inverse lifetime of 1/4.7 years [274]. We repeat the 1978 surface source estimates of Prinn et al. [439], modulated by a cosine with a period of one year and an amplitude of 10% the source strength. To mimic a large-scale application, the model is integrated with 10^7 time steps per year. We use the same initial concentration of 100 ppt (parts per trillion) for both boxes, which is about 10% off the equilibrium.

Table 1. Convergence of the spin-up.

Iteration	x(1)	x(2)	dx(1)/db
1	108.51645929	91.94163466	−7.3450784858
2	109.60820945	91.22018250	−8.1485930691
3	109.85705135	91.27066920	−8.2364935166
4	109.98888128	91.38080007	−8.2461093827
5	110.08786208	91.47740710	−8.2471613100
6	110.16704315	91.55632850	−8.2472763855
8	110.28261339	91.67186723	−8.2472903514
10	110.35811940	91.74737286	−8.2472905185
12	110.40745656	91.79671001	−8.2472905205
20	110.48351777	91.87277122	−8.2472905205
30	110.49845465	91.88770810	−8.2472905205
40	110.50023387	91.88948732	−8.2472905205
49	110.50043900	91.88969246	−8.2472905205
50	110.50044580	91.88969925	−8.2472905205
60	110.50047104	91.88972449	−8.2472905205
70	110.50047405	91.88972750	−8.2472905205
80	110.50047441	91.88972786	−8.2472905205
90	110.50047445	91.88972790	−8.2472905205
100	110.50047446	91.88972791	−8.2472905205

Table 1 shows the convergence of the spin-up in double precision. The first column lists the iteration number, and the next two columns both components of the state vector. The last column will be discussed later. For our *base* case we choose

Table 2. Performance for derivatives of boxmod equilibrium state with respect to mixing rate.

Case	Spin-up [s]	Std AD [s]	Jacobian [s]	Std/Jac
base	4.4	5.9	0.17	35.2
ifort	4.4	5.3	0.21	25.0
low accuracy	0.53	0.72	0.17	4.3

to iterate until the relative difference between y and x (as defined in (1)) is less than 10^{-7} for both components, which is reached after 49 iterations. We also look at a *low accuracy* case with a relative difference of 10^{-3}, which is reached after six iterations.

To compare the *standard* approach and the *full Jacobian* approach, two derivative codes are generated in TAF's pure forward mode (see Sect. 3). As our state vector has only two components, solving (4) for $\frac{dx_e}{db}$ is trivial. For our *base* case with 49 iterations, the relative difference in the spin-up sensitivities from both approaches is below 10^{-14}.

We also test solving (4) by iterating (3) (with the full Jacobian). The convergence of $\frac{dy}{db}$ is shown in the last column of Table 1. After only 8 iterations, the relative difference of the $\frac{dy}{db}$ value from *standard* is below 10^{-7}. Note that this procedure is different from *delayed derivative propagation*, as we are using the precomputed Jacobian for the equilibrium state. Also, *delayed derivative propagation* faces the decision when to start the derivative propagation without knowing how many iterations are still needed by the function code iteration to converge.

For the *low accuracy* case, the derivatives of both approaches (*standard* and *full Jacobian*) each have a relative difference below 10^{-4} to the 'true' sensitivity (from 100 iterations in the *standard* approach, see Table 1).

We run performance tests on a 3GHz Pentium 4 processor. Each test is repeated 10 times, and the average run time is recorded. Table 2 lists run times for three test cases. The cases *base* and *low accuracy* use the Lahey-Fujitsu Fortran 95 compiler lf95 with high optimisation level and double precision (flags **-O3 --dbl**). Case *ifort* equals the *base* case, with lf95 replaced by the Intel Fortran compiler, again with high optimisation level and double precision (flags **-O3 --autodouble**). The second column shows the run time for the spin-up integration; columns three and four refer to the *standard* and *full Jacobian* approaches, respectively. The last column shows the quotient of columns three and four. The relative run times depend strongly on the compiler and on compiler options and platform (not shown here). The *full Jacobian* approach is considerably faster than *standard*, even in the *low accuracy* case.

5 Performance analysis

In the box-model example, the *full Jacobian* strategy outscores the *standard* strategy considerably. Why is that? Let $r(m)$ denote the computational cost of a Jacobian product with m vectors for a single year run of boxmod. Our standard measure for computational cost of derivative code is CPU time in multiples of the CPU time

spent for the evaluation of the underlying function, but let's use the number of operations for a moment. Then $r(\cdot)$ is an affine function of m, i.e.

$$r(m) = r(1) + s \cdot (m-1) \,, \tag{5}$$

where $r(1)$ is the number of flops for the product of the Jacobian times the first vector and s being the number of flops per additional vector. The first vector is more expensive, because the computation of required values has to be included.

When returning to CPU time as performance measure, the form of $r(\cdot)$ depends on additional factors, most of which are platform and compiler dependent. Examples of such factors are data locality, vector length, level of compiler optimisation, and other compiler options. Also, from a certain m, the computation exceeds the available memory and hence needs to be split up. A previous study [293] has tested the performance for TAF-generated code for forward and reverse mode Jacobian evaluations within CCDAS: In reverse mode, testing $r(m)$ for fourteen values of m between 1 and 96 (see Figure 4 of [293]) indicates that (5) is indeed a good approximation, with $s \approx 0.25$ and $r(1)$ between 3 and 4. In forward mode, $r(1) = 1.5$ and $r(58) = 12$ yield $s \approx 0.2$.

Our present example is a bit more complex. In addition to increasing the number of Jacobian-vector products, we are also increasing the set of quantities with respect to which we are differentiating. On the other hand, the state, x, is active [57,207] even when differentiating only with respect to b (*standard*), i.e. derivatives of the state are propagated for both approaches *standard* and *full Jacobian*. We can estimate the extra cost for extending the Jacobian from derivatives with respect to b to derivatives with respect to b and x from the numbers in Table 2. With 10^7 time steps per year, we can safely neglect the CPU time spent outside boxmod and its derivatives. For the *base* case with its $k = 49$ iterations, we have $r(1) = (5.909/49)/(4.3651/49) \approx 1.35$, and $r(3) = 0.17/(4.3651/49) \approx 1.86$, which yields a slope s of about 0.25.

If we can neglect the cost of solving (4) and if (5) is a good approximation for capturing the cost of adding derivatives with respect to x, the *full Jacobian* strategy (left hand side) is preferable to *standard* (right hand side), if

$$k + r(1) + s \cdot (p + n - 1) \quad < \quad k \cdot (r(1) + s \cdot (p-1)) \,, \tag{6}$$

where p denotes the dimension of b. The left hand side k quantifies the cost of the spin-up itself. Since we have always $r(1) > 1$, the *full Jacobian* gets more favourable with increasing k, as seen in the example. Increasing n favours the *standard* approach, whereas increasing p favours the *full Jacobian* approach.

When increasing p, from a certain point the reverse mode is more efficient than the forward mode. This point depends on the number of dependent variables, q. If the dependent variables are the equilibrium state then $q = n$. The dependent variables may also be some function of the equilibrium state, including, e.g., a transient integration as illustrated in Fig. 1. In reverse mode, the *Christianson* approach is more efficient than the *standard* approach, as the former needs to provide fewer required values. If we denote the cost of evaluating the Jacobian times q vectors in reverse mode by $\tilde{r}(q)$ and its slope by $\tilde{s}$, the corresponding cost estimates for the *full Jacobian* and *Christianson* approaches are respectively

$$k + \tilde{r}(1) + \tilde{s} \cdot (n-1) \,, \text{ and} \tag{7}$$

$$k \cdot (\tilde{r}(1) + \tilde{s} \cdot (q-1)) \,. \tag{8}$$

We can illustrate (6) with numbers from [210], where the authors apply TAF to a Navier-Stokes solver, in a simple configuration with $k = 500$, $n = 5 \times 801$, $p = 2$ (Mach number and angle of attack), and $q = 1$ (scalar objective function of lift and drag). With their $r(1) = 2.4$ and an assumed $s = 0.25$, (6) yields a cost of about 1500 for the *full Jacobian* approach versus about 1300 for the *standard* approach. As the *Christianson* approach yields $\tilde{r}(1) = 3.4$, the *standard* approach (in forward mode) may be preferable up to $p = 5$ (using the assumed $s = 0.25$). For any larger p, (8) yields a cost of 1700. Hence, at first glance, the *full Jacobian* appears slightly favourable for $5 < p < 800$. However, for $n = 5 \times 801$, we cannot ignore the cost of solving (4). The cost of standard methods for solving systems of linear equations (e.g. LU decomposition) grows with n^3 [438]. Solving (4) iteratively (as done in Sect. 4), requires one matrix-vector product in R^n per iteration. The only relevant component of the spin-up sensitivity $\frac{dx_e}{db}$ is given by the derivative of the objective function with respect to the equilibrium state. Thus, one can focus on convergence of that component.

The comparison looks much different for the spin-up of terrestrial biosphere models. There is no exchange of information across borders of grid cells, so the Jacobian $\frac{\partial f}{\partial x}$ has a block diagonal structure, with the block size equal to the dimension of the state space per grid cell, n_{eff}, which is the effective size of state space to be used in (6), as the sparse Jacobian can be retrieved from n_{eff} Jacobian-vector products. Also, (4) can be solved block by block. As n_{eff} is usually below 10, the computation of $\frac{\partial f}{\partial x}$ and solving (4) are inexpensive. Regarding s, one can, for instance, assume the above mentioned value for CCDAS of $s \approx 0.2$. With $n = 5$ and $k = 3000$, the cost of the *full Jacobian* approach is dominated by the cost of the spin-up itself, which is 3000. The sensitivities come at an extra cost of only about 2.5.

6 Summary and Outlook

We have explored the *full Jacobian* approach to the computation of sensitivities for the spin-up phase. For a simple box-model of the atmospheric transport, the *full Jacobian* approach is 4 to 35 times more efficient than the *standard* approach of propagating derivatives through the entire spin-up phase. This benefit increases with the number of iterations and decreases with the size of the state space.

We have shown that for terrestrial biosphere models, which are characterised by a low dimensional effective state-space, the *full Jacobian* approach looks promising. As a first large-scale application, we plan to compute spin-up sensitivities for a biosphere model within CCDAS.

Acknowledgements

The idea for this study originates from a discussion with Srikanth Akkaram on sensitivities of a structural mechanics code. Wolfgang Knorr and Marko Scholze have provided valuable advice and comments.

Streamlined Circuit Device Model Development with fREEDA®ãnd ADOL-C*

Frank P. Hart[1], Nikhil Kriplani[1], Sonali R. Luniya[1], Carlos E. Christoffersen[2], and Michael B. Steer[1]

[1] Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, USA
fphart@eos.ncsu.edu,nmkripla@unity.ncsu.edu,srluniya@unity.ncsu.edu, m.b.steer@ieee.org

[2] Department of Electrical Engineering, Lakehead University, Thunder Bay, Ontario, Canada
c.christoffersen@ieee.org.

Summary. Time-marching simulation of electronic circuits using the U.C. Berkeley program Spice and variants has been a standard practice for electronics engineers since the mid-1970s. Unfortunately, the development cycle of Spice models may be lengthy because device model equations and their derivatives must be coded manually. Also, many files in the source tree must be modified to define a new model. *f*REEDA®, www.freeda.org, an object-oriented circuit simulator under development at several universities, overcomes many limitations of the conventional electronic model development paradigm. A key to this implementation is the ADOL-C package, which is used to automatically evaluate the derivatives of the device model equations. Resulting models are more compact, and the development time is shorter. The development history of selected Spice models and their *f*REEDA® counterparts are presented to illustrate the advantages of this approach.

Key words: Circuit simulation, semiconductor device modelling, electronic device modelling, *f*REEDA®, ADOL-C

1 Introduction

Computer-aided simulation of electronic circuits has been common since the mid-1970s when the U.C. Berkeley program Spice [389] was made available to an electronics industry that was increasingly engaged in the development and manufacture of

* This material is based upon work supported in part by the Space and Naval Warfare Systems Center San Diego under grant number N66001-01-1-8921 as part of the DARPA NeoCAD Program and in part by the U.S. Army Research Laboratory and the U.S. Army Research Office on Multifunctional Adaptive Radio Radar and Sensors (MARRS) under grant number DAAD19-01-1-0496.

integrated circuits based on semiconductor devices. In prior decades, most electronics systems were based on collections of discrete semiconductor devices or vacuum tubes connected by discrete wires or by printed circuit board wiring. These systems were amenable to relatively low-cost prototype production and laboratory observation of every interconnection point using measuring equipment familiar to electrical engineers. However, by the mid 1970s, integrated circuits with hundreds or thousands of interconnection points – most not observable in a laboratory setting – had rendered the existing prototyping paradigm obsolete. Since then, pre-production validation of integrated circuit designs has relied heavily upon Spice simulation. Spice was not the first simulator produced at U.C. Berkeley, but it benefited from research done on earlier generations of simulator engines during the 1960s. Over the years, Spice has been ported successfully to generations of less expensive computers. Today it is both economical and common to simulate even discrete circuits implemented on printed circuit boards prior to building prototype models.

Time-marching simulation using state variable-based models was also initially developed in the 1960s [437]. Such simulators formed and solved systems of differential equations. However, these programs fell out of favor because Spice's modelling philosophy led to purely algebraic systems of equations that were inherently more sparse than the state variable approaches. Interest in this approach was renewed when researchers in the discipline of microwave engineering became interested in combining a device's interactions with electromagnetic fields [415]. Spice analyses are limited to voltage and currents only, and so a new simulator environment based on state variable analysis was created to permit this form of analysis. This initial effort has evolved into *f*REEDA® [120]. Presently, *f*REEDA® is the only *netlist-driven* circuit simulator available to the public which uses Automatic Differentiation (AD). One prior effort including AD is disclosed in [364], but this simulator was not netlist-driven. One other effort [308] reported significant results in modelling Metal Oxide Semiconductor transistors with several (AD) tools, but the simulator environment was not made publicly available.

2 Background

2.1 Constitutive Equations and Network Equations

Computer-aided circuit analysis in its most basic form is comprised of *constitutive* and *network* equations [123,416]. Constitutive equations usually express voltage as a function of current (in units of Amperes) or vice versa for a *particular* element. Figure 1(a) shows a generic two-terminal element where the voltage across the element is defined with respect to positive and negative terminals, and current is defined positively as flowing from the positive to the negative terminals. The current has a vector quality in the sense that the arrow in Fig. 1(a) may be reversed and the magnitude negated. Figure 1(b) shows a linear resistor, a simple impedance element described by Ohm's law, $v(t) = Ri(t)$ [250] or the admittance form, $i(t) = Gv(t)$, where $G = 1/R$. *Network* equations govern the interconnection of elements, and there are two forms based on Kirchoff's Current (KCL) and Voltage (KVL) Laws. Owing to the simplicity of matrix formulation [144, 523] for circuit simulation, the KCL equation form is preferred. The KCL equation form of the matrix is called a Modified Nodal Admittance Matrix (MNAM) [264] because most entries in the

matrix have the physical dimension of an admittance (a ratio of current to voltage), but provisions are made through a form of domain decomposition to permit the entry of circuit elements with different physical units. Figure 1(c) shows a network of three generic two-terminal devices. One node (or terminal) in the circuit is always designated as the reference node, so that the MNAM formulation is non-singular.

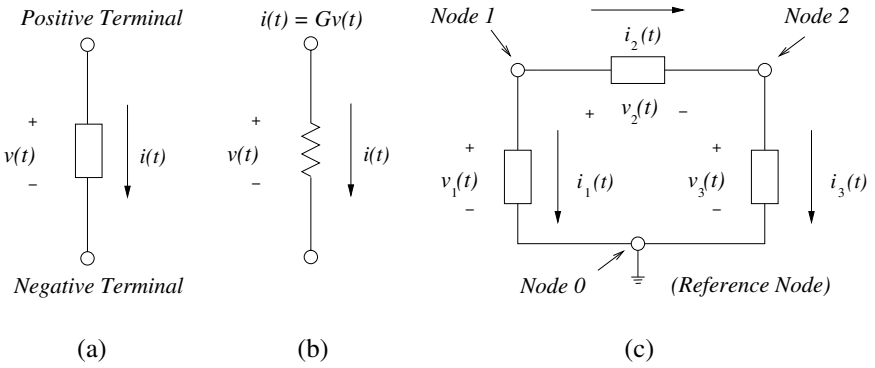


Fig. 1. (**a**) Element constitutive equations; (**b**) Resistor; (**c**) Network equations.

2.2 Forms of Circuit Simulation

The two best-known and longest established forms of circuit simulation are known as *transient* analysis and *AC* (i.e., "alternating current") analysis. Both forms were available in the first release of Spice. Transient analysis is time-domain simulation of the circuit using quadrature integration. Linear and nonlinear differential circuit equations are discretized using either the backward Euler or trapezoidal interpolating functions, and the integration from one time step to the next is governed by an iterative procedure using either Newton's method or the minimization of some error function. AC analysis is frequency-domain simulation of the circuit and is supported only for linear circuit elements in *f*REEDA®, so it will not be discussed further; the emphasis here is on transient analysis. Underlying both analysis forms is a form known as *DC* (i.e., "direct current") analysis used to establish the initial operating conditions for both transient and AC analysis.

One other popular form of circuit simulation that is not present in Berkeley's Spice but has been implemented in *f*REEDA® [121] and other simulators [322, 464] is called *Harmonic Balance* (HB) analysis. HB is an implementation of Galerkin's methods [446] for finding the steady-state response of a nonlinear network. HB will not be discussed in depth, but its method for formulating the MNAM – which differs drastically from that of Spice – has been applied to *f*REEDA® transient analysis [119], and this will be discussed in some detail in Sect. 3.2.

2.3 Constitutive Equations for Elements of Interest

Most transient simulation models are composed of combinations of simpler element models. Those models most useful to the present discussion will be briefly reviewed.

Resistors were mentioned in Sect. 2.1. Inductors and capacitors are linear dynamic elements and are described by differential equations [144]. Resistors, capacitors, and inductors often appear within semiconductors in nonlinear forms and are described by Taylor series expansions in these cases.

Nonlinear elements such as semiconductor diodes and transistors are also described by a set of constitutive equations [492]. The diode may be the simplest nonlinear element. Figure 2(a) shows the diode, a device whose current is an exponential function of voltage. A simplified equation for a discrete diode is $i(t) = I_{\mathrm{s}}(\exp(v(t)/V_{\mathrm{T}}) - 1)$, where I_{s} is known as the reverse saturation current and V_{T}, the thermal voltage ($V_{\mathrm{T}} \approx 26\,\mathrm{mV}$ at 300K), is a threshold voltage beyond which the exponential behavior becomes apparent. Figure 2(b) shows the behavior of a typical 1N4153 diode [368]. Figure 3(a) shows an abstract element form called

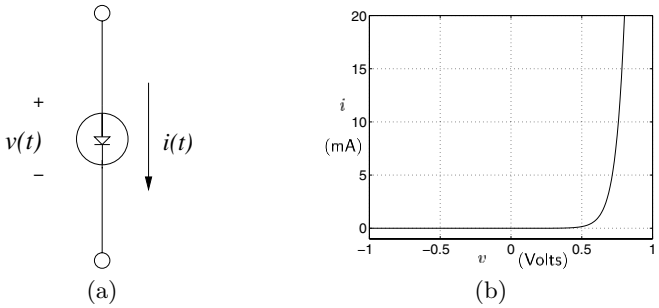


Fig. 2. (**a**) Diode element; (**b**) Current as a function of voltage for a 1N4153 diode.

a 2-port [144], which allows for the arbitrary definition of transfer function relationships between two pairs of conductors. For example, $i_1(t) = f(v_1(t), v_2(t), i_2(t))$ or $i_2(t) = f(v_2(t), v_1(t), i_1(t))$. A generalization to an N-port allows for the arbitrary definition of transfer functions from each of N ports to $N-1$ other ports. Port transfer functions are usually provided in matrix form. The constitutive equations for multi-terminal semiconductor devices may be viewed as specific instances of a multi-port.

Finally, circuit simulation often uses two other abstract forms known as ideal voltage and current sources. These are shown in Fig. 3(b–c). They are "ideal" in the sense that their voltages and currents are not related by intrinsic device behavior, but instead are functions of the circuit connections. Specifically, an ideal voltage source has zero internal impedance, so its current is a function of the rest of the circuit. Also, an ideal current source has an infinite internal impedance, so its voltage is a function of the rest of the circuit. These ideal sources are important in equivalent circuit modelling of real devices.

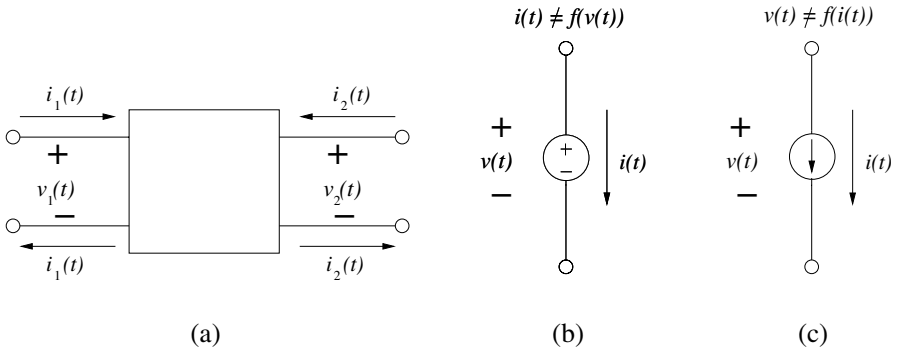


Fig. 3. (**a**) Abstract 2-port element; (**b**) Ideal voltage source; (**c**) Ideal current source.

3 Transient Circuit Simulation Device Modelling

3.1 Spice and Companion Modelling

A 1984 publication reviewing the history of circuit simulation [430] credits Rohrer and a group of graduate students with finding that nonlinear circuit elements could be modelled in the time domain by permitting an equivalent linear circuit of resistors and ideal current sources to have their model values updated not just at time steps within the simulation, but also at different iterates of a Newton iterative loop at a given time step. (Partial disclosure of this technique was given in [362].) This equivalent circuit model was first described as the "Associated Discrete Model" in the literature [123], but more recently it has been termed simply a "companion model." Companion modelling became the standard method for implementing a nonlinear device model in Spice, and it remains unchanged to this day.

Companion modelling usually features time discretization of the element's equations and incremental linearization of the nonlinear models to permit Newton iteration. The effect of *only* time discretization only can be seen in the companion models for linear capacitors and inductors [123, 434]. Here the companion model for a nonlinear device – a semiconductor diode – will be developed. In the equations that follow, time will be discretized assuming the backward Euler rule and j will indicate the iterate. The companion model begins with the constitutive equation:

$$f(v) \equiv i(t) = I_{\mathrm{s}} \left[\exp\left(\frac{v(t)}{V_{\mathrm{T}}} \right) - 1 \right] . \tag{1}$$

Next, to facilitate Newton iteration, the partial derivative of (1) with respect to v must be obtained:

$$\frac{\partial f}{\partial v} = \frac{I_{\mathrm{s}}}{V_{\mathrm{T}}} \exp\left(\frac{v(t)}{V_{\mathrm{T}}} \right) . \tag{2}$$

Substitute (1) and (2) into (3) defining the Newton iteration, yielding (4):

$$f(v^{j+1}) = f(v^j + \triangle v) \approx f(v^j) + \frac{\partial f(v^j)}{\partial v^j} \triangle v \Rightarrow \tag{3}$$

$$i^{j+1} = I_{\mathrm{s}} \left[\exp\left(\frac{v^j}{V_{\mathrm{T}}} \right) - 1 \right] + \frac{I_{\mathrm{s}}}{V_{\mathrm{T}}} \exp\left(\frac{v^j}{V_{\mathrm{T}}} \right) (v^{j+1} - v^j) . \tag{4}$$

From the form of (4), it can be seen that the result is an equivalent circuit consisting of an ideal current source (with value set during the previous iteration) and a resistor. These terms are identified in (5) and (6). Time is not explicitly discretized in this model, but time discretization is implied in the iterated voltage values, which are functions of time.

$$I_{\mathrm{eq}}^{j} = I_{\mathrm{s}}\left[\exp\left(\frac{v^{j}}{V_{\mathrm{T}}}\right) - 1\right] - \frac{I_{\mathrm{s}}v^{j}}{V_{\mathrm{T}}}\exp\left(\frac{v^{j}}{V_{\mathrm{T}}}\right) \tag{5}$$

$$g_{\mathrm{eq}}^{j} = \frac{I_{\mathrm{s}}}{V_{\mathrm{T}}}\exp\left(\frac{v^{j}}{V_{\mathrm{T}}}\right) . \tag{6}$$

Consider now a simple circuit consisting of an ideal current source, a resistor, and a diode as shown in Fig. 4(a). In this circuit, there is only one node other than the reference node, and thus the Newton iteration is on only one equation. It is still illustrative, but circuits with more than one node require Newton iteration on vectors, requiring Jacobian matrices. For this simple circuit, the analysis task is to determine the currents through the resistor and diode at all times. Note that the resistor current is dependent on the diode voltage, so the solution depends wholly upon the diode. The Spice transient analysis method substitutes the companion model for the diode, transforming the circuit into the collection of resistors and current sources shown in Fig. 4(b). Equations (7)–(8) show the results of the companion model substitution. Applying KCL at the top node (or terminal) of Fig. 4,

$$I_{\mathrm{src}} = Gv^{j+1} + i^{j+1} \Rightarrow i^{j+1} = I_{\mathrm{src}} - Gv^{j+1} = I_{\mathrm{eq}}^{j} + g_{\mathrm{eq}}^{j}v^{j+1} \tag{7}$$

$$\Rightarrow v^{j+1} = \frac{I_{\mathrm{src}} - I_{\mathrm{eq}}^{j}}{G + g_{\mathrm{eq}}^{j}} . \tag{8}$$

The Spice transient analysis routine performs Newton iteration at every time step, updating the MNAM with new g_{eq}^{j} values and the vector of sources with new I_{eq}^{j} values at every iteration until convergence. At each iterate, the time step is reduced so that the values of v^{j+1} approach v^{j}.

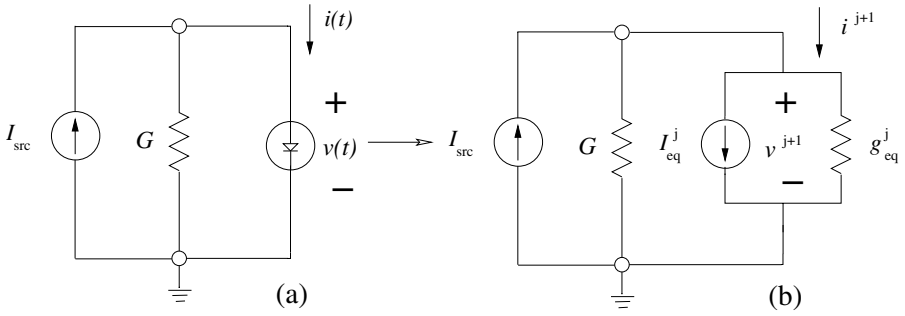


Fig. 4. (**a**) Simple circuit containing a diode; (**b**) Equivalent circuit containing companion model.

The advantage of companion modelling is now apparent. Through companion model substitution, nonlinear behavior is reduced to a simple linear form. In addition, Spice transient analysis is reduced to a simple extension of Spice DC analysis

with Newton iteration. However, the algebraic model form must be obtained by mathematical analysis and inserted into the model code, and this form is not always easily obtained. Moreover, an unfortunate consequence of updating model values as a simulation progresses is that the MNAM contents change at every time step and at every Newton iterate within a time step, and so the computationally expensive LU factorization performed on the MNAM gets no reuse. Philosophically, the companion model is a mathematical abstraction that risks departing from the underlying physics of the device to facilitate a simulation method consisting of equivalent circuit models with very few elements.

3.2 *f*REEDA® and State Variable Based Modelling Using ADOL-C

In *f*REEDA®, device element models are faithful replicas of the physical equations describing the device. Consider the currents and voltages at the ports of a nonlinear device to be expressed as functions of independent parameters called state variables ($\mathbf{x}()$), i.e.,

$$\mathbf{v}_{\mathrm{p}}(t) = \mathbf{v}\left[\mathbf{x}(t), \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}, \cdots, \frac{\mathrm{d}^m\mathbf{x}}{\mathrm{d}t^m}, \mathbf{x}_{\mathrm{D}}(t)\right] \tag{9}$$

$$\mathbf{i}_{\mathrm{p}}(t) = \mathbf{i}\left[\mathbf{x}(t), \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}, \cdots, \frac{\mathrm{d}^m\mathbf{x}}{\mathrm{d}t^m}, \mathbf{x}_{\mathrm{D}}(t)\right] , \tag{10}$$

where $\mathbf{x}_{\mathrm{D}}(t)$ is a time-delayed version of $\mathbf{x}(t)$. To avoid charge conservation problems in transient analysis [122], (9)–(10) must be reformulated in stages as follows:

$$\text{stage } 1 : \begin{cases} \mathbf{f}_1(\mathbf{x}(t), \mathbf{x}_{\mathrm{D}}(t)) \\ \mathbf{g}_1(\mathbf{x}(t), \mathbf{x}_{\mathrm{D}}(t)) \end{cases} \tag{11}$$

$$\text{stage } 2 : \begin{cases} \mathbf{f}_2(\mathbf{f}_1(t), \mathrm{d}\mathbf{g}_1/\mathrm{d}t) \\ \mathbf{g}_2(\mathbf{f}_1(t), \mathrm{d}\mathbf{g}_1/\mathrm{d}t) \end{cases} \tag{12}$$

$$\vdots$$

$$\text{stage } n-1 : \begin{cases} \mathbf{f}_{n-1}(\mathbf{f}_{n-2}(t), \mathrm{d}\mathbf{g}_{n-2}/\mathrm{d}t) \\ \mathbf{g}_{n-1}(\mathbf{f}_{n-2}(t), \mathrm{d}\mathbf{g}_{n-2}/\mathrm{d}t) \end{cases} \tag{13}$$

$$\text{stage } n : \begin{cases} \mathbf{v}(\mathbf{f}_{n-1}(t), \mathrm{d}\mathbf{g}_{n-1}/\mathrm{d}t) \\ \mathbf{i}(\mathbf{f}_{n-1}(t), \mathrm{d}\mathbf{g}_{n-1}/\mathrm{d}t) \end{cases} . \tag{14}$$

An example and advantages of this formulation will be given in Sect. 4.3. All arguments in (11)–(14) become ADOL-C active variables [228] so that the derivatives of functions $\mathbf{f}_1$, $\mathbf{g}_1$, $\mathbf{f}_2$, etc. can be obtained automatically and used in the model code. Derivatives are calculated in the forward mode in *f*REEDA®. Comparison of forward and reverse modes is a matter for future research. Through the use of object-oriented programming techniques, all device models are derived C++ classes that inherit the characteristics of a C++ base class [120]. For nonlinear devices in *f*REEDA®, "AdolcElement" is the base class, and other nonlinear devices are derived classes as illustrated in the Unified Modelling Language (UML) diagram, Fig. 5(a). Model developers describe the number of terminals and state variables required for the element in the *init()* function of the derived class and implement the nonlinear equations unique to the derived class in the *eval()* function for the

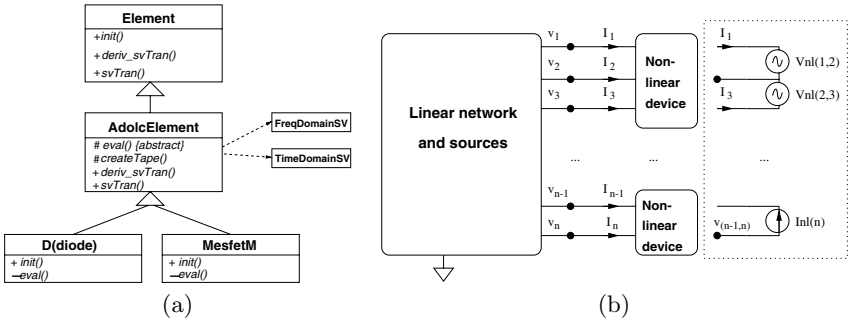


Fig. 5. (**a**) UML diagram; (**b**) Transient simulation circuit partitioning.

derived class. If more than one stage is necessary, they are implemented in functions called *eval2()* and *eval3()*. The AdolcElement class contains the interface to ADOL-C for initializing and manipulating the ADOL-C 'tapes' of active variables. AdolcElement also encapsulates and hides details of working with ADOL-C. Calls to the *eval()* routine for the derived class are bracketed between ADOL-C *trace* statements in the AdolcElement code. The calculation of total derivatives and time delayed variables is performed by the TimeDomainSV or the FreqDomainSV classes, depending on the type of analysis being performed. Thus, the same code can be used in any circuit analysis. This is possible because time is discretized in the *f*REEDA® transient analysis code and not in the device model code. ADOL-C is used to calculate the derivatives of the functions at each stage and thus obtain the Jacobian of the currents and voltages with respect to the state variables (as shown in [120]). The procedure to obtain the Jacobian is embedded in AdolcElement so that model developers need not code that in their derived classes. Absent ADOL-C, a state-variable approach would require manual coding of the Jacobian matrices. Thus the importance of the ADOL-C package to facilitating this state variable approach in *f*REEDA® cannot be understated.

The results of evaluating (9)–(10) for each port are collected by the *f*REEDA® state variable transient analysis routine into a vector of equations describing the nonlinear portion of the circuit:

$$\mathbf{v}_{\mathrm{NL}}(t) = \mathbf{v}\left[\mathbf{x}(t), \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}, \cdots, \frac{\mathrm{d}^m\mathbf{x}}{\mathrm{d}t^m}, \mathbf{x}_{\mathrm{D}}(t)\right] \tag{15}$$

$$\mathbf{i}_{\mathrm{NL}}(t) = \mathbf{i}\left[\mathbf{x}(t), \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t}, \cdots, \frac{\mathrm{d}^m\mathbf{x}}{\mathrm{d}t^m}, \mathbf{x}_{\mathrm{D}}(t)\right] . \tag{16}$$

The formulation of the system of equations for *f*REEDA®'s transient analysis may now be described. Figure 5(b) shows the partitioning of the complete circuit into linear and nonlinear portions, with the port abstractions for each nonlinear device indicated. At the boundary between the linear and nonlinear portions, the voltages must be equal. Thus two voltage vectors, one a function of the linear circuit behavior and the other a function of the nonlinear circuit behavior, must be equal and can be used to form an error function. Let the linear portion of the circuit in Fig. 5(b) be described by two MNAMs, $\mathbf{G}$ and $\mathbf{C}$, where $\mathbf{G}$ describes the linear static elements (such as resistors), and $\mathbf{C}$ describes the linear dynamic elements (such as

linear capacitors and inductors). Also, let $\mathbf{u}$ be the vector of unknown voltages and currents and $\mathbf{s}$ be a vector of sources. Then,

$$\mathbf{G}\mathbf{u}(t) + \mathbf{C}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathbf{s}(t) \tag{17}$$

$$\mathbf{s}(t) = \mathbf{s}_{\mathrm{f}}(t) + \mathbf{s}_{\mathrm{v}}(t)\ . \tag{18}$$

In (18), the source vector $\mathbf{s}$ is comprised of $\mathbf{s}_{\mathrm{f}}$, a vector of independent *forcing* sources, and $\mathbf{s}_{\mathrm{v}}$, a vector of currents injected by the nonlinear circuit into the linear circuit. Now, through the use of an incidence matrix[3], $\mathbf{T}$, which specifies connectivity information relating the circuit's node assignments to its state variables, the following relationships hold:

$$\mathbf{v}_{\mathrm{L}}(t) = \mathbf{T}\mathbf{u}(t) \tag{19}$$

$$\mathbf{s}_{\mathrm{v}}(t) = \mathbf{T}^{\mathrm{T}}\mathbf{i}_{\mathrm{NL}}(t)\ . \tag{20}$$

In (19), $\mathbf{v}_{\mathrm{L}}(t)$ is the vector of port voltages from the linear elements at the linear/nonlinear interface boundary. Substituting (20) into (17)–(18) and simplifying yields the equation for the state of the system:

$$\mathbf{G}\mathbf{u}(t) + \mathbf{C}\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}t} = \mathbf{s}_{\mathrm{f}}(t) + \mathbf{T}^{\mathrm{T}}\mathbf{i}_{\mathrm{NL}}(t)\ . \tag{21}$$

Equation (21) is subject to the equation for the error function:

$$\mathbf{f}(t) = \mathbf{v}_{\mathrm{L}}(t) - \mathbf{v}_{\mathrm{NL}}(t) = \mathbf{0} \tag{22}$$

$$\mathbf{f}(t) = \mathbf{T}\mathbf{u}(t) + \mathbf{v}_{\mathrm{NL}}(t) = \mathbf{0}\ . \tag{23}$$

It was noted earlier in this section that time discretization does not occur within the element class definition in *f*REEDA®; instead it occurs within the analysis routine. This has the dual advantages of simplifying the coding of the element classes and also allowing for different time interpolating functions. *f*REEDA® allows a choice of either Backward Euler or Trapezoidal interpolating functions. After discretizing the vector of unknowns $\mathbf{u}(t)$, its derivative $\mathrm{d}\mathbf{u}(t)/\mathrm{d}t$ is defined as

$$\frac{\mathrm{d}\mathbf{u}(t)}{\mathrm{d}t} \Rightarrow \mathbf{u}'_n = a\mathbf{u}_n + \mathbf{b}_{n-1}\ , \tag{24}$$

where the scalar a and vector $\mathbf{b}$ depend upon the choice of interpolating function. Substituting the discretized $\mathbf{u}$ into (21),

$$\mathbf{G}\mathbf{u}_n + \mathbf{C}[a\mathbf{u}_n + \mathbf{b}_{n-1}] = \mathbf{s}_{\mathrm{f},n} + \mathbf{T}^{\mathrm{T}}\mathbf{i}_{\mathrm{NL}}(\mathbf{x}_n)\ , \tag{25}$$

and solving for $\mathbf{u}_n$,

$$\mathbf{u}_n = [\mathbf{G} + a\mathbf{C}]^{-1}[\mathbf{s}_{\mathrm{f},n} - \mathbf{C}\mathbf{b}_{n-1} + \mathbf{T}^{\mathrm{T}}\mathbf{i}_{\mathrm{NL}}(\mathbf{x}_n)]\ . \tag{26}$$

Discretizing the error function defined in (22-23),

[3] The number of rows of $\mathbf{T}$ is equal to the total number of nonlinear ports and the number of columns is equal to the number of nodes. For each row, a $+1$ entry denotes the $+$ terminal of a port, and a -1 denotes the $-$ terminal. All other entries are 0.

$$\mathbf{f}(\mathbf{x}_n) = \mathbf{T}\mathbf{u}_n - \mathbf{v}_{\mathrm{NL}}(\mathbf{x}_n) = \mathbf{0}\ . \tag{27}$$

Now, define the following quantities comprised of $\mathbf{T}\mathbf{u}_n$:

$$\mathbf{s}_{\mathrm{sv},n} = \mathbf{T}[\mathbf{G} + a\mathbf{C}]^{-1}[\mathbf{s}_{\mathrm{f},n} - \mathbf{C}\mathbf{b}_{n-1}] \tag{28}$$

$$\mathbf{M}_{\mathrm{sv}} = \mathbf{T}[\mathbf{G} + a\mathbf{C}]^{-1}\mathbf{T}^{\mathrm{T}}\ . \tag{29}$$

Substituting (28)–(29) into (27) leads to

$$\mathbf{f}(\mathbf{x}_n) = \mathbf{s}_{\mathrm{sv},n} + \mathbf{M}_{\mathrm{sv}}\mathbf{i}_{\mathrm{NL}} - \mathbf{v}_{\mathrm{NL}}(\mathbf{x}_n) = \mathbf{0}\ . \tag{30}$$

For a fixed step size – usually the case – $\mathbf{M}_{\mathrm{sv}}$ is a constant, and the matrix $[\mathbf{G}+a\mathbf{C}]^{-1}$ appearing in (28)–(s29) is LU-factored only once per simulation.

4 Selected Modelling Examples

4.1 Illustrative Comparison of Diode Models

The development of models for the semiconductor diode provides an illustrative comparison. A full description of aspects of the Spice diode model development is given in [434]. Details of the *f*REEDA® diode model are described in [118]. The Spice companion model for a simplified diode model was described in Sect. 3.1, and it was noted there that the process for developing companion models requires model developers to perform derivatives on the constitutive equations and manually code the derivative equations to facilitate Newton iteration. The *f*REEDA® code for the *init()* and *eval()* functions for the simplified two parameter diode derived class is shown in Listing 1. In this case, the *eval()* function is literally coding the constitutive equations. These two functions take up only 17 lines of code. The complete C++ code and header files for the simplified diode are both only 56 lines long.

```
void DiodeJcn::init() throw(string&) {
  // Set the number of terminals
  setNumTerms(2);
  // Set number of states
  setNumberOfStates(1);
  // create tape
  IntVector var(1,0); // create vector of 1 element set to 0
  createTape(var);    // creates state var x[0]
}

void DiodeJcn::eval(adoublev& x, adoublev& vp, adoublev& ip) {
  vp[0] = x[0];    // x[0] == input voltage
  ip[0] = is * ( exp(x[0]/vT) - 1 );
}
```

Listing 1. Critical model code for simplified diode.

4.2 The Berkeley Short-channel IGFET Model Version 4 (BSIM4)

The Device Group at the University of California at Berkeley has been at the forefront of specifying semiconductor physics models for field effect transistors in the most advanced semiconductor process technologies. Figure 6(a) shows the model for a contemporary Metal Oxide Semiconductor Field Effect Transistor (MOSFET) device. The first Field Effect Transistor (FET) models were published in 1968 and had only 41 parameters to fully describe any transistor. In the 1980s, as semiconductor process technologies continued to shrink the minimum size of their features, other physical phenomena were observed which necessitated the addition of more parameters to the models, and the BSIM models were created.

In 2000, a fourth major version of the BSIM models called BSIM4 was released. Figure 6(a) shows the model for a contemporary short-channel MOSFET device consisting of four terminals (gate, drain, source, and bulk). The BSIM4 device semiconductor physics model has over 200 parameters, and the current high level of interest in this model makes it a good choice for a case study comparison of a Spice model with a *f*REEDA® model. Such a study was completed by one of the authors, who implemented the BSIM4 model in *f*REEDA® as a Master's Thesis [315] in 2002. Starting with the BSIM4 semiconductor physical model documentation, the model consisted of about 1500 lines of C++ code in two files and on 25 printed pages, and required 7 months to develop. Much of this time was spent implementing code structures to support particular features of the BSIM models for the first time.

Due to industry involvement with UC-Berkeley in the development of the BSIM4 model, it is not possible to make precise development cycle comparisons with *f*REEDA®, but from archival materials at UC-Berkeley, it can be observed [147] that 21 months passed from the final BSIM3 release until the first BSIM4 release. It is also known that the contemporary BSIM4 models consist of about 20,000 lines of C code spanning 21 source code files [333]. It should be noted that Berkeley's MOSFET models are not the only ones available, and implementing other models has proven to be less labor-intensive. For example, ETH-Switzerland's EPFL-EKV model [76], which has only 44 parameters, was implemented by one student [283] as a semester project for a circuit simulation class at NC State University. At the time, the student had limited knowledge of *f*REEDA®'s internals.

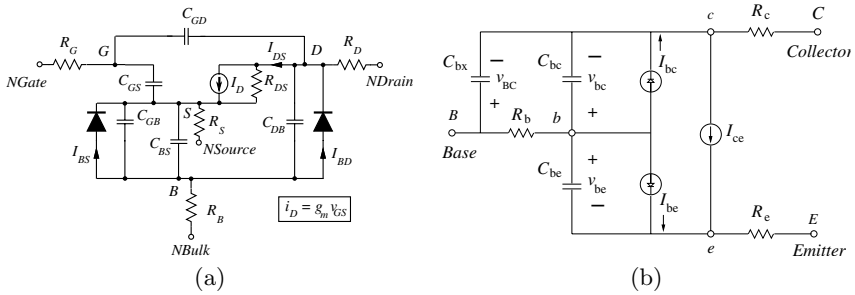


Fig. 6. Equivalent circuit model schematic diagrams for the (**a**) MOSFET; (**b**) Bipolar Junction Transistor (BJT).

4.3 fREEDA® Universal Modelling Approach

A universal state variable-based modelling approach was proposed in [122]. The central notion of this approach is to choose as state variables the quantities most appropriate to accurately model the physical device as illustrated in Sect. (3.2), and then derive any additional required variables through a set of hierarchically evaluated functions. To illustrate the approach, consider the simplified NPN-type Bipolar Junction (BJT) model of Fig. 6(b). By applying the universal modelling approach, it is possible to create a device model for the BJT that conserves charge [122]. Charge conservation has been a problem with some Spice MOSFET models [124, 565], thus rendering the models suspect to semiconductor physicists. Referring to Fig. 6(b), the voltages across the base collector capacitor $v_{\rm bc}$ and base emitter capacitor $v_{\rm be}$ are chosen as state variables (or independent variables). A charge-conserving model of the bipolar transistor is then described using three ADOL-C tapes. Each tape has two input and two output variables. In tape 1 the quiescent current components $I_{\rm bc}$ and $I_{\rm be}$ are computed as

$$I_{\rm be} = \frac{I_{\rm bf}}{\beta_{\rm F}} + I_{\rm le} \tag{31}$$

$$I_{\rm bc} = \frac{I_{\rm br}}{\beta_{\rm R}} + I_{\rm lc} \ , \tag{32}$$

where $\beta_{\rm F}$ is a current gain parameter, $I_{\rm bf}/\beta_{\rm F}$ and $I_{\rm le}$ are the components of the currents through the base-emitter diode, and $I_{\rm br}/\beta_{\rm F}$ and $I_{\rm lc}$ are components of the the currents through the base-collector diode. The first output vector from tape 1 stores the charge across the base collector and base emitter capacitors which can be evaluated as

$$q_{\rm bc}(v) = \int_0^v c_{\rm bc}(v_{\rm bc}){\rm d}v_{\rm bc} \tag{33}$$

$$q_{\rm be}(v) = \int_0^v c_{\rm be}(v_{\rm be}){\rm d}v_{\rm be} \ , \tag{34}$$

where the integrals are evaluated analytically. The second output vector stores the diode current components and junction voltages. The derivatives of the charge across the capacitors,

$$I_{\rm Cbc} = \frac{{\rm d}q_{\rm bc}}{{\rm d}t} \tag{35}$$

$$I_{\rm Cbe} = \frac{{\rm d}q_{\rm be}}{{\rm d}t} \ , \tag{36}$$

are input parameters in tape 2. These derivatives are obtained with a formula that depends of the type of quadrature being used. The approximation of the time derivatives is used to calculate the charge across the distributed base collector capacitor $C_{\rm bx}$ in a manner similar to that done in (33-34). Inputs to tape 3 contain the corresponding charge in $C_{\rm bx}$ and the junction voltages. In tape 3 the current across $C_{\rm bx}$ is computed in a manner analogous to (35-36), and the final external voltages and currents are calculated using the intermediate variables generated at the previous tapes.

The ADOL-C tapes are generated once and then used in the main program every time the bipolar transistor model equations or its derivatives with respect

to its input parameters are needed. The procedure to calculate the derivatives of the model equations from the set of tapes is the same for all models [122] and is handled by a base class common to all elements. Therefore, the addition of the bipolar transistor model in *f*REEDA® is accomplished by adding a new derived class (one ".cc" file and a header file) with the definition of the three ADOL-C tapes plus other information such as the number of terminals and model parameter names. A hierarchical state-variable approach similar to that shown here can be applied to assure charge conservation in other nonlinear devices, and it is advocated for all nonlinear devices.

5 Conclusion

Through a deft combination of the use of ADOL-C's automatic differentiation capabilities, object-oriented programming techniques to encapsulate and hide much of the ADOL-C interfacing details from the model code, and choosing to discretize time outside of device model code, *f*REEDA® has dramatically eased nonlinear circuit device model development. The guiding philosophy behind *f*REEDA® is to facilitate the implementation of models as close to the physics of devices as possible. In most cases, it is possible literally to code the constitutive equations for the device. *f*REEDA® is freely available under GNU Public License at `www.freeda.org`.

Adjoint Differentiation of a Structural Dynamics Solver*

Mohamed Tadjouddine[1], Shaun A. Forth[1], and Andy J. Keane[2]

[1] Applied Mathematics & Operational Research, ESD, Cranfield University (RMCS Shrivenham), Swindon, UK
{M.Tadjouddine,S.A.Forth}@cranfield.ac.uk
[2] Computational Engineering and Design Group, School of Engineering Sciences, University of Southampton, UK
andy.keane@soton.ac.uk

Summary. The design of a satellite boom using passive vibration control by Keane [J. of Sound and Vibration, 1995, 185(3), 441–453] has previously been carried out using an energy function of the design geometry aimed at minimising mechanical noise and vibrations. To minimise this cost function, a Genetic Algorithm (GA) was used, enabling modification of the initial geometry for a better design. To improve efficiency, it is proposed to couple the GA with a local search method involving the gradient of the cost function. In this paper, we detail the generation of an adjoint solver by automatic differentiation via ADIFOR 3.0. This has resulted in a gradient code that runs in 7.4 times the time of the function evaluation. This should reduce the rather time-consuming process (over 10 CPU days by using parallel processing) of the GA optimiser for this problem.

Key words: Reverse mode AD, hybrid GA-local search, structural dynamics, performance, ADIFOR

1 Introduction

In space missions, lightweight cantilever structures are often used to suspend scientific instruments, such as antenna, a few metres away from the satellite. An example of this kind of structure is the satellite boom shown in Fig. 1. Vibrations can be transmitted through the structure from satellite to the instrument. Such vibrations can damage the boom structure or prevent it from being used for its intended purpose. To ensure correct functioning of the instrument, the vibrations or mechanical noise through the structure must be kept at tolerable levels. Typically, the structure is excited by a point transverse force near an end beam, and the energy level is measured at the opposite end beam. To minimise vibrations and noise, the geometry of the structure is modified to reduce the frequency average response of the satellite boom. This is known as *passive vibration control* [296].

* This work is supported by UK's EPSRC under grant GR/R85358/01

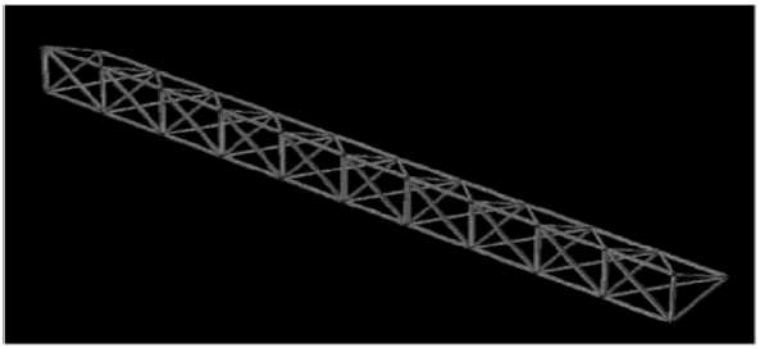


Fig. 1. Initial geometry of the satellite boom.

In this paper, we consider the satellite boom of Fig. 1 described in Sect. 2.1 and previously studied in [298, 384]. The structural dynamics of the three-dimensional satellite boom are modelled by a Fortran computer code named BEAM3D [476,477] using receptance theory, whereby the behaviour of the global structure is predicted from the Green functions of the individual components, evaluated as summations over their mode shapes.

In previous work [298,384], the minimisation of the frequency average response (in the range 150–250 Hz) at the end beam was carried out to find a superior design or geometry. For that purpose, a Genetic Algorithm (GA) was used. GAs are known to work for fairly large cost with a good chance of finding the global minimum. Generally, GAs do not require the gradient of the cost function. However, the application of GAs in large scale industrial applications is limited due to the large number of expensive evaluations of the cost function. For our application, the first 10 generations for a population size of 100 took over 10 days to complete using parallel processing [298].

Attention has now shifted to a hybrid genetic algorithm-local search approach combining Darwinian and Lamarckian evolution models. Darwinian evolution, based on "natural selection" considers that the most fit individuals are likely to survive. Lamarckian evolution takes the view that individuals may improve within their environment as they become adapted to it and that the resulting changes are inheritable. Consequently, the genotype of an improved individual is forced to reflect the result of such an improvement by replacing the individual into the population for reproduction [417]. In our application, the local search is carried out using a gradient method, which requires the calculation of the gradient of a cost function $F : \mathbb{R}^{90} \rightarrow \mathbb{R}$. Here, the 90 inputs are the coordinates of the joint positions in the design geometry and represent the independent variables. The computation of transmitted power $F(\mathbf{x})$ involves complex variable calculations, which are handled by ADIFOR 3.0 [96] since the dependents and independents are real values [445].

In theory, the gradient ∇F of such a function is cheaply calculated using the reverse mode since the cost of the gradient is independent of the number of the independents and is bounded above by a small factor of the cost of evaluating the function [225]. In practice, large memory requirement may prohibit use of the

adjoint code (reverse AD generated code). This paper details the differentiation of the BEAM3D code by the ADIFOR, the successor of the AD tool ADIFOR 2.0 [57]. ADIFOR employed in reverse mode produces an adjoint code which, after being tuned manually for performance enhancement, calculated the function and its gradient in 7.4 times the CPU time required for its function evaluation. Moreover, the adjoint code runs 12.6 times faster than one-sided finite-differencing (FD) on a Sun Blade 1000 machine with 1200 MHz CPU, 8 MB external cache and 2 GB RAM.

2 Optimisation of the Boom Structure

We aim at minimising vibrations through the structure represented in Fig. 1. There are at least three ways to achieve this: increasing the mass of elements or coating elements with damping material, using active anti-vibration to cancel unwanted vibrations; and as considered here, modifying the geometry of the structure to filter and reflect the vibrations.

2.1 The Initial Geometry

The initial boom structure to be optimised is three-dimensional and composed of 90 Euler-Bernouilli beams each having the same properties per unit length. Because the structure is used to mount a scientific instrument away from a space satellite, the length of the boom structure must be chosen within reasonable limits. Typical values of the aluminium were used for the physical properties of the beams. The bay length is 45 cm, and the overall length of the boom structure is 4.5 m.

The beams were arranged in a regular manner along the XYZ axes so that the YZ cross-section of the boom structure formed an equilateral triangle. The three joints at the left hand end of the structure were fixed, i.e., they were clamped to prevent motion. The beams were connected together with 30 free joints. Geometric constraints were used to avoid beams overlapping or becoming extremely long. The free joints were kept within fixed distances of their original positions. The connectivity of the diagonal beams was chosen so that a maximum of six beams met at any one joint.

Typically, the structure is excited by a point transverse force applied to a left hand end beam of the structure. The vibrational energy level is calculated at a right hand end beam using receptance methods [296]. The optimisation aims at minimising the vibrations by minimising the frequency averaged response in the range $150-250$ Hz. For that purpose, the optimiser is allowed to modify the geometry of the satellite boom by changing the coordinates of the 30 free joints in the structure.

2.2 An Optimised Geometry

A GA from the optimisation software package [297] was used to generate an optimised boom geometry by improving the frequency response curve. The principles of a GA can be found in [217]. In short, GAs work on the premise that a population of competing individuals can be combined to produce improved individuals. They mimic "natural" selection, or Darwinian evolution. The number of generations and their population size are usually chosen in advance. Common operations are:

- selection: whereby the fittest individuals are chosen to "inter-breed" and pass their attributes to their offspring.
- crossover: where random portions of two of the most fit individuals are combined to form a new individual.
- mutation: where small changes are introduced to one individual at a time.

As reported in [384], an optimised design geometry was obtained using an objective function F set to be the square root of the sum of velocity squared in the X, Y, Z directions for the end three joints labelled $31, 32$ and 33,

$$F = \int_{\text{freq}} \sum_{j=31}^{33} \sqrt{V_{xj}^2 + V_{yj}^2 + V_{zj}^2} \, .$$

A run of the GA for 10 generations and a population size of 300, gave the novel design geometry of the boom structure shown in Fig. 2. However, GAs applied to large-scale optimisation problems can take CPU days even using parallel processing [298]. To enhance their performance, they may be combined with local search methods.

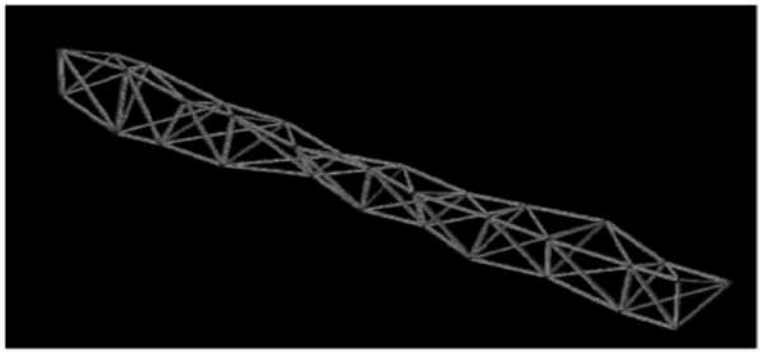


Fig. 2. An optimised geometry of the satellite boom.

2.3 Coupling GAs and Local Search Methods

For optimizing the design of the satellite boom of Fig. 1, a GA coupled with a local search method based on gradient descent should outperform a stand-alone GA. We aim to make them efficient by taking advantage of the accuracy and efficiency of gradient calculation by reverse mode AD.

In essence, the hybridised GA-Local Search method performs similar steps to that of the GA except that each individual of the population is locally improved using a local search method, here steepest descents, following the meta-Lamarkian learning approach as shown in Fig. 3 and detailed in [417]. In Fig. 3, the while loop is executed until either t exceeds some maximum number of generations t_{max} or convergence is detected. We now detail the differentiation of the BEAM3D code to enable the gradient descent method of the hybrid algorithm.

```
t = 0
Initialise a GA population P(t) = {x1, x2, ..., x^m}
While ( EndCondition is not satisfied )
     Evaluate Fitness(P(t)) giving F(x1), F(x2), ..., F(x^m)
     For each individual x^i ∈ P(t)
          Improve x^i using the Gradient Descent method GD(x^i)
          Replace x^i by the improved x^i_new = GD(x^i) in P(t)
     EndFor
     Generate P(t+1) from the x^i_new by using standard
     GA operations (Selection, Mutation, or Crossover).
     t = t + 1
EndDo
```

Fig. 3. Hybrid GA-Gradient Descent Method.

3 Differentiation of the BEAM3D Code

The BEAM3D code, as sketched in Fig. 4, starts by reading in data from files representing the boom geometry and certain properties of each beam. Given extra information such as the range of frequencies over which to solve, the number of data points within the specified frequency range, the joint numbers at which to calculate the energies, and the number of axial[torsional] and transverse modes in the modal summations for the Green functions; the program builds up a linear complex system $\mathbf{Af} = \mathbf{b}$ for nodal forces $\mathbf{f}$ and solves it for each frequency. An averaged energy function is calculated at the specified end beam.

```
Read in:
     No. of beams, beam properties, and list of connections
     Frequency range [ω_min, ω_max, N] (N ≡ No. of frequencies)
     Coordinates of beam ends x_{n,j}
Calculate some geometric information
Initialise F = 0 (integral of power)
Δω = (ω_max − ω_min)/(N − 1)
For k = 1, N
     ω = ω_min + (k − 1) * Δω
     Assemble:
          Green Function Matrix A(x, iω)
          r.h.s. forcing b(x, iω)
     Solve A(x, iω)f = b(x, iω) (LAPACK) - 50% of CPU time
     Obtain displacement D_{n,j} at ends of beam n
     Obtain power P = (1/2) Re(f_j · D_{n,j})
     Update integral F = F + P2 * Δω
End For
```

Fig. 4. Schematic of the BEAM3D code.

We aim to calculate the sensitivities of the energy function with respect to the coordinates of the free joints. This represents a gradient calculation with 90 independent variables. Prior to differentiation, the code was restructured so all reading of data is done outside the subroutines to be differentiated. Furthermore, to allow the code to be processed by ADIFOR, the code was rewritten according to the Fortran 77 standard. Actually, the original code contained (non-standard) language extensions in the form of structures defined as follows:

```
STRUCTURE /PROPERTY/
          INTEGER ID
          CHARACTER*6 ENDCON
          DOUBLE PRECISION ANGLE(3,3)
          DOUBLE PRECISION LENG
          ...
          COMPLEX *16  FM2
 END STRUCTURE
 RECORD /PROPERTY/ BEAM(150)
```

This structure is replaced using arrays corresponding to the components of the structure. The restructured code contains the following array declarations:

```
INTEGER BEAM_ID(150)
CHARACTER*6 BEAM_ENDCON(150)
DOUBLE PRECISION BEAM_ANGLE(3,3,150)
DOUBLE PRECISION BEAM_LENG(150)
...
COMPLEX*16 BEAM_FM2(150)
```

A `sed` [153] script was written to replace any instance `BEAM(I).X(K,J)`, where `X` represents any component of the structure, by the array element `BEAM_X(K,J,I)`. If `X` is a scalar variable, obviously no indices are used. The resulting computer code is differentiated using FD, and AD via ADIFOR.

3.1 Initial Differentiation

We first computed a single directional derivative $\dot{\mathbf{y}} = \nabla F(\mathbf{x})\dot{\mathbf{x}}$ for a random direction $\dot{\mathbf{x}}$, by using one sided FD, AD in forward mode, and a single adjoint $\bar{\mathbf{x}} = \nabla F(\mathbf{x})^T\bar{\mathbf{y}}$ for $\bar{\mathbf{y}} = 1$ via reverse mode AD. By definition of the adjoint operator, we have $\bar{\mathbf{y}}\dot{\mathbf{y}} \equiv \bar{\mathbf{x}}\dot{\mathbf{x}}$, which allows us to validate the results of the differentiation. The initial ADIFOR generated codes gave incorrect results inconsistent with those from FD, caused partly by non-differentiable statements in the code for the function F.

3.2 Dealing With Non-Differentiability

A major assumption in AD is that the function F to be differentiated is composed of elemental functions ϕ that are continuously differentiable on their open domains [225]. At a point on the boundary of an open domain, F is continuous, but ∇F may jump to a finite value or even infinity. This is important when the computer code that represents the function contains branches, some kink functions

(e.g., `abs`), or inverse functions (e.g., `sqrt`, `arctan`, or `arccos`). To compute reliable derivatives, such pathological cases must be handled correctly. It is known that these cases can be tackled by calculating derivatives in a given direction [225]. Insights or knowledge of the computer code can also be exploited. The BEAM3D code contains at least two types of non-differentiability.

```
l2 = datan2(xdiff,zdiff)
m2 = dsqrt(xdiff*xdiff+zdiff*zdiff)/beam_leng(i)
sgn1 = -1.0d0
sgn2 = -1.0d0
sgn3 = -1.0d0
if (xdiff.lt.0.0d0) sgn1 = -sgn1
if (ydiff.gt.0.0d0) sgn2 = -sgn2
if (zdiff.lt.0.0d0) sgn3 = -sgn3
yor(1,i) = sgn1*dabs(dsin(dacos(m2))*dsin(l2))
yor(2,i) = sgn2*dabs(m2)
yor(3,i) = sgn3*dabs(dsin(dacos(m2))*dcos(l2))
```

Fig. 5. A code fragment that is non-differentiable.

The first type of non-differentiability is due to the presence of the functions `arccos` and `abs`. ADIFOR allows us to locate possible non-differentiable points by generating the derivative code with the *Exception Handling* enabled [96]. On running this code, warnings were raised concerning the functions `arccos` and `abs`. The part of the code, containing such anomalies is shown in Fig. 5. We then used algebra and trigonometric formula to rewrite some of the algebraic expressions containing (`arccos`, `abs`, `sin` and `tan`) as in Fig. 6.

This transformation resulted in equivalent expressions calculating the same values but differentiable in the vicinity of their arguments.

```
myt1 = ydiff/beam_leng(i)
myt2 = dsqrt(xdiff*xdiff+zdiff*zdiff)
sgn1 = -1.0d0
sgn2 = -1.0d0
sgn3 = -1.0d0
if (xdiff.lt.0.0d0) sgn1 = -sgn1
if (ydiff.gt.0.0d0) sgn2 = -sgn2
if (zdiff.lt.0.0d0) sgn3 = -sgn3
yor(1,i) = sgn1*myt1*xdiff/myt2
yor(2,i) = sgn2*myt2/beam_leng(i)
yor(3,i) = sgn3*myt1*zdiff/myt2
```

Fig. 6. An equivalent but differentiable version of the code fragment of Fig. 5.

The second type of non-differentiability encountered in BEAM3D was due to a branching construct illustrated by the code fragment of Fig. 7. As `xdiff` and `ydiff` are active variables, the differentiation of this code fragment gave point-valued derivatives that prevented the function **F** from being differentiable. Such branches represent constraints on the design geometry and, in our case, may be safely removed.

```
if (xdiff.eq.0.0 .and. ydiff.eq.0.0)
then
   yor(1,i) = zdiff/beam_leng(i)
   yor(2,i) = 0.0
   yor(3,i) = 0.0
else ....
```

Fig. 7. A code fragment testing whether an active variable is zero.

Finally, the complex linear solver **Af**=**b** employs the LAPACK routine `zgesv` [7]. Differentiating the LAPACK source code routines for `zgesv` using an AD tool without taking account insights into the nature of the linear solver would give inefficient code. Mechanical generation of the `zgesv` derivative by ADIFOR gave not only inefficient code but also results inconsistent with FD. Therefore, we hand-coded its derivative as described in Sect. 3.3.

3.3 Complex Linear Solver

Instead of using ADIFOR to differentiate the complex linear solve,

$$\mathbf{A}\mathbf{f} = \mathbf{b} \,, \tag{1}$$

of the LAPACK routine `zgesv`, we instead use hand-coding for both forward and reverse mode. Differentiating $\mathbf{A}\mathbf{f} = \mathbf{b}$, using the matrix-equivalent of the product rule for a single directional derivative, we obtain $\mathbf{A}\dot{\mathbf{f}} + \dot{\mathbf{A}}\mathbf{f} = \dot{\mathbf{b}}$, and giving the derivatives $\dot{\mathbf{f}}$ by the solution of

$$\mathbf{A}\dot{\mathbf{f}} = \dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{f} \,. \tag{2}$$

In the forward mode, we may re-use the LU-decomposition of **A** to solve efficiently for the derivatives $\dot{\mathbf{f}}$. The following procedure is used:

1. Perform an **LU** decomposition of the matrix **A**
2. Solve $\mathbf{A}\mathbf{f} = \mathbf{b}$
3. Form $\mathbf{b}_{new} = \dot{\mathbf{b}} - \dot{\mathbf{A}}\mathbf{f}$
4. Re-use LU-decomposition to solve $\mathbf{A}\dot{\mathbf{f}} = \mathbf{b}_{new}$

$\dot{\mathbf{A}}$ and $\dot{\mathbf{b}}$ are calculated by applying ADIFOR to the `Assemble` procedure in Fig. 4.

For the reverse mode, deriving the adjoint update corresponding to (1) is more problematic. Defining $\mathbf{C} = \mathbf{A}^{-1}$, we may write

$$\dot{\mathbf{f}} = \mathbf{C}\dot{\mathbf{b}} - \mathbf{C}\dot{\mathbf{A}}\mathbf{f} \,.$$

Then $\dot{f}_i$, the i^{th} element of $\dot{\mathbf{f}}$, is given by

$$\dot{f}_i = \sum_j c_{ij}\dot{b}_j - \sum_j c_{ij} \sum_k \dot{a}_{jk} f_k \,, \tag{3}$$

where c_{ij}, $\dot{a}_{jk}$, $\dot{b}_j$, and f_k are the elements of $\mathbf{C}$, $\dot{\mathbf{A}}$, $\dot{\mathbf{b}}$, and $\mathbf{f}$, respectively. Now we use the identity $\bar{\mathbf{y}}\dot{\mathbf{y}} \equiv \bar{\mathbf{x}}\dot{\mathbf{x}}$ for the system $\mathbf{y} = \mathbf{F}(\mathbf{x})$ [225, Equation (3.7)]. In the context of the vector and matrix arguments of the system (1), this identity gives

$$\sum_i \bar{f}_i \dot{f}_i = \sum_i \bar{b}_i \dot{b}_i + \sum_i \sum_j \bar{a}_{ij} \dot{a}_{ij} \ . \tag{4}$$

From (3), we obtain

$$\sum_i \bar{f}_i \dot{f}_i = \sum_i \bar{f}_i \sum_j c_{ij} \dot{b}_j - \sum_i \bar{f}_i \sum_j c_{ij} \sum_k \dot{a}_{jk} f_k \ ,$$

by reordering summations,

$$\sum_i \bar{f}_i \dot{f}_i = \sum_j \dot{b}_j \sum_i c_{ij} \bar{f}_i - \sum_j \sum_k \dot{a}_{jk} f_k \sum_i c_{ij} \bar{f}_i \ ,$$

and by swapping indices (i, j, k) to (k, i, j),

$$\sum_i \bar{f}_i \dot{f}_i = \sum_i \dot{b}_i \sum_j c_{ji} \bar{f}_j - \sum_i \sum_j \dot{a}_{ij} f_j \sum_k c_{ki} \bar{f}_k \ .$$

Comparing with (4), we see that

$$\bar{b}_i = \sum_j c_{ji} \bar{f}_j \ ,$$

giving $\bar{\mathbf{b}} = \mathbf{C}^T \bar{\mathbf{f}} = \mathbf{A}^{-T} \bar{\mathbf{f}}$, or that $\bar{\mathbf{b}}$ is the solution of

$$\mathbf{A}^T \bar{\mathbf{b}} = \bar{\mathbf{f}} \ . \tag{5}$$

Similarly,

$$\bar{a}_{ij} = -f_j \sum_k c_{ki} \bar{f}_k = -f_j \bar{b}_i \ , \text{ or}$$

$$\bar{\mathbf{A}} = -\bar{\mathbf{b}} \mathbf{f}^T \ . \tag{6}$$

The adjoint $\bar{\mathbf{b}}$ is updated by solving the linear system (5), while $\bar{\mathbf{A}}$ is updated by adding the right hand side of the equation (6). Adjoint formulae (5) and (6) are equivalent to those given in [520].

Using (5) and (6), we obtain the following procedure for the adjoint of the linear solve:

1. In the forward sweep,
   a) Perform an $\mathbf{LU}$ decomposition of the matrix $\mathbf{A}$,
   b) Store $\mathbf{L}$ and $\mathbf{U}$ and the pivot sequence `IPIV`,
   c) Solve $\mathbf{Af} = \mathbf{b}$.
2. In the reverse sweep,
   a) Load $\mathbf{L}$ and $\mathbf{U}$ and the pivot sequence `IPIV`,
   b) Solve $\mathbf{A}^T \bar{\mathbf{b}} = \bar{\mathbf{f}}$ for $\bar{\mathbf{b}}$,
   c) Update $\bar{\mathbf{A}} = \bar{\mathbf{A}} - \bar{\mathbf{b}} \mathbf{f}^T$.

Since both $\mathbf{A}$ and $\mathbf{b}$ depend on the beam endpoint location $\mathbf{x}$ (see Fig. 4), the adjoined linear solver must modify their adjoints $\bar{\mathbf{A}}$ and $\bar{\mathbf{b}}$. The adjoint for $\bar{\mathbf{A}}$ is the usual increment, while for $\bar{\mathbf{b}}$ it is an assignment because the LAPACK routine `zgesv` overwrites $\mathbf{b}$ with $\mathbf{f}$. Here, the memory storage is dramatically reduced compared with black-box application of ADIFOR. If $\mathbf{A}$ is an $N \times N$ matrix, we store only $N2$ complex coefficients instead of $O(N3)$ when ADIFOR tapes all variables on the left of assignment statements in the LU decomposition.

3.4 Initial Results and Validation

After implementing the procedures described in Sect. 3.2 and 3.3 on the ADIFOR generated code, we obtained tangent and adjoint derivative codes that calculate directional derivatives consistent with one-sided FD. The obtained codes were compiled with maximum compile optimisations and run on a Sun Blade 1000 machine. Table 1 shows the results and timings of forward mode AD, reverse mode AD, and one-sided FD for that calculation. These results showed that forward and reverse AD gave the same directional derivative value within roundoff, while the maximum difference with the FD result is around 10^{-6}. This difference is of the order of the square root of the machine relative precision. This validates the AD results as being in agreement with the one-sided FD result.

Table 1. Results for a single directional derivative, timings are in CPU seconds.

Method	$\bar{\mathbf{x}}\dot{\mathbf{x}}$	$\bar{\mathbf{y}}\dot{\mathbf{y}}$	CPU$(F, \nabla F)$
FD (1-sided)		0.124578003587	48.7
ADIFOR(fwd)		0.124571139127	54.0
ADIFOR(rev)	0.124571139130		311.5

From Table 1, we see that while the AD reverse mode calculates the gradient in around 5 minutes, one-sided FD and forward AD requires 91 function evaluations and 90 directional derivatives respectively and consequently run times of over 35 minutes. We see that using reverse mode AD can speed up the gradient calculation by a factor of around 7 over FD while giving accurate derivatives. However, the core of the calculation (building the linear system, solving it, and calculating the local energy contribution for each frequency) of the BEAM3D code is an independent loop and therefore can be differentiated in parallel as we now describe.

4 Performance Issues

Usually, after checking that the AD forward and reverse modes agreed with the finite differences, we seek to improve efficiency of the automatically generated code. As shown by the results of Table 1, the reverse mode is superior to the finite differences and forward mode, but it requires a very large amount of memory to run because the tape required 12 GB. By hand coding the adjoint of the linear solver, we reduced the size of the tape to around 6 GB.

Furthermore, the core of the calculation of the BEAM3D code is carried out in a parallel loop, which is the loop over k in Fig. 4. Because the iterations of such a loop are independent, we can run the loop body taping all the required information in just one iteration, then immediately adjoin the body of the loop [254]. This reduced the tape size of the adjoint code to around 0.3 GB. This represents a memory reduction by a factor of 20, the number of extra iterations performed by the parallel loop.

The second row of Table 2 shows that after this optimisation, the ratio between the gradient calculation and the function is 7.4. It also shows a speed up factor of 12.6 over the popular one-sided FD method.

Table 2. CPU Timings (in Seconds) on a SUN Blade 1000, UltraSparcIII.

Method	CPU($F, \nabla F$)	CPU($F, \nabla F$)/CPU(F)
ADIFOR(rev.)	311.5	13.3
ADIFOR(rev.,par.)	174.7	7.4
FD (1-sided)	2215.9	93.1

5 Conclusions

ADIFOR allowed us to build an adjoint for a code that makes extensive use of complex variable arithmetic to accurately calculate the gradient of a cost function. The adjoint code requires only 7.4 times the CPU time of the original function code, and the memory requirement for taping is a modest 0.3 GB. It also runs 12.6 times faster than calculating the gradient using one-sided finite differencing.

Future work is planned to compare the performance of gradient calculation using both ADIFOR [96] and TAF [163] capabilities. The design optimisation of the lightweight cantilever structure will be carried out using the meta-Lamarckian learning strategy [417], which efficiently combines GAs with local search methods. The reduction in computational time of the gradient calculation will be of great benefit in allowing the meta-Lamarckian algorithm to be used to optimise the design of boom structures.

Acknowledgements

The authors thank Mike Fagan for his help in using ADIFOR.

A Bibliography of Automatic Differentiation

H. Martin Bücker[1] and George F. Corliss[2]

[1] Institute for Scientific Computing, RWTH Aachen University, Aachen, Germany
buecker@sc.rwth-aachen.de
[2] Electrical and Computer Engineering, Marquette University, Milwaukee, USA
George.Corliss@Marquette.edu

Summary. This is a bibliography of scientific literature cited by all chapters in the volume, *Automatic Differentiation: Applications, Theory, and Implementations*, Martin Bücker, George Corliss, Paul Hovland, Uwe Naumann, and Boyana Norris (eds.), Springer, New York, 2005 [78]. The collection contains more than 570 references, mostly related to automatic differentiation.

Key words: Automatic differentiation, autodiff.org

Comments

Following the tradition of the collections devoted to the three previous international conferences on automatic differentiation at Breckenridge [227], Santa Fe [42] and Nice [136], this bibliography represents the common list of references for all chapters in this volume [78]. For each chapter, the authors compiled a separate bibliography; the resulting bibliographies were merged into a single BibTeX database. Since all papers from the three previous volumes [42, 136, 227] are also contained, the present database includes most widely cited work in automatic differentiation (AD) as well as many references from other scientific disciplines that are not directly related to AD, but in which AD applications have been described in [78].

The Web site `http://www.autodiff.org` is currently set up to serve as the central Web-based information resource for the AD community. The BibTeX database of this volume [78] will be available there. While still in its infancy, the community portal tries to be useful for research in AD by providing an extensive AD publication database. To this end, the collected BibTeX bibliographies by Corliss [132, 134, 566] were taken as the starting point, from which all references not directly related to AD were removed. The goal is to provide a collection of AD references that are structured by a coarse classification scheme consisting of the three categories

- application area,
- tools,
- theory and techniques.

Syntactically, the three categories are specified by three additional fields in the database, namely `ad_area`, `ad_tools` and `ad_theotec`, which are ignored by BibTeX. An entry in this database is classified by specifying one or more of these categories. For instance, a paper using the AD tool TAF in a chemical application could be classified by adding the following two fields:

```
@article{key,
   . . .
   ad_tools = "TAF",
   ad_area  = "Chemistry",
}
```

As a second example, consider a more theoretical paper investigating a checkpointing strategy. Adding the following field would be an appropriate classification:

```
@article{key,
   . . .
   ad_theotec = "Checkpointing, Reverse Mode",
}
```

The benefit of this classification is that, by searching the publication database of `autodiff.org`, the AD research community can find adequate references on a certain topic. Though the number of references that are currently classified is limited, the database is already useful today for finding AD references on, say, parallelism.

More information on autodiff.org's classification system is available at `http://www.autodiff.org/Publications/info.php`. The editors hope that researchers publishing in the field of AD will actively classify and submit BibTeX entries of their work to autodiff.org, helping other researchers to find their way through the extensive literature of automatic differentiation.

References

1. J. Abate, S. Benson, L. Grignon, P. D. Hovland, L. C. McInnes, and B. Norris. Integrating AD with object-oriented toolkits for high-performance scientific computing. In Corliss et al. [136], chapter 20, pages 173–178.
2. M. Abramowitz and A. Stegun. *Handbook of Mathematical Functions*. National Bureau of Standards, 1964.
3. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.
4. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
5. M. M. Alsharo'a et al. Recent progress in neutrino factory and muon collider research within the Muon Collaboration. *Physical Review ST-AB*, 6:081001, 2003.
6. P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.
7. E. Anderson, Z. Bai, C. H. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.
8. J. A. Anderson and E. Rosenfeld, editors. *Talking Nets: An Oral History of Neural Networks*. MIT Press, Cambridge, MA, 1998.
9. J. Andersson. A survey of multiobjective optimization in engineering design. Technical Report LiTH–IKP–R–1097, Department of Mechanical Engineering, Linkoping University, 2000.
10. A. Arakawa. Computational design for long-term numerical integrations of the equations of atmospheric motion. Part I: Two dimensional incompressible flow. *Journal of Computational Physics*, 1:119–143, 1966.
11. G. J. Arcement and V. R. Schneider. Guide for selecting Manning's roughness coefficients for natural channels and flood plains. Technical Report FHWA–TS–84–204, U.S. Department of Transportation, Federal Highways Administration, 1984.
12. U. Ascher, R. Mattheij, and R. Russell. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*. SIAM, Philadelphia, PA, 1995.
13. P. Aubert and N. Di Césaré. Expression templates and forward mode automatic differentiation. In Corliss et al. [136], chapter 37, pages 311–315.

14. P. Aubert, N. Di Césaré, and O. Pironneau. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualization in Science*, 3:197–208, 2001.
15. autodiff.org. The 4th International Conference on Automatic Differentiation. `http://www.autodiff.org/ad04`, 2004.
16. autodiff.org. List of automatic differentiation tools. `http://www.autodiff.org/Tools`, 2005.
17. B. M. Averick, R. G. Carter, J. J. Moré, and G.-L. Xue. The MINPACK-2 test problem collection. Preprint MCS–P153–0692, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1992.
18. A. Back, J. Guckenheimer, and M. Myers. A dynamical simulation facility for hybrid systems. In R. L. Grossmann, A. Nerode, and A. P. Ravn, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 255–267. Springer, New York, NY, 1993.
19. E. Balsa-Canto, J. R. Banga, A. A. Alonso, and V. S. Vassiliadis. Dynamic optimization of distributed parameter systems using second order directional derivatives. *Industrial and Engineering Chemistry Research*, 43:6756–6765, 2004.
20. A. R. Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information Theory*, 39(3):930–945, 1993.
21. M. C. Bartholomew-Biggs, S. Brown, B. Christianson, and L. C. W. Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124:171–190, 2000.
22. R. A. Bartlett. Private communication, 2004.
23. D. Barton, I. M. Willers, and R. V. M. Zahar. The automatic solution of ordinary differential equations by the method of Taylor series. *The Computer Journal*, 14(3):243–248, 1970.
24. D. Barton, I. M. Willers, and R. V. M. Zahar. Taylor series methods for ordinary differential equations — An evaluation. In J. Rice, editor, *Mathematical Software*, pages 369–390. Academic Press, New York, NY, 1971.
25. P. I. Barton and C. C. Pantelides. Modeling of combined discrete/continuous processes. *AIChE Journal*, 40:966–979, 1994.
26. L. A. Bastidas, H. V. Gupta, S. Sorooshian, W. J. Shuttleworth, and Z. L. Yang. Sensitivity analysis of a land surface scheme using multi-criteria methods. *Journal of Geophysical Research*, 104(D16):19,481–19,490, 1999.
27. P. D. Bates, K. J. Marks, and M. S. Horritt. Optimal use of high-resolution topographic data in flood inundation models. *Hydrological Processes*, 17:537–557, 2003.
28. F. Bauer. Computational graphs and rounding error. *SIAM Journal on Numerical Analysis*, 11(1):87–96, 1974.
29. J.-D. Beley, S. Garreau, F. Thevenon, and M. Masmoudi. Application of higher order derivatives to parameterization. In Corliss et al. [136], chapter 40, pages 335–341.
30. E. T. Bell. *The Development of Mathematics*. McGraw-Hill, New York, NY, 2nd edition, 1945.
31. D. A. Belsley, A. E. Kuh, and R. E. Welsch. *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*. John Wiley & Sons, New York, NY, 1980.
32. A. Ben-Haj-Yedder, E. Cances, and C. Le Bris. Optimal laser control of chemical reactions using AD. In Corliss et al. [136], chapter 24, pages 205–211.

33. J. Benary. Parallelism in the reverse mode. In Berz et al. [42], chapter 12, pages 137–147.
34. C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, Aug. 1996.
35. D. Bernstein and I. Gertner. Scheduling expressions on a pipelined processor with a maximun delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11:57–66, 1989.
36. M. Berz. Algorithms for higher derivatives in many variables with applications to beam physics. In Griewank and Corliss [227], chapter 15, pages 147–156.
37. M. Berz. Calculus and numerics on Levi-Civita fields. In Berz et al. [42], chapter 2, pages 19–35.
38. M. Berz. *Modern Map Methods in Particle Beam Physics*. Academic Press, San Diego, CA, 1999.
39. M. Berz. COSY INFINITY Version 8.1 — User's Guide and Reference Manual. Technical Report MSUHEP–20704, Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, 2002.
40. M. Berz. Towards a universal data type for scientific computing. In Corliss et al. [136], chapter 45, pages 373–381.
41. M. Berz. Introduction to beam physics. Lecture Notes, 2004. Virtual University Beam Physics Course.
42. M. Berz, C. H. Bischof, G. F. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, PA, 1996.
43. M. Berz, J. Hoefkens, and K. Makino. COSY INFINITY Version 8.1 — programming manual. Technical Report MSUHEP–20703, Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, 2001.
44. M. Berz and K. Makino. Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. *Reliable Computing*, 4(4):361–369, 1998.
45. M. Berz, K. Makino, K. Shamseddine, G. H. Hoffstätter, and W. Wan. COSY INFINITY and its applications in nonlinear dynamics. In Berz et al. [42], chapter 32, pages 363–365.
46. K. J. Beven and A. M. Binley. The future of distributed models: Model calibration and predictive uncertainty. *Hydrological Processes*, 6:279–298, 1992.
47. K. J. Beven and J. Freer. Equifinality, data assimilation, and uncertainty estimation in mechanistic modelling of complex environmental systems using the GLUE methodology. *Journal of Hydrology*, 249(1–4):11–29, 2001.
48. U. S. Bhalla and J. M. Bower. Exploring parameter space in detailed single neuron models: Simulations of the mitral and granule cells of the olfactory bulb. *Journal of Neurophysiology*, 69(6):1948–1964, 1993.
49. C. H. Bischof. Issues in parallel automatic differentiation. In Griewank and Corliss [227], chapter 11, pages 100–113.
50. C. H. Bischof, H. M. Bücker, and D. an Mey. A case study of computational differentiation applied to neutron scattering. In Corliss et al. [136], chapter 6, pages 69–74.
51. C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and A. Vehreschild. Combining source transformation and operator overloading techniques to compute

derivatives for MATLAB programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 65–72, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
52. C. H. Bischof, H. M. Bücker, W. Marquardt, M. Petera, and J. Wyes. Transforming equation-based models in process engineering. In Bücker et al. [78], pages 189–198.
53. C. H. Bischof, H. M. Bücker, and A. Vehreschild. A macro language for derivative definition in ADiMat. In Bücker et al. [78], pages 181–188.
54. C. H. Bischof and A. Carle. Users' experience with ADIFOR 2.0. In Berz et al. [42], chapter 35, pages 385–392.
55. C. H. Bischof, A. Carle, G. F. Corliss, A. Griewank, and P. D. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1:11–29, 1992.
56. C. H. Bischof, A. Carle, P. D. Hovland, P. Khademi, and A. Mauer. ADIFOR 2.0 user's guide (revision D). Technical Memorandum ANL/MCS–TM–192, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1998.
57. C. H. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.
58. C. H. Bischof, G. F. Corliss, and A. Griewank. Structured second- and higher-order derivatives through univariate Taylor series. *Optimization Methods and Software*, 2:211–232, 1993.
59. C. H. Bischof and M. R. Haghighat. Hierarchical approaches to automatic differentiation. In Berz et al. [42], chapter 7, pages 83–94.
60. C. H. Bischof and P. D. Hovland. Automatic differentiation: Parallel computation. In C. A. Floudas and P. M. Pardalos, editors, *Encyclopedia of Optimization*, volume I, pages 102–108. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.
61. C. H. Bischof, P. D. Hovland, and B. Norris. Implementation of automatic differentiation tools. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 98–107, New York, NY, USA, 2002. ACM Press.
62. C. H. Bischof, B. Lang, and A. Vehreschild. Automatic differentiation for MATLAB programs. *Proceedings in Applied Mathematics and Mechanics*, 2(1):50–53, 2003.
63. C. H. Bischof, L. Roh, and A. Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.
64. S. Blessing. Development and applications of an adjoint GCM. Master's thesis, University of Hamburg, Hamburg, Germany, 2000.
65. F. Bodin and A. Monsifrot. Performance issues in automatic differentiation on superscalar processors. In Corliss et al. [136], chapter 4, pages 51–57.
66. G. Bohlender, C. Ullrich, J. W. von Gudenberg, and L. B. Rall. *Pascal-SC, A Computer Language for Scientific Computation.* Academic Press, New York, NY, 1987.
67. C. F. Bornstein, B. M. Maggs, and G. L. Miller. Tradeoffs between parallelism and fill in nested dissection. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 191–200. ACM, 1999.

68. F. Bouttier and P. Courtier. Data assimilation concepts and methods. Meteorological Training Course Lecture Series, ECMWF, 1999.
69. T. Braconnier and P. Langlois. From rounding error estimation to automatic correction with automatic differentiation. In Corliss et al. [136], chapter 42, pages 351–357.
70. A. Brahme. Optimization of radiation therapy and the development of multileaf collimation. *International Journal of Radiation Oncology Biology Physics*, 25:373–375, 1993.
71. M. Brendel, J. Oldenburg, M. Schlegel, and K. Stockmann. DyOS 2.1 user's guide. Technical Report LPT–pro–2002–104, Process Systems Engineering, RWTH Aachen University, 2002.
72. K. S. Brown and J. P. Sethna. Statistical mechanics approaches to models with many poorly known parameters. *Physical Review E*, 68(9):21904–21904, 2003.
73. P. Brown, A. C. Hindmarsh, and L. R. Petzold. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM Journal on Scientific Computing*, 15:1467–1488, 1994.
74. P. Brown, A. C. Hindmarsh, and H. Walker. Experiments with quasi-Newton methods in solving stiff ODE systems. *SIAM Journal on Statistical and Scientific Computing*, 6:297–313, 1985.
75. A. E. Bryson and Y. C. Ho. *Applied Optimal Control*. Ginn, Waltham, MA, 1969.
76. M. Bucher, C. Lallement, C. Enz, F. Theodoloz, and F. Krummenacher. The EPFL-EKV MOSFET model equation for simulation. Technical Report Model Version 2.6, Swiss Federal Institute of Technology (EPFL), Lausanne, Switzerland, June 1997.
77. H. M. Bücker and G. F. Corliss. A bibliography on automatic differentiation. In Bücker et al. [78], pages 319–352.
78. H. M. Bücker, G. F. Corliss, P. D. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*. Springer, New York, NY, 2005.
79. H. M. Bücker, B. Lang, D. an Mey, and C. H. Bischof. Bringing together automatic differentiation and OpenMP. In *Proceedings of the 15th ACM International Conference on Supercomputing, Sorrento, Italy, June 17–21, 2001*, pages 246–251, New York, NY, 2001. ACM Press.
80. H. M. Bücker, B. Lang, A. Rasch, and C. H. Bischof. Computation of sensitivity information for aircraft design by automatic differentiation. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 1069–1076, Berlin, 2002. Springer.
81. H. M. Bücker, B. Lang, A. Rasch, and C. H. Bischof. Automatic parallelism in differentiation of Fourier transforms. In *Proceedings of the 18th ACM Symposium on Applied Computing, Melbourne, Florida, USA, March 9–12, 2003*, pages 148–152, New York, NY, 2003. ACM Press.
82. H. M. Bücker and A. Rasch. Efficient derivative computations in neutron scattering via interface contraction. In *Proceedings of the 17th ACM Symposium on Applied Computing, Madrid, Spain, March 10–14, 2002*, pages 184–188, New York, NY, 2002. ACM Press.

83. H. M. Bücker and A. Rasch. Modeling the performance of interface contraction. *ACM Transactions on Mathematical Software*, 29(4):440–457, 2003.
84. H. M. Bücker, A. Rasch, E. Slusanschi, and C. H. Bischof. Delayed propagation of derivatives in a two-dimensional aircraft design optimization problem. In D. Sénéchal, editor, *Proceedings of the 17th Annual International Symposium on High Performance Computing Systems and Applications and OSCAR Symposium, Sherbrooke, Québec, Canada, May 11–14*, pages 123–126. NRC Research Press, 2003.
85. C. Büskens and H. Maurer. Sensitivity analysis and real-time control of parametric optimal control problems using nonlinear programming methods. In M. Grötschel, S. O. Krumke, and J. Rambau, editors, *Online Optimization of Large Scale Systems*, pages 57–68. Springer, 2001.
86. D. G. Cacuci. Sensitivity theory for nonlinear systems. I. Nonlinear functional analysis approach. *Journal of Mathematical Physics*, 22(12):2794–2802, 1981.
87. D. G. Cacuci. *Sensitivity and Uncertainty Analysis, Volume I: Theory*. Chapman & Hall/CRC, Boca Raton, FL, 2003.
88. J.-B. Caillau and J. Noailles. Continous optimal control sensitivity analysis with AD. In Corliss et al. [136], chapter 11, pages 109–115.
89. S. L. Campbell and R. Hollenbeck. Automatic differentiation and implicit differential equations. In Berz et al. [42], chapter 19, pages 215–227.
90. E. B. Canto, J. R. Banga, A. A. Alonso, and V. S. Vassiliadis. Restricted second order information for the solution of optimal control problems using control vector parameterization. *Journal of Process Control*, 12:243–255, 2002.
91. Y. Cao, S. Li, and L. Petzold. Adjoint sensitivity analysis for differential-algebraic equations: Algorithms and software. *Journal of Computational and Applied Mathematics*, 149:171–191, 2002.
92. Y. Cao, S. Li, L. Petzold, and R. Serban. Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution. *SIAM Journal on Scientific Computing*, 24(3):1076–1089, 2003.
93. CAPE-OPEN Laboratories Network. `http://www.colan.org`, 2005.
94. B. Cappelaere, D. Elizondo, and C. Faure. Odyssée versus hand differentiation of a terrain modeling application. In Corliss et al. [136], chapter 7, pages 75–82.
95. A. Carle and M. Fagan. Improving derivative performance for CFD by using simplified recurrences. In Berz et al. [42], chapter 30, pages 343–351.
96. A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM–TR–00–02, Department of Computational and Applied Mathematics, Rice University, Houston, TX, 2000.
97. A. Carle and M. Fagan. Automatically differentiating MPI-1 datatypes: The complete story. In Corliss et al. [136], chapter 25, pages 215–222.
98. D. Casanova, R. S. Sharp, M. Final, B. Christianson, and P. Symonds. Application of automatic differentiation to race car performance optimisation. In Corliss et al. [136], chapter 12, pages 117–124.
99. R. J. Casey. *Periodic Orbits in Neural Models: Sensitivity Analysis and Algorithms for Parameter Estimation*. PhD thesis, Cornell University, Ithaca, NY, 2004.
100. W. Castaings, D. Dartus, M. Honnorat, F.-X. Le Dimet, Y. Loukili, and J. Monnier. Automatic differentiation: A tool for variational data assimilation and adjoint sensitivity analysis for flood modeling. In Bücker et al. [78], pages 249–262.

101. Y. F. Chang and G. F. Corliss. ATOMFT: Solving ODEs and DAEs using Taylor series. *Computers & Mathematics with Applications*, 28(10–12):209–233, 1994.
102. B. Chapman and H. Zima. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. Addison-Wesley, New York, NY, 1990.
103. B. W. Char. Computer algebra as a toolbox for program generation and manipulation. In Griewank and Corliss [227], chapter 6, pages 53–60.
104. V. I. Charney. Two methods of integrating the equations of motion. *Cosmic Research*, 8(5):676–683, 1970. (In Russian).
105. I. Charpentier. Checkpointing schemes or adjoint codes: Application to the meteorological model Meso-NH. *SIAM Journal on Scientific Computing*, 22(6):2135–2151, 2001.
106. I. Charpentier and M. Ghemires. Efficient adjoint derivatives: Application to the meteorological model Meso-NH. *Optimization Methods and Software*, 13(1):35–63, 2000.
107. I. Charpentier, N. Jakse, and F. Veersé. Second order exact derivatives to perform optimization on self-consistent integral equations problems. In Corliss et al. [136], chapter 22, pages 189–195.
108. G. Chavent. On the theory and practice of non-linear least squares. *Advances in Water Resources*, 14(2):55–63, 1991.
109. G. Chavent, J. Jaffré, S. Jégou, and J. Liu. A symbolic code generator for parameter estimation. In Berz et al. [42], chapter 11, pages 129–136.
110. B. Christianson. Reverse accumulation and accurate rounding error estimates for Taylor series coefficients. *Optimization Methods and Software*, 1(1):81–94, 1991.
111. B. Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3:311–326, 1994.
112. B. Christianson. Reverse accumulation and implicit functions. *Optimization Methods and Software*, 9(4):307–322, 1998.
113. B. Christianson. Cheap Newton steps for optimal control problems: Automatic differentiation and Pantoja algorithm. *Optimization Methods and Software*, 10:729–743, 1999.
114. B. Christianson. A self-stabilizing Pantoja-like indirect algorithm for optimal control. *Optimization Methods and Software*, 16:131–149, 2001.
115. B. Christianson and M. C. Bartholomew-Biggs. Globalization of Pantoja's optimal control algorithm. In Corliss et al. [136], chapter 13, pages 125–130.
116. B. Christianson and M. Cox. Automatic propagation of uncertainties. In Bücker et al. [78], pages 47–58.
117. B. Christianson, L. C. W. Dixon, and S. Brown. Sharing storage using dirty vectors. In Berz et al. [42], chapter 9, pages 107–115.
118. C. E. Christoffersen. State variable harmonic balance simulation of a quasi-optical power combining system. Master's Thesis, North Carolina State University, Raleigh, NC, 1998.
119. C. E. Christoffersen. *Global Modeling Of Nonlinear Microwave Circuits*. Ph.D Dissertation, North Carolina State University, Raleigh, NC, 2000.
120. C. E. Christoffersen, U. A. Mughal, and M. B. Steer. Object oriented microwave circuit simulation. *International Journal of RF and Microwave Computer-Aided Engineering*, 10(3):164–182, 2000.

121. C. E. Christoffersen, M. B. Steer, and M. A. Summers. Harmonic balance analysis for systems with circuit-field iterations. In *1998 IEEE MTT-S International Microwave Symposium Digest*, volume 2, pages 1131–1134. IEEE MTT-S, IEEE Press, June 1998.
122. C. E. Christoffersen, S. Velu, and M. B. Steer. A universal parameterized nonlinear device model formulation for microwave circuit simulation. In *2002 IEEE MTT-S International Microwave Symposium Digest*, volume 3, pages 2189–2192. IEEE MTT-S, IEEE Press, June 2002.
123. L. O. Chua and P. M. Lin. *Computer-Aided Analysis of Electronic Circuits: Algorithms & Computational Techniques*. Prentice-Hall, Englewood Cliffs, NJ, 1975.
124. M. A. Cirit. The Meyer model revisited: Why is charge not conserved? *IEEE Transactions on Computer-Aided Design*, 8(10):1033–1037, 1989.
125. E. G. Cofman and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213, 1972.
126. M. Cohen, U. Naumann, and J. Riehme. Towards differentiation-enabled Fortran 95 compiler technology. In *Proceedings of the 18th ACM Symposium on Applied Computing, Melbourne, Florida, USA, March 9–12, 2003*, pages 143–147, New York, NY, 2003. ACM Press.
127. S. D. Cohen and A. C. Hindmarsh. CVODE, a stiff/nonstiff ODE solver in C. *Computers in Physics*, 10:138–143, 1996.
128. T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, 1983.
129. T. F. Coleman and A. Verma. Structure and efficient Jacobian calculation. In Berz et al. [42], chapter 13, pages 149–159.
130. T. F. Coleman and A. Verma. ADMIT-1: Automatic differentiation and MATLAB interface toolbox. *ACM Transactions on Mathematical Software*, 26(1):150–175, 2000.
131. M. J. Colemen, M. Garcia, K. Mombaur, and A. Ruina. Prediction of stable walking for a toy that cannot stand. *Physical Review E*, 64(2):022901, 2001.
132. G. F. Corliss. Automatic differentiation bibliography. In Griewank and Corliss [227], pages 331–353.
133. G. F. Corliss. Overloading point and interval Taylor operators. In Griewank and Corliss [227], chapter 14, pages 139–146.
134. G. F. Corliss. Bibliography of automatic differentiation. In Corliss et al. [136], pages 383–425.
135. G. F. Corliss and Y. F. Chang. Solving ordinary differential equations using Taylor series. *ACM Transactions on Mathematical Software*, 8(2):114–144, 1982.
136. G. F. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Computer and Information Science. Springer, New York, NY, 2002.
137. G. F. Corliss and L. B. Rall. Adaptive, self-validating numerical quadrature. *SIAM Journal on Statistical and Scientific Computing*, 8:831–847, 1987.
138. M. G. Cox, M. P. Dainton, and P. M. Harris. *Uncertainty and Statistical Modelling*. HMSO, 2001. Software Support for Metrology Best Practice Guide No. 6.
139. F. D. Crary. A versatile precompiler for nonstandard arithmetics. *ACM Transactions on Mathematical Software*, 5(2):204–217, 1979.

140. B. Creusillet and F. Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24(6):513–546, 1996.
141. A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse Jacobian matrices. *Journal of the Institute of Mathematics and Applications*, 13:117–119, 1974.
142. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
143. J. E. Dennis, Jr. and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, volume 16 of *Classics in Applied Mathematics*. SIAM, Philadelphia, PA, 1996.
144. C. A. Desoer and E. S. Kuh. *Basic Circuit Theory*. McGraw-Hill, New York, NY, 1969.
145. P. Deuflhard. Recent advances in multiple shooting techniques. In I. Gladwell and D. K. Sayers, editors, *Computational Techniques for Ordinary Differential Equations*, pages 217–272. Academic Press, New York, NY, 1980.
146. K. W. Deutsch. *Nationalism and Social Communications*. MIT Press, Cambridge, MA, 2nd revised edition, 1966.
147. Device Group, University of California, Berkeley. BSIM homepage. `http://www-device.eecs.berkeley.edu/~bsim3`, 2004.
148. H. J. Diekhoff, P. Lory, H. J. Oberle, H. J. Pesch, R. Rentrop, and R. Seydel. Comparing routines for the numerical solution of initial value problems of ordinary differential equations in multiple shooting. *Numerische Mathematik*, 27(4):449–469, 1977.
149. F. Dignath, P. Eberhard, and A. Fritz. Analytical aspects and practical pitfalls in technical applications of AD. In Corliss et al. [136], chapter 14, pages 131–136.
150. L. C. W. Dixon. Use of automatic differentiation for calculating Hessians and Newton steps. In Griewank and Corliss [227], chapter 12, pages 114–125.
151. E. J. Doedel, H. B. Keller, and J. P. Kernevez. Numerical analysis and control of bifurcation problems, Part I. *International Journal of Bifurcation and Chaos*, 1(3):493–520, 1991.
152. E. J. Doedel, H. B. Keller, and J. P. Kernevez. Numerical analysis and control of bifurcation problems, Part II. *International Journal of Bifurcation and Chaos*, 1(4):745–772, 1991.
153. D. Dougherty. *sed & awk*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.
154. I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, 1989.
155. P. Eberhard. Adjoint variable method for the sensitivity analysis of multibody systems interpreted as continuous, hybrid form of automatic differentiation. In Berz et al. [42], chapter 28, pages 319–328.
156. D. Elizondo, C. Faure, and B. Cappelaere. Automatic– versus manual– differentiation for non-linear inverse modeling. Rapport de recherche 3981, INRIA, Sophia Antipolis, July 2000.
157. B. Erdélyi and M. Berz. Optimal symplectic approximation of Hamiltonian flows. *Physical Review Letters*, 87,11:114302, 2001.
158. R. Errico. What is an adjoint model. *Bulletin of the American Meteorological Society*, 78(11):2577–2591, 1997.

159. R. M. Errico and T. Vukicevic. Sensitivity analysis using an adjoint of the PSU-NCAR mesoscale model. *Monthly Weather Review*, 120:1644–1660, 1992.
160. V. Estupina-Borrell, M. M. Maubourguet, J. Chorda, M. Alquier, and D. Dartus. Use of direct simulation with space technology for flash flood events analysis. *Ecosystems and Floods, Hanoï*, pages 72–87, 2000.
161. Y. G. Evtushenko. Automatic differentiation viewed from optimal control theory. In Griewank and Corliss [227], chapter 3, pages 25–30.
162. Y. G. Evtushenko, E. S. Zasuhina, and V. I. Zubov. FAD method to compute second order derivatives. In Corliss et al. [136], chapter 39, pages 327–333.
163. FastOpt. *Transformation of Algorithms in Fortran, Manual, Draft Version, TAF Version 1.6*, Nov. 2003.
164. FastOpt. TAF homepage. `http://www.FastOpt.com`, 2005.
165. C. Faure. Splitting of algebraic expressions for automatic differentiation. In Berz et al. [42], chapter 10, pages 117–127.
166. C. Faure and P. Dutto. Extension of Odyssée to the MPI library — Direct mode. Rapport de recherche 3715, INRIA, Sophia Antipolis, June 1999.
167. C. Faure and P. Dutto. Extension of Odyssée to the MPI library — Reverse mode. Rapport de recherche 3774, INRIA, Sophia Antipolis, Oct. 1999.
168. C. Faure and U. Naumann. Minimizing the tape size. In Corliss et al. [136], chapter 34, pages 293–298.
169. W. F. Feehery and P. I. Barton. A differentiation-based approach to dynamic simulation and optimization with high-index differential-algebraic equations. In Berz et al. [42], chapter 21, pages 239–252.
170. W. F. Feehery, J. E. Tolsma, and P. I. Barton. Efficient sensitivity analysis of large-scale differential-algebraic systems. *Applied Numerical Mathematics*, 25:41–54, 1997.
171. U. Feige and C. Lund. On the hardness of computing the permanent of random matrices. In *Proceedings of the 24th annual ACM Symposium on the Theory of Computing (STOC 1992), Victoria, British Columbia, Canada, May 4–6, 1992*, pages 643–654, New York, NY, 1992. ACM Press.
172. E. A. Feigenbaum and J. Feldman, editors. *Computers and Thought*. McGraw-Hill, New York, NY, 1963.
173. H. Fischer. Special problems in automatic differentiation. In Griewank and Corliss [227], chapter 5, pages 43–50.
174. H. Fischer, B. Riedmüller, and S. Schäffler, editors. *Applied Mathematics and Parallel Computation*. Physica-Verlag, Berlin, 1996.
175. H. Flanders. Automatic differentiation of composite functions. In Griewank and Corliss [227], chapter 10, pages 95–99.
176. H. Flanders. Application of AD to a family of periodic functions. In Corliss et al. [136], chapter 38, pages 319–326.
177. H. Flanders. Solutions of ODEs with removable singularities. In Bücker et al. [78], pages 35–45.
178. S. A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. Technical Report AMOR Report 2004/03, Cranfield University, 2004.
179. S. A. Forth and T. P. Evans. Aerofoil optimisation via AD of a multigrid cell-vertex Euler flow solver. In Corliss et al. [136], chapter 17, pages 153–160.
180. S. A. Forth and R. Ketzscher. High-level interfaces for the MAD (MATLAB automatic differentiation) package. In P. Neittaanmäki, T. Rossi, S. Korotov,

E. Oñate, J. Périaux, and D. Knörzer, editors, *ECCOMAS 2004: Fourth European Congress on Computational Methods in Applied Sciences and Engineering*, volume 2. European Community on Computational Methods in Applied Sciences, 2004.

181. S. A. Forth and R. Ketzscher. MAD, MATLAB automatic differentiation package. `http://tomlab.biz/products/mad`, 2005.
182. S. A. Forth, M. Tadjouddine, J. D. Pryce, and J. K. Reid. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM Transactions on Mathematical Software*, 30(3):266–299, 2004.
183. W. R. Foster, L. H. Ungar, and J. S. Schwaber. Significance of conductances in Hodgkin-Huxley models. *Journal of Neurophysiology*, 70(6):2502–2518, 1993.
184. R. Fourer, D. M. Gay, and B. Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, 1990.
185. R. Fourer, D. M. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, Belmont, CA, 2nd edition, 2003.
186. L. Freitag, P. Knupp, T. Munson, and S. Shontz. A comparison of optimization software for mesh shape-quality improvement problems. In *Proceedings of the 11th International Meshing Roundtable, Ithaca, NY, September 15-18, 2002*, Albuquerque, NM, 2002. Sandia National Laboratories.
187. L. Freitag Diachin, P. Knupp, T. Munson, and S. Shontz. A comparison of inexact Newton and coordinate descent mesh optimization techniques. In *Proceedings of the 13th International Meshing Roundtable, Williamsburg, VA, September 19–22, 2004*, Albuquerque, NM, 2004. Sandia National Laboratories.
188. P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press, Piscataway, NJ, 2004.
189. M. Fürer. Approximating permanents of complex matrices. In *Proceedings of the 32nd annual ACM Symposium on the Theory of Computing (STOC 2000), Portland, OR, USA, May 21–23, 2000*, pages 667–669, New York, NY, 2000. ACM Press.
190. H. N. Gabow and R. E. Tarjan. A linear algorithm for a special case of disjoint set union. In *Proceedings of the 15th annual ACM Symposium on the Theory of Computing (STOC 1983), Boston, MA, USA, April 25–27, 1983*, pages 57–66, New York, NY, 1983. ACM Press.
191. E. Galanti, E. Tziperman, M. Harrison, A. Rosati, R. Giering, and Z. Sirkes. The equatorial thermocline outcropping — a seasonal control on the tropical pacific ocean-atmosphere instability. *Journal of Climate*, 15(19):2721–2739, 2002.
192. J. Gao, M. Xue, Z. Wang, and K. K. Droegemeier. The initial condition and explicit prediction of convection using ARPS adjoint and other retrievals methods with WSR-88D data. In *Proceedings of 12th Conference on Numerical Weather Prediction*, pages 176–178, Phoenix, AZ, 1998. American Meteorological Society.
193. M. Garcia. Disc foot walker simulation code. Private communication, 1998.
194. M. S. Garcia. *Stability, Scaling, and Chaos in Passive-Dynamic Gait Models.* PhD thesis, Cornell University, Ithaca, NY, 1999.
195. O. García. A system for the differentiation of Fortran code and an application to parameter estimation in forest growth models. In Griewank and Corliss [227], chapter 27, pages 273–285.

196. D. M. Gay. Automatic differentiation of nonlinear AMPL models. In Griewank and Corliss [227], chapter 7, pages 61–73.
197. D. M. Gay. Hooking your solver to AMPL. Numerical Analysis Manuscript No. 93–10, AT&T Bell Laboratories, Murray Hill, NJ, 1993, revised 1997.
198. D. M. Gay. More AD of nonlinear AMPL models: Computing Hessian information and exploiting partial separability. In Berz et al. [42], chapter 15, pages 173–184.
199. D. M. Gay. Semiautomatic differentiation for efficient gradient computations. In Bücker et al. [78], pages 147–158.
200. D. M. Gay. *RAD* package for reverse AD. `http://endo.sandia.gov/~dmgay/rad.tar.gz`, 2005.
201. U. Geitner, J. Utke, and A. Griewank. Automatic computation of sparse Jacobians by applying the method of Newsam and Ramsdell. In Berz et al. [42], chapter 14, pages 161–172.
202. N. Gelfand. Reshimming the Tevatron dipoles. Technical Report Beams-doc-1290-v1, Fermi National Accelerator Laboratory, Batavia, IL, Aug. 2004.
203. J. A. George and J. W. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal of Numerical Analysis*, 15:345–363, 1978.
204. R. Giering. *Tangent Linear and Adjoint Model Compiler, Users Manual*. Center for Global Change Sciences, Department of Earth, Atmospheric, and Planetary Science, MIT, Cambridge, MA, Dec. 1997. Unpublished.
205. R. Giering. *Tangent linear and Adjoint Model Compiler: Users Manual 1.4*, 1999. `http://www.autodiff.com/tamc`.
206. R. Giering and T. Kaminski. Recipes for adjoint code construction. Technical Report 212, Max-Planck-Institut für Meteorologie, Hamburg, Germany, 1996.
207. R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24(4):437–474, 1998.
208. R. Giering and T. Kaminski. Generating recomputations in reverse mode AD. In Corliss et al. [136], chapter 33, pages 283–291.
209. R. Giering and T. Kaminski. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. *Proceedings in Applied Mathematics and Mechanics*, 2(1):54–57, 2003.
210. R. Giering, T. Kaminski, and T. Slawig. Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil. *Future Generation Computer Systems*, 21(8):1345–1355, 2005.
211. R. Giering, T. Kaminski, R. Todling, R. Errico, R. Gelaro, and N. Winslow. Tangent linear and adjoint versions of NASA/GMAO's Fortran 90 global weather forecast model. In Bücker et al. [78], pages 273–282.
212. J.-C. Gilbert and C. Lemaréchal. Some numerical experiments with variable storage quasi-Newton algorithms. *Mathematical Programming, Ser. A*, 45(1–3):407–435, 1989.
213. M. B. Giles. On the iterative solution of adjoint equations. In Corliss et al. [136], chapter 16, pages 145–151.
214. P. E. Gill, W. Murray, and M. A. Saunders. SNOPT: An SQP algorithm for large-scale constrained optimization. *SIAM Journal on Optimization*, 12:979–1006, 2002.
215. M. S. Gockenbach, D. R. Reynolds, and W. W. Symes. Automatic differentiation and the adjoint state method. In Corliss et al. [136], chapter 18, pages 161–166.

216. S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM, Philadelphia, PA, 2001.
217. D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, Reading, MA, 1989.
218. D. Goldfarb and P. L. Toint. Optimal estimation of Jacobian and Hessian matrices that arise in finite difference calculations. *Mathematics of Computation*, 43(167):69–88, 1984.
219. V. V. Goldman and G. Cats. Automatic adjoint modeling within a program generation framework: A case study for a weather forecasting grid-point model. In Berz et al. [42], chapter 16, pages 185–194.
220. V. V. Goldman, J. A. van Hulzen, and J. H. J. Molenkamp. Efficient numerical program generation and computer algebra environments. In Griewank and Corliss [227], chapter 8, pages 74–83.
221. H. Goldstein. *Classical Mechanics*. Addison-Wesley, Reading, MA, 2nd edition, 1980.
222. G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.
223. A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming*, pages 83–107. Kluwer Academic Publishers, Amsterdam, The Netherlands, 1989.
224. A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.
225. A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, PA, 2000.
226. A. Griewank. A mathematical view of automatic differentiation. In *Acta Numerica*, volume 12, pages 321–398. Cambridge University Press, 2003.
227. A. Griewank and G. F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA, 1991.
228. A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, and A. Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Institute of Scientific Computing, Technical University Dresden, Mar. 1999.
229. A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.
230. A. Griewank and C. Mitev. Detecting Jacobian sparsity patterns by Bayesian probing. *Mathematical Programming, Ser. A*, 93(1):1–25, 2002.
231. A. Griewank and C. Mitev. Verifying Jacobian sparsity. In Corliss et al. [136], chapter 32, pages 271–279.
232. A. Griewank and S. Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Griewank and Corliss [227], chapter 13, pages 126–135.
233. A. Griewank and A. Verma. The Newsam-Ramsdell-Lagrange method for calculating sparse Jacobians. Technical Report IOKOMO–07–2001, TU Dresden, 2001.
234. A. Griewank and O. Vogel. Analysis and exploitation of Jacobian scarcity. In H. Bock, E. Kostina, H. Phu, and R. Rannacher, editors, *Modelling, Simulation*

and Optimization of Complex Processes, pages 149–164, New York, NY, 2004. Springer.
235. A. Griewank and A. Walther. On constrained optimization by adjoint based quasi-Newton methods. *Optimization Methods and Software*, 17:869–889, 2002.
236. A. K. Griffith and N. K. Nichols. Accounting for model error in data assimilation using adjoint models. In Berz et al. [42], chapter 17, pages 195–204.
237. J. Grimm. Complexity analysis of automatic differentiation in the hyperion software. In Corliss et al. [136], chapter 36, pages 305–310.
238. J. Grimm, L. Pottier, and N. Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In Berz et al. [42], chapter 8, pages 95–106.
239. W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Improving the performance of sparse matrix-vector multiplication by blocking. Talk by D. Keyes at SIAM Annual Meeting, July 2000. `http://www-fp.mcs.anl.gov/petsc-fun3d/Talks/multivec_siam00_1.pdf`.
240. W. D. Gropp, E. Lusk, and A. Skjellum. *Using MPI — Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, 1994.
241. J. Grote, M. Berz, and K. Makino. High-order representation of Poincaré maps. In Bücker et al. [78], pages 59–66.
242. D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen. *Modern Compiler Design*. Wiley & Sons, Chichester, UK, 2000.
243. J. Guckenheimer. Computer proofs for bifurcations of planar dynamical systems. In Berz et al. [42], chapter 20, pages 229–237.
244. J. Guckenheimer and B. Meloon. Computing periodic orbits and their bifurcations with automatic differentiation. *SIAM Journal on Scientific Computing*, 22(3):951–985, 2000.
245. GUM. *Guide to the Expression of Uncertainty in Measurement*. International Organisation for Standardisation, Geneva, 2nd edition, 1995.
246. GUM-SUP. *Guide to the Expression of Uncertainty in Measurement, Supplement 1: Numerical Methods for the Propagation of Probability Distributions*. International Organisation for Standardisation, 2004. Draft Technical Report by the Joint Committee for Guides in Metrology.
247. G. Haase, U. Langer, E. Lindner, and W. Mühlhuber. Optimal sizing of industrial structural mechanics problems using AD. In Corliss et al. [136], chapter 21, pages 181–188.
248. S. Hague and U. Naumann. Present and future scientific computation environments. In Corliss et al. [136], chapter 5, pages 59–66.
249. E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II*. Number 14 in Springer Series in Computational Mathematics. Springer, Berlin, 2nd edition, 1996.
250. D. Halliday and R. Resnick. *Fundamentals of Physics*. John Wiley & Sons, New York, NY, 2nd edition, 1981.
251. C. R. Hardnett, R. M. Rabbah, K. V. Palem, and W. F. Wong. Cache sensitive instruction scheduling. Technical Report CREST–TR–01–003, GIT–CC–01–15, Center for Research in Embedded Systems and Technologies, Georgia Institute of Technology, June 2001.
252. F. P. Hart, N. Kriplani, S. R. Luniya, C. E. Christoffersen, and M. B. Steer. Streamlined circuit device model development with fREEDA® and ADOL-C. In Bücker et al. [78], pages 293–305.

253. L. Hascoët and M. Araya-Polo. The adjoint data-flow analyses: Formalization, properties, and applications. In Bücker et al. [78], pages 135–146.
254. L. Hascoët, S. Fidanova, and C. Held. Adjoining independent computations. In Corliss et al. [136], chapter 35, pages 299–304.
255. L. Hascoët, R.-M. Greborio, and V. Pascual. Computing adjoints by automatic differentiation with TAPENADE. In B. Sportisse and F.-X. Le Dimet, editors, *École INRIA-CEA-EDF "Problèmes non-linéaires appliqués"*. Springer, 2005. To appear.
256. L. Hascoët, U. Naumann, and V. Pascual. TBR analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8), 2005.
257. L. Hascoët and V. Pascual. TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis, 2004.
258. L. Hascoët, V. Pascual, and R.-M. Greborio. The TAPENADE AD tool, tropics project, INRIA Sophia-Antipolis. Talk at AD Workshop, Cranfield, June 5–6, 2003.
259. E. Hassold and A. Galligo. Automatic differentiation applied to convex optimization. In Berz et al. [42], chapter 25, pages 287–296.
260. M. D. Hebden. An algorithm for minimization using exact second derivatives. Technical Report TP515, Atomic Energy Research Establishment, Harwell, England, 1973.
261. P. Heimbach, C. Hill, and R. Giering. Automatic generation of efficient adjoint code for a parallel Navier-Stokes solver. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 1019–1028, Berlin, 2002. Springer.
262. P. Heimbach, C. Hill, and R. Giering. An efficient exact adjoint of the parallel MIT General Circulation Model, generated via automatic differentiation. *Future Generation Computer Systems*, 21(8):1356–1371, 2005.
263. J. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5:422–448, 1983.
264. C. W. Ho, A. E. Ruehli, and P. A. Brennan. The modified nodal approach to network analysis. *IEEE Transactions on Circuits and Systems*, CAS-22(6):504–509, 1975.
265. A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117:500–544, 1952.
266. J. Hoefkens, M. Berz, and K. Makino. Efficient high-order methods for ODEs and DAEs. In Corliss et al. [136], chapter 41, pages 343–348.
267. D. Hömberg and S. Volkwein. Control of laser surface hardering by a reduced-order approach using proper orthogonal decomposition. *Mathematical and Computer Modelling*, 38:1003–1028, 2003.
268. G. M. Hornberger and R. C. Spear. An approach to the preliminary analysis of environmental systems. *Journal of Environmental Management*, 12:7–18, 1981.
269. J. E. Horwedel. GRESS, a preprocessor for sensitivity studies of Fortran programs. In Griewank and Corliss [227], chapter 24, pages 243–250.
270. S. Hossain and T. Steihaug. Reducing the number of AD passes for computing a sparse Jacobian matrix. In Corliss et al. [136], chapter 31, pages 263–270.

271. S. Hossain and T. Steihaug. Sparsity issues in the computation of Jacobian matrices. In T. Mora, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computing (ISSAC)*, pages 123–130, New York, NY, 2002. ACM.
272. S. Hossain and T. Steihaug. Optimal direct determination of sparse Jacobian matrices. Technical Report 254, Department of Informatics, University of Bergen, Norway, Oct. 2003. Revised version to appear in *Optimization Methods and Software*.
273. S. Hossain and T. Steihaug. Computing sparse Jacobian matrices optimally. In Bücker et al. [78], pages 77–87.
274. S. Houweling, F. Dentener, and J. Lelieveld. The impact of nonmethane hydrocarbon compounds on tropospheric photochemistry. *Journal of Geophysical Research*, 103(D9):10673–10696, 1998.
275. P. D. Hovland, C. H. Bischof, D. Spiegelman, and M. Casella. Efficient derivative codes through automatic differentiation and interface contraction: An application in biostatistics. *SIAM Journal on Scientific Computing*, 18(4):1056–1066, 1997.
276. P. D. Hovland, U. Naumann, and B. Norris. An XML-based platform for semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications*, pages 530–538, Anaheim, CA, 2002. ACTA Press.
277. M. J. Huiskes. Automatic differentiation for modern nonlinear regression. In Corliss et al. [136], chapter 8, pages 83–90.
278. U. Hutschenreiter. A new method for bevel gear tooth flank computation. In Berz et al. [42], chapter 29, pages 329–341.
279. INRIA Tropics Project. TAPENADE 2.0. `http://www-sop.inria.fr/tropics`, 2005.
280. M. Iri. Simultaneous computation of functions, partial derivatives, and estimates of rounding errors — complexity and practicality. *Japan Journal of Applied Mathematics*, 1:223–252, 1984.
281. M. Iri. History of automatic differentiation and rounding error estimation. In Griewank and Corliss [227], chapter 1, pages 3–16.
282. D. Jacobson and D. Mayne. *Differential Dynamic Programming*. Elsevier, New York, NY, 1970.
283. W. Jang. EKV.DOC. See the Mosnekv .cc and .h files in a fREEDA® installation, Apr. 2003.
284. K.-W. Jee, D. L. McShan, and B. A. Fraass. Implementation of automatic differentiation tools for multicriteria IMRT optimization. In Bücker et al. [78], pages 225–234.
285. M. E. Jerrell. Estimation of econometric functions. In Berz et al. [42], chapter 23, pages 265–272.
286. M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with non-negative entries. In *Proceedings of the 33rd Annual ACM Symposium on the Theory of Computing (STOC 1991), Hersonissos, Greece, July 6–8, 1991*, pages 712–721, New York, NY, 1991. ACM Press.
287. T. C. Johns, R. E. Carnell, J. F. Crossley, J. M. Gregory, J. F. B. Mitchell, C. A. Senior, S. F. B. Tett, and R. A. Wood. The second Hadley Centre coupled ocean-atmosphere GCM: Model description, spinup and validation. *Climate Dynamics*, 13(2):103–134, 1997.

288. D. W. Juedes. A taxonomy of automatic differentiation tools. In Griewank and Corliss [227], chapter 31, pages 315–329.
289. D. W. Juedes and K. Balakrishnan. Generalized neural networks, computational differentiation, and evolution. In Berz et al. [42], chapter 24, pages 273–285.
290. M. M. Junge and T. W. N. Haine. Mechanisms of North Atlantic wintertime sea surface temperature anomalies. *Journal of Climate*, 14:4560–4572, 2001.
291. K. Kabaya and M. Iri. Sum of uniformly distributed random variables and a family of nonanalytic C^∞-functions. *Japan Journal of Applied Mathematics*, 4(1):1–22, 1987.
292. D. Kalman and R. Lindell. Automatic differentiation in astrodynamical modeling. In Griewank and Corliss [227], chapter 23, pages 228–239.
293. T. Kaminski, R. Giering, M. Scholze, P. Rayner, and W. Knorr. An example of an automatic differentiation-based modelling system. In V. Kumar, L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *Computational Science – ICCSA 2003, International Conference Montreal, Canada, May 2003, Proceedings, Part II*, volume 2668 of *Lecture Notes in Computer Science*, pages 95–104, Berlin, 2003. Springer.
294. T. Kaminski, R. Giering, and M. Voßbeck. Efficient sensitivities for the spin-up phase. In Bücker et al. [78], pages 283–291.
295. E. W. Kaucher and W. L. Miranker. *Self-Validating Numerics for Function Space Problems*. Academic Press, New York, NY, 1984.
296. A. J. Keane. Passive vibration control via unusual geometries: The application of genetic algorithm optimization to structural design. *Journal of Sound and Vibration*, 185(3):441–453, 1995.
297. A. J. Keane. *The OPTIONS Design Exploration System User Guide and Reference Manual*. Computational Engineering and Design Group, School of Engineering Sciences, University of Southampton, Southampton SO17 1BJ, UK, 2001. `http://www.soton.ac.uk/~ajk/options/welcome.html`.
298. A. J. Keane and S. M. Brown. The design of a satellite boom with enhanced vibration performance using genetic algorithm techniques. In *Proceedings of ACEDC'96*, PEDC, University of Plymouth, UK, 1996.
299. B. Kearfott. *Rigorous Global Search: Continuous Problems*, volume 23 of *Nonconvex Optimization and its Applications*. Kluwer, Dordrecht, The Netherlands, 1996.
300. R. B. Kearfott. Automatic differentiation of conditional branches in an operator overloading context. In Berz et al. [42], chapter 6, pages 75–81.
301. R. B. Kearfott and A. Arazyan. Taylor models in deterministic global optimization. In Corliss et al. [136], chapter 44, pages 365–372.
302. G. Kedem. Automatic differentiation of computer programs. *ACM Transactions on Mathematical Software*, 6(2):150–165, 1980.
303. B. Keeping and C. C. Pantelides. WP4.1 Numerical Solvers Open Interface Specification Draft. Technical Report CO–NUMR–EL–03, CAPE-OPEN, 1999.
304. E. H. Kerner. Universal formats for nonlinear ordinary differential equations. *Journal of Mathematical Physics*, 22:1366–1371, 1981.
305. M. L. Kessler, D. L. McShan, M. A. Epelman, K. A. Vineberg, A. Eisbruch, T. S. Lawrence, and B. A. Fraass. Costlets: A generalized approach to cost functions for automated optimization of IMRT treatment plans. *Optimization and Engineering*, 6(4):421–448, 2005.

306. D. E. Keyes, P. D. Hovland, L. C. McInnes, and W. Samyono. Using automatic differentiation for second-order matrix-free methods in PDE-constrained optimization. In Corliss et al. [136], chapter 3, pages 35–50.
307. J. G. Kim and P. D. Hovland. Sensitivity analysis and parameter tuning of a sea-ice model. In Corliss et al. [136], chapter 9, pages 91–98.
308. W. Klein. Comparisons of different automatic differentiation tools in circuit simulation. In Berz et al. [42], chapter 26, pages 297–307.
309. W. Klein, A. Griewank, and A. Walther. Differentiation methods for industrial strength problems. In Corliss et al. [136], chapter 1, pages 3–23.
310. J. B. Klemp and R. B. Wilhelmson. The simulation of three-dimensional convective storm dynamics. *Journal of the Atmospheric Sciences*, 35(6):1070–1096, 1978.
311. W. Knorr. Satellitengestützte Fernerkundung und Modellierung des Globalen CO_2-Austauschs der Landvegetation: Eine Synthese. Examensarbeit 49, Max-Planck-Institut für Meteorologie, Hamburg, Germany, 1997.
312. W. Knorr. Annual and interannual CO_2 exchanges of the terrestrial biosphere: Process based simulations and uncertainties. *Global Ecology and Biogeography*, 9(3):225–252, 2000.
313. P. Knupp. Private communication, 2004.
314. D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, MA, 3rd edition, 1998.
315. N. M. Kriplani. Transistor modeling using advanced circuit simulator technology. Master's Thesis, North Carolina State University, Raleigh, NC, 2002.
316. K. Kubota. PADRE2, a Fortran precompiler yielding error estimates and second derivatives. In Griewank and Corliss [227], chapter 25, pages 251–262.
317. K. Kubota. Padre2 – Fortran precompiler for automatic differentiation and estimates of rounding errors. In Berz et al. [42], chapter 33, pages 367–374.
318. K. Kubota. A Fortran77 preprocessor for reverse mode automatic differentiation with recursive checkpointing. *Optimization Methods and Software*, 10(2):315–335, 1998.
319. K. Kubota. Computation of matrix permanent with automatic differentiation. Technical Report TRISE 04–02, Department of Information and System Engineering, Faculty of Science and Engineering, Chuo University, Tokyo, 2004.
320. K. Kubota. Computation of matrix permanent with automatic differentiation. In Bücker et al. [78], pages 67–76.
321. K. Kubota and M. Iri. Estimates of rounding errors with fast automatic differentiation and interval analysis. *Journal of Information Processing*, 14(4):508–515, 1991.
322. K. S. Kundert, J. K. White, and A. Sangiovanni-Vincentelli. *Steady-State Methods for Simulating Analog and Microwave Circuits*. Kluwer Academic Publishers, Boston, MA, 1990.
323. R. M. Law, P. J. Rayner, A. S. Denning, D. Erickson, M. Heimann, S. C. Piper, M. Ramonet, S. Taguchi, J. A. Taylor, C. M. Trudinger, and I. G. Watterson. Variations in modelled atmospheric transport of carbon dioxide and the consequences for CO_2 inversions. *Global Biogeochemical Cycles*, 10:783–796, 1996.
324. C. L. Lawson. Automatic differentiation of inverse functions. In Griewank and Corliss [227], chapter 9, pages 87–94.

325. J. D. Layne. Applying automatic differentiation and self-validation numerical methods in satellite simulations. In Griewank and Corliss [227], chapter 21, pages 211–217.
326. F.-X. Le Dimet, I. M. Navon, and D. N. Daescu. Second-order information in data assimilation. *Monthly Weather Review*, 130(3):629–648, 2002.
327. F.-X. Le Dimet and O. Talagrand. Variational algorithms for analysis and assimilation of meteorological observations: Theoretical aspects. *Tellus*, 38A:97–110, 1986.
328. V. Lebedev. Tevatron beam physics overview. Private communication, 2002.
329. V. Lebedev. Lattice measurements, injection errors, diagnostics. Talk at DOE Review, Fermi National Accelerator Laboratory, Batavia, IL, July 2003.
330. V. Lebedev. Overview of accelerator physics integration. Technical Report Beams-doc-1055-v1, Fermi National Accelerator Laboratory, Batavia, IL, May 2004.
331. S. L. Lee and P. D. Hovland. Sensitivity analysis using parallel ODE solvers and automatic differentiation in C: SensPVODE and ADIC. In Corliss et al. [136], chapter 26, pages 223–229.
332. R. Lehoucq, D. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, PA, 1997.
333. L. Lemaitre, C. McAndrew, and S. Hamm. ADMS – automatic device model synthesizer. In *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference*, volume 1, pages 27–30. IEEE Press, May 2002.
334. A. Leung, K. V. Palem, and C. Ungureanu. Run-time versus compile-time instruction scheduling in superscalar (RISC) processors: Performance and tradeoff. *Journal of Parallel and Distributed Computing*, 45(1):13–28, 1997.
335. S. Li and L. Petzold. Description of DASPKADJOINT: An adjoint sensitivity solver for differential-algebraic equations. Technical report, University of California, Department of Computer Science and Engineering, Santa Barbara, CA 93106, 2001.
336. Z. Li and I. M. Navon. Sensitivity analysis of outgoing radiation at the top of the atmosphere in the NCEP/MRF model. *Journal of Geophysical Research – Atmospheres*, 103(D4):3801–3814, 1998.
337. S.-J. Lin. A finite volume integration method for computing pressure gradient force in general vertical coordinates. *Quarterly Journal of the Royal Meteorological Society*, 123:1749–1762, 1997.
338. S.-J. Lin and R. B. Rood. Multidimensional flux-form semi-Lagrangian transport scheme. *Monthly Weather Review*, 124(9):2046–2070, 1996.
339. S.-J. Lin and R. B. Rood. An explicit flux-form semi-Lagrangian shallow-water model on the sphere. *Quarterly Journal of the Royal Meteorological Society*, 123:2477–2498, 1997.
340. N. Linial, A. Samorodnitsky, and A. Wigderson. A deterministic strongly polynomial algorithm for matrix scaling and approximate permanents. In *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing (STOC 1998), Dallas, TX, USA, May 24–26, 1998*, pages 644–652, New York, NY, 1998. ACM Press.
341. J.-L. Lions. *Optimal Control of Systems Governed by Partial Differential Equations*. Springer, Berlin, 1971.
342. I. Lira. *Evaluating the Measurement Uncertainty: Fundamentals and Practical Guidance*. Institute of Physics Publishing, Bristol, 2002.

343. R. J. Lohner. Enclosing the solutions of ordinary initial and boundary value problems. In E. W. Kaucher, U. W. Kulisch, and C. Ullrich, editors, *Computer Arithmetic: Scientific Computation and Programming Languages*, Series in Computer Science, pages 255–286. Wiley-Teubner, Stuttgart, 1987.
344. Y. Loukili and A. Soulaïmani. Numerical tracking of shallow water waves by the unstructured Finite Volume WAF approximation. *Journal of Computational Methods in Sciences and Engineering*, 2005. To appear.
345. S. R. Luniya, M. B. Steer, and C. E. Christoffersen. High dynamic range transient simulation of microwave circuits. In *IEEE Radio and Wireless Conference (RAWCON) 2004*. IEEE Press, 2004.
346. K. Makino and M. Berz. Remainder differential algebras and their applications. In Berz et al. [42], chapter 5, pages 63–74.
347. K. Makino and M. Berz. New applications of Taylor model methods. In Corliss et al. [136], chapter 43, pages 359–364.
348. K. Makino and M. Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 4,4:379–456, 2003.
349. K. Makino, M. Berz, C. J. Johnstone, and D. Errede. High order map treatment of superimposed cavities, absorbers, and magnetic multipole and solenoid fields. *Nuclear Instruments and Methods A*, 519:162–174, 2004.
350. S. Manabe and R. Stouffer. Two climate equilibria of a coupled ocean-atmosphere model. *Climate Dynamics*, 1:841–866, 1988.
351. M. Mancini. A parallel hierarchical approach for automatic differentiation. In Corliss et al. [136], chapter 27, pages 231–236.
352. S. Margulis and D. Entekhabi. A coupled land surface-boundary layer model and its adjoint. *Journal of Hydrometeorology*, 2(3):274–296, 2001.
353. H. M. Markowitz. The elimination form of the inverse and its application. *Management Science*, 3:257–269, 1957.
354. J. Marotzke, R. Giering, Q. K. Zhang, D. Stammer, C. N. Hill, and T. Lee. Construction of the adjoint MIT ocean general circulation model and application to Atlantic heat transport sensitivity. *Journal of Geophysical Research*, 104:29,529–29,548, 1999.
355. M. Martens. Notes on skew quadrupole fields in the Tevatron. Technical Report Beams-doc-485-v1, Fermi National Accelerator Laboratory, Batavia, IL, Mar. 2004.
356. Mathworks. *Statistics Toolbox User's Guide, Version 6.5*, June 2002. See `http://www.mathworks.com/access/helpdesk/help/toolbox/stats/`.
357. Mathworks. MATLAB homepage. `http://www.mathworks.com`, 2005.
358. H. Maurer and D. Augustin. Sensitivity analysis and real-time control of parametric optimal control problems using boundary value methods. In M. Grötschel, S. O. Krumke, and J. Rambau, editors, *Online Optimization of Large Scale Systems*, pages 17–56. Springer, 2001.
359. C. Mazauric. *Data Assimilation for Hydraulic Models. Parameters Estimation, Sensitivity Analysis and Domain Decomposition*. PhD thesis, Université Joseph Fourier, Grenoble, 2003. (In French).
360. V. Mazourik. Integration of automatic differentiation into application programs for PC's. In Griewank and Corliss [227], chapter 28, pages 286–293.
361. F. Mazzia and F. Iavernaro. The Bari test set for IVP solvers (release 2.2). `http://www.dm.uniba.it/~testset`, 2003.

362. W. J. McCalla and W. G. Howard. BIAS-3 — A program for the nonlinear DC analysis of bipolar transistor circuits. *IEEE Journal of Solid-State Circuits*, SC-6(1):14–19, 1971.
363. T. McGeer. Passive dynamic ciped catalogue. In R. Chatila, editor, *Proceedings of the 2nd International Symposium of Experimental Robotics*, pages 465–490, New York, NY, 1991. Springer.
364. B. Melville, P. Feldmann, and S. Moinian. A C++ based environment for analog circuit simulation. In *IEEE 1992 International Conference on Computer Design*, pages 516–519. IEEE Press, Oct. 1992.
365. M. Metcalf, J. Reid, and M. Cohen. *Fortran 95/2003 Explained*. Oxford University Press, Oxford, 2004.
366. L. Michelotti. MXYZPTLK: A C++ hacker's implementation of automatic differentiation. In Griewank and Corliss [227], chapter 22, pages 218–227.
367. K. Miettinen. *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers, Boston, MA, 1999.
368. J. Millman. *Microelectronics*. McGraw-Hill, New York, NY, 1979.
369. H. Minc. *Permanents*. Addison-Wesley, Reading, MA, 1978.
370. M. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, 1969.
371. Modelica Association. Modelica homepage. `http://www.modelica.org`, 2005.
372. B. Mohammadi, J.-M. Malé, and N. Rostaing-Schmidt. Automatic differentiation in direct and reverse modes: Application to optimum shapes design in fluid mechanics. In Berz et al. [42], chapter 27, pages 309–318.
373. R. Mohan, X. H. Wang, A. Jackson, T. Bortfeld, A. L. Boyer, G. J. Kutcher, S. A. Leibel, Z. Fuks, and C. C. Ling. The potential and limitations of the inverse radiotherapy technique. *Radiotherapy and Oncology*, 32:232–248, 1994.
374. M. Monagan and R. R. Rodoni. An implementation of the forward and reverse modes of automatic differentiation in Maple. In Berz et al. [42], chapter 31, pages 353–362.
375. R. E. Moore. *Interval Arithmetic and Automatic Error Analysis in Digital Computing*. PhD thesis, Department of Computer Science, Stanford University, 1962.
376. R. E. Moore. The automatic analysis and control of error in digital computation based on the use of interval numbers. In L. B. Rall, editor, *Error in Digital Computation*, volume 1, pages 61–130. Wiley, New York, NY, 1965.
377. R. E. Moore. Automatic local coordinate transformations to reduce the growth of error bounds in interval computation of solutions of ordinary differential equations. In L. B. Rall, editor, *Error in Digital Computation*, volume 2, pages 103–140. Wiley, New York, NY, 1965.
378. R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.
379. R. E. Moore. *Methods and Applications of Interval Analysis*. SIAM, Philadelphia, PA, 1979.
380. R. E. Moore, J. A. Davidson, H. R. Jaske, and S. Shayer. DIFEQ integration routine—user's manual. Technical Report LMSC 6–90–64–6, Lockheed Missiles and Space Co., Palo Alto, CA, 1964.
381. J. J. Moré. The Levenberg-Marquardt algorithm: Implementation and theory. In G. A. Watson, editor, *Numerical Analysis*, volume 630 of *Lecture Notes in Mathematics*, pages 105–116. Springer, New York, NY, 1977.
382. J. J. Moré. Automatic differentiation tools in optimization software. In Corliss et al. [136], chapter 2, pages 25–34.

383. D. D. Morrison, J. D. Riley, and J. F. Zancanaro. Multiple shooting method for 2-point boundary value problems. *Communications of the ACM*, 5(12):613–614, 1962.
384. M. Moshfreti-Torbati, A. J. Keane, S. J. Elliott, M. J. Brennan, and E. Rogers. The integration of advanced active and passive structural vibration control. In *Proceedings of VETOMAC-I*, Banglore, India, Oct. 2000.
385. R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling: (Technical summary). Technical Report TR 698, Courant Institute, New York University, New York, NY, July 1995.
386. M. Mu and J. Wang. A method for adjoint variational data assimilation with physical "on-off" processes. *Journal of the Atmospheric Sciences*, 60(16):2010–2018, 2003.
387. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.
388. NAG. Automatic differentiation: Differentiation enabled Fortran compiler technology. `http://www.nag.co.uk/nagware/research/ad_overview.asp`, 2005.
389. L. W. Nagel and R. A. Rohrer. SPICE2, A computer program to simulate semiconductor circuits. Technical Report ERL-M520, University of California at Berkeley, May 1975.
390. M. Nakamura, P. H. Stone, and J. Marotzke. Destabilization of the thermohaline circulation by atmospheric eddy transports. *Journal of Climate*, 7:1870–1882, 1994.
391. S. G. Nash. Newton-type minimization via the Lanczos method. *SIAM Journal on Numerical Analysis*, 21(4):770–788, 1984.
392. S. G. Nash. A survey of truncated-Newton methods. *Journal of Computational and Applied Mathematics*, 124:45–59, 2000.
393. S. G. Nash and A. Sofer. A barrier method for large-scale constrained optimization. *ORSA Journal on Computing*, 5:40–53, 1993.
394. U. Naumann. *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs*. PhD thesis, Technical University of Dresden, June 1999.
395. U. Naumann. Elimination techniques for cheap Jacobians. In Corliss et al. [136], chapter 29, pages 247–253.
396. U. Naumann. Optimal pivoting in tangent-linear and adjoint systems of nonlinear equations. Preprint ANL–MCS/P944–0402, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 2002.
397. U. Naumann. Reducing the memory requirement in reverse mode automatic differentiation by solving TBR flow equations. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 1039–1048, Berlin, 2002. Springer.
398. U. Naumann. Statement level optimality of tangent-linear and adjoint models. Preprint ANL–MCS/P1066–0603, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 2003.
399. U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Mathematical Programming, Ser. A*, 99(3):399–421, 2004.
400. U. Naumann and J. Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. In Bücker et al. [78], pages 159–169.

401. U. Naumann and J. Riehme. A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software*, 31(4), 2005.
402. U. Naumann, J. Utke, A. Lyons, and M. Fagan. Control flow reversal for adjoint code generation. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 55–64, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
403. U. Naumann, J. Utke, and A. Walther. An introduction to developing and using software tools for automatic differentiation. In P. Neittaanmäki, T. Rossi, S. Korotov, E. Oñate, J. Périaux, and D. Knörzer, editors, *ECCOMAS 2004: Fourth European Congress on Computational Methods in Applied Sciences and Engineering*, volume 2. European Community on Computational Methods in Applied Sciences, 2004.
404. W. Nauta and M. Feirtag. *Fundamental Neuroanatomy*. W. H. Freeman, San Francisco, CA, 1986.
405. I. M. Navon and X. Zou. Application of the adjoint model in meteorology. In Griewank and Corliss [227], chapter 20, pages 202–207.
406. N. S. Nedialkov and J. D. Pryce. An effective high-order interval method for validating existence and uniqueness of the solution of an IVP for an ODE. *Reliable Computing*, 7(6):449–465, 2001.
407. N. S. Nedialkov and J. D. Pryce. Solving differential-algebraic equations by Taylor series (I): Computing Taylor coefficients. *BIT*, 2005. To appear.
408. T. Nehrkorn, G. D. Modica, M. Cemiglia, F. H. Ruggiero, J. G. Michalakes, and X. Zou. MM5 adjoint development using TAMC: Experiences with an automatic code generator. In *Proceedings of 14th Conference on Numerical Weather Prediction*, pages 481–484. American Meteorological Society, 2001.
409. T. Nehrkorn, G. D. Modica, M. Cemiglia, F. H. Ruggiero, J. G. Michalakes, and X. Zou. 4DVAR development using an automatic code generator: Application to MM5v3. In *Proceedings of 12th PSU/NCAR Mesoscale Model Users' Workshop*, Boulder, CA, 2002. National Center for Atmospheric Research.
410. R. Neidinger. Directions for computing truncated multivariate Taylor series. *Mathematics of Computation*, 74(249):321–340, 2005.
411. A. Neumaier. *Introduction to Numerical Analysis*. Cambridge University Press, Cambridge, 2001.
412. G. N. Newsam and J. D. Ramsdell. Estimation of sparse Jacobian matrices. *SIAM Journal on Discrete Mathematics*, 4(3):404–417, 1983.
413. P. Ngnepieba, F.-X. Le Dimet, A. Boukong, and G. Nguetseng. Inverse problem formulation for parameters determination using the adjoint method. *ARIMA Journal — Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées*, 1:127–157, 2002.
414. N. Nielsen. *Die Gammafunktion*. Chelsea, New York, NY, 1965. Reprint of *Handbuch der Theorie der Gammafunction*, 1906.
415. F. Nusseibeh, T. W. Nuteson, J. Patwardhan, M. A. Summers, C. E. Christoffersen, J. Kreskovsky, and M. B. Steer. Computer-aided engineering environment for spatial power combining systems. In *1997 IEEE MTT-S International Microwave Symposium Digest*, volume 2, pages 1073–1076. IEEE MTT-S, IEEE Press, June 1997.
416. J. Ogrodzki. *Circuit Simulation Methods and Algorithms*. CRC Press, Boca Raton, FL, 1994.
417. Y. S. Ong and A. J. Keane. Meta-Lamarckian learning in memetic algorithms. *IEEE Transactions on Evolutionary Computing*, 8(2):99–109, 2004.

418. OpenMP Architecture Review Board. OpenMP Fortran application program interface, version 1.1. `http://www.openmp.org`, 1999.
419. OpenMP Architecture Review Board. OpenMP Fortran application program interface, version 2.0. `http://www.openmp.org`, 2000.
420. G. M. Ostrovskii, Y. M. Volin, and W. W. Borisov. Über die Berechnung von Ableitungen. *Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13:382–384, 1971.
421. A. Ostrowski. Neuer Beweis des Hölderschen Satzes, daß die Gammafunktion keiner algebraischen Differentialgleichung genügt. *Mathematische Annalen*, 79:286–288, 1919.
422. S. Ozaki, R. Palmer, M. Zisman, and J. Gallardo. Feasibility study-II of a muon-based neutrino source. Technical Report 52623, Muon Collider Collaboration, Brookhaven National Laboratory, Upton, NY, 2001.
423. D. B. Özyurt and P. I. Barton. Application of targeted automatic differentiation to large-scale dynamic optimization. In Bücker et al. [78], pages 235–247.
424. D. B. Özyurt and P. I. Barton. Cheap second order directional derivatives of stiff ODE embedded functionals. *SIAM Journal on Scientific Computing*, 26(5):1725–1743, 2005.
425. K. V. Palem and B. Simons. Scheduling time-critical instructions on RISC machines. *ACM Transactions on Programming Languages and Systems*, 15(4):632–658, 1993.
426. X. Pang and P. J. Werbos. Neural network design for J function approximation in dynamic programming. *Mathematical Modeling and Scientific Computing*, 5(2/3), 1996. Also available at `http://arxiv.org/abs/adap-org/9806001`.
427. J. Pantoja. Differential dynamic programming and Newton's method. *International Journal on Control*, 47:1539–1553, 1988.
428. S. K. Park, K. K. Droegemeier, and C. H. Bischof. Automatic differentiation as a tool for sensitivity analysis of a convective storm in a 3-D cloud model. In Berz et al. [42], chapter 18, pages 205–214.
429. V. Pascual and L. Hascoët. Extension of TAPENADE toward Fortran 95. In Bücker et al. [78], pages 171–179.
430. D. O. Pederson. A historical review of circuit simulation. *IEEE Transactions on Circuits and Systems*, CAS-31(1):103–111, 1984.
431. L. Petzold, J. B. Rosen, P. E. Gill, L. O. Jay, and K. Park. Numerical optimal control of parabolic PDEs using DASOPT. In L. T. Biegler, T. F. Coleman, A. R. Conn, and F. N. Santosa, editors, *Large Scale Optimization with Applications: Part II*. Springer, New York, NY, 1997.
432. E. Phipps, R. Casey, and J. Guckenheimer. Periodic orbits of hybrid systems and parameter estimation via AD. In Bücker et al. [78], pages 211–223.
433. E. T. Phipps. *Taylor Series Integration of Differential-Algebraic Equations: Automatic Differentiation as a Tool for Simulating Rigid Body Mechanical Systems*. PhD thesis, Cornell University, 2003.
434. L. T. Pillage, R. A. Rohrer, and C. Visweswariah. *Electronic Circuit and System Simulation Methods*. McGraw-Hill, New York, NY, 1995.
435. F. A. Potra and W. C. Rheinboldt. On the numerical solution of Euler-Lagrange equations. *Mechanics of Structures and Machines*, 19(1):1–18, 1991.
436. F. A. Potra and J. Yen. Implicit numerical integration for Euler-Lagrange equations via tangent space parameterization. *Mechanics of Structures and Machines*, 19(1):77–98, 1991.

437. C. Pottle. A "Textbook"computerized state-space network analysis algorithm. *IEEE Transactions on Circuit Theory*, 16(4):566–568, 1969.
438. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in Fortran*. Cambridge University Press, Cambridge, 2nd edition, 1992.
439. Prinn et al. Global average concentration and trend for hydroxil radical deduction from ALE/GAGE trichloroethane (methyl chloroform) data for 1987–1990. *Journal of Geophysical Research*, 97:2445–2461, 1992.
440. Process Systems Enterprise. gPROMS homepage. `http://www.psenterprise.com`, 2005.
441. J. D. Pryce. Solving high-index DAEs by Taylor series. *Numerical Algorithms*, 19:195–211, 1998.
442. J. D. Pryce. A simple structural analysis method for DAEs. *BIT*, 41(2):364–294, 2001.
443. J. D. Pryce and M. Tadjouddine. Cheap Jacobians by AD regarded as compact LU factorization, 2005. In preparation.
444. G. D. Pusch. Jet space as the geometric arena of automatic differentiation. In Berz et al. [42], chapter 4, pages 53–62.
445. G. D. Pusch, C. H. Bischof, and A. Carle. On automatic differentiation of codes with complex arithmetic with respect to real variables. Technical Memorandum ANL/MCS–TM–188, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1995.
446. A. Quarteroni, R. Sacco, and F. Saleri. *Numerical Mathematics*, volume 37 of *Texts in Applied Mathematics*. Springer, New York, NY, 2000.
447. F. Rabier, E. Klinker, J.-F. Mahfouf, and A. Simmons. The ECMWF operational implementation of four-dimensional variational assimilation, I: Experimental results with simplified physics. *Quarterly Journal of the Royal Meteorological Society*, 126(A):1143–1170, 2000.
448. H. Raiffa and R. Schlaifer. *Applied Statistical Decision Theory*. Wiley-Interscience, New York, 2000.
449. L. B. Rall. *Computational Solution of Nonlinear Operator Equations*. Wiley, New York, 1969.
450. L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, Berlin, 1981.
451. L. B. Rall. Differentiation in Pascal-SC: Type GRADIENT. *ACM Transactions on Mathematical Software*, 10(2):161–184, 1984.
452. L. B. Rall. The arithmetic of differentiation. *Mathematics Magazine*, 59:275–282, 1986.
453. L. B. Rall. Point and interval differentiation arithmetics. In Griewank and Corliss [227], chapter 2, pages 17–24.
454. L. B. Rall. Perspectives on automatic differentiation: Past, present, and future? In Bücker et al. [78], pages 1–14.
455. L. B. Rall and G. F. Corliss. An introduction to automatic differentiation. In Berz et al. [42], chapter 1, pages 1–18.
456. P. Rayner, R. Giering, T. Kaminski, R. Ménard, R. Todling, and C. Trudinger. Exercises. In P. Kasibhatla, M. Heimann, D. Hartley, P. J. Rayner, N. Mahowald, and R. Prinn, editors, *Inverse Methods in Global Biogeochemical Cycles, Geophys. Monogr. Ser.*, volume 114, pages 324–347. American Geophysical Union, Washington, D. C., 1999.

457. P. Rayner, M. Scholze, W. Knorr, T. Kaminski, R. Giering, and H. Widmann. Two decades of terrestrial carbon fluxes from a carbon cycle data assimilation system (CCDAS). *Global Biogeochemical Cycles*, 19:GB2026, 2005.
458. L. Reichel and G. Opfer. Chebyshev-Vandermonde Systems. *Mathematics of Computation*, 57(196):703–721, 1991.
459. T. W. Reps and L. B. Rall. Computational divided differencing and divided-difference arithmetics. *Higher-Order and Symbolic Computation*, 16:93–l19, 2003.
460. W. C. Rheinboldt. On the existence and computation of solutions of nonlinear semi-implicit algebraic equations. *Nonlinear Analysis: Theory, Methods & Applications*, 16(7–8):647–661, 1991.
461. J. R. Rice, editor. *Mathematical Software*. Academic Press, New York, NY, 1971.
462. J. Riehme and U. Naumann. Using the differentiation-enabled NAGWare Fortran 95 compiler – A guided tour. In P. Neittaanmäki, T. Rossi, S. Korotov, E. Oñate, J. Périaux, and D. Knörzer, editors, *ECCOMAS 2004: Fourth European Congress on Computational Methods in Applied Sciences and Engineering*, volume 2. European Community on Computational Methods in Applied Sciences, 2004.
463. K. Röbenack and K. J. Reinschke. Nonlinear observer design using automatic differentiation. In Corliss et al. [136], chapter 15, pages 137–142.
464. P. J. C. Rodrigues. *Computer-Aided Analysis of Nonlinear Microwave Circuits*. Artech House, Boston, MA, 1998.
465. P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981.
466. F. H. Ruggiero, G. D. Modica, T. Nehrkorn, M. Cemiglia, J. G. Michalakes, and X. Zou. MM5-based 4DVAR: Current status and future plans. In *Proceedings of 12th PSU/NCAR Mesoscale Model Users' Workshop*, Boulder, CA, 2002. National Center for Atmospheric Research.
467. S. M. Rump. INTLAB – INTerval LABoratory. In T. Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999.
468. S. Saarinen, R. Bramley, and G. Cybenko. Neural networks, backpropagation, and automatic differentiation. In Griewank and Corliss [227], chapter 4, pages 31–42.
469. S. Schlenkrich, A. Walther, and A. Griewank. Application of AD-based quasi-Newton-methods to stiff ODEs. In Bücker et al. [78], pages 89–98.
470. A. Schmittner and T. F. Stocker. A seasonally forced ocean-atmosphere model for paleoclimate studies. *Journal of Climate*, 14:1055–1068, 2001.
471. M. Scholze. Model studies on the response of the terrestrial carbon cycle on climate change and variability. Examensarbeit 90, Max-Planck-Institut für Meteorologie, Hamburg, Germany, 2003.
472. D. J. Seo, V. Koren, and N. Cajina. Real-time variational assimilation of hydrologic and hydrometeorological data into operational hydrologic forecasting. *Journal of Hydrometeorology*, 4(3):627–641, 2003.
473. P. Shah. Application of adjoint equations to estimation of parameters in distributed dynamic systems. In Griewank and Corliss [227], chapter 18, pages 181–190.
474. L. F. Shampine and A. Witt. A simple stepsize selection algorithm for ODE codes. *Journal of Computational and Applied Mathematics*, 58:345–354, 1995.

475. K. Shamseddine and M. Berz. Exception handling in derivative computation with nonarchimedean calculus. In Berz et al. [42], chapter 3, pages 37–51.
476. K. Shankar and A. J. Keane. Energy flow predictions in a structure of rigidly joined beams using receptance theory. *Journal of Sound and Vibration*, 185(5):867–890, 1995.
477. K. Shankar and A. J. Keane. A study of the vibrational energies of two coupled beams by finite element and Green function (receptance) methods. *Journal of Sound and Vibration*, 181(5):801–838, 1995.
478. D. Shiriaev. ADOL–F automatic differentiation of Fortran codes. In Berz et al. [42], chapter 34, pages 375–384.
479. J. Si, A. G. Barto, W. B. Powell, and D. Wunsch, editors. *Handbook of Learning and Approximate Dynamic Programming*. Wiley-IEEE Press, Piscataway, NJ, 2004.
480. Z. Sirkes and E. Tziperman. Finite difference of adjoint or adjoint of finite difference? *Monthly Weather Review*, 125(12):3373–3378, 1997.
481. S. Sitch, I. C. Prentice, B. Smith, A. Arneth, A. Bondeau, W. Cramer, J. O. Kaplan, S. Levis, W. Lucht, M. T. Sykes, K. Thonicke, and S. Venevsky. Evaluation of ecosystem dynamics, plant geography and terrestrial carbon cycling in the LPJ dynamic global vegetation model. *Global Change Biology*, 9:161–185, 2003.
482. P. Snopok, C. Johnstone, and M. Berz. Simulation and optimization of the Tevatron accelerator. In Bücker et al. [78], pages 199–209.
483. I. S. Sokolnikoff and E. S. Sokolnikoff. *Higher Mathematics for Engineers and Physicists*. McGraw-Hill, New York, NY, 2nd edition, 1941.
484. D. C. Sorensen. Newton's method with a model trust region modification. *SIAM Journal on Numerical Analysis*, 19:409–426, 1982.
485. E. J. Soulié. User's experience with FORTRAN compilers in least squares problems. In Griewank and Corliss [227], chapter 29, pages 297–306.
486. E. J. Soulié, C. Faure, T. Berclaz, and M. Geoffroy. Electron paramagnetic resonance, optimization and automatic differentiation. In Corliss et al. [136], chapter 10, pages 99–106.
487. B. Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Jan. 1980.
488. J. Sternberg. Reduction of storage requirement by checkpointing for time-dependent optimal control problems. PhD Thesis, in preparation.
489. J. Sternberg. Adaptive Umkehrschemata für Schrittfolgen mit nicht-uniformen Kosten. Diploma thesis, Technical University Dresden, 2002.
490. J. Sternberg and A. Griewank. Reduction of storage requirement by checkpointing for time-dependent optimal control problems in ODEs. In Bücker et al. [78], pages 99–110.
491. R. Steuer. *Multiple Criteria Optimization: Theory, Computation and Application*. Wiley, New York, NY, 1985.
492. B. G. Streetman. *Solid-State Electronic Devices*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 1980.
493. M. Syphers. Skew quadrupole tuning and vertical dispersion in the Tevatron. Technical Report Beams-doc-611-v1, Fermi National Accelerator Laboratory, Batavia, IL, June 2003.

494. M. Syphers. Estimate of skew quadrupole field in Tevatron dipoles due to creep. Technical Report Beams-doc-513-v1, Fermi National Accelerator Laboratory, Batavia, IL, Mar. 2004.
495. M. Syphers. Strong transverse coupling in the Tevatron. Technical Report Beams-doc-1159-v1, Fermi National Accelerator Laboratory, Batavia, IL, May 2004.
496. M. Syphers. Tevatron accelerator physics. Technical Report Beams-doc-1046-v1, Fermi National Accelerator Laboratory, Batavia, IL, Feb. 2004.
497. M. Tadjouddine, F. Bodman, J. D. Pryce, and S. A. Forth. Improving the performance of the vertex elimination algorithm for derivative calculation. In Bücker et al. [78], pages 111–120.
498. M. Tadjouddine, S. A. Forth, and A. J. Keane. Adjoint differentiation of a structural dynamics solver. In Bücker et al. [78], pages 307–317.
499. M. Tadjouddine, S. A. Forth, and J. D. Pryce. AD tools and prospects for optimal AD in CFD flux Jacobian calculations. In Corliss et al. [136], chapter 30, pages 255–261.
500. M. Tadjouddine, S. A. Forth, and J. D. Pryce. Hierarchical automatic differentiation by vertex elimination and source transformation. In V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, editors, *Computational Science and Its Applications – ICCSA 2003, Proceedings of the International Conference on Computational Science and its Applications, Montreal, Canada, May 18–21, 2003. Part II*, volume 2668 of *Lecture Notes in Computer Science*, pages 115–124, Berlin, 2003. Springer.
501. M. Tadjouddine, S. A. Forth, J. D. Pryce, and J. K. Reid. Performance issues for vertex elimination methods in computing Jacobians using automatic differentiation. In P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, editors, *Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, volume 2330 of *Lecture Notes in Computer Science*, pages 1077–1086, Berlin, 2002. Springer.
502. O. Talagrand. The use of adjoint equations in numerical modelling of the atmospheric circulation. In Griewank and Corliss [227], chapter 17, pages 169–180.
503. P. P. Tans. A note on isotopic ratios and the global atmospheric methane budget. *Global Biogeochemical Cycles*, 11:77–81, 1997.
504. R. K. Ten Haken, B. A. Fraass, A. S. Lichter, L. H. Marsh, E. H. Radany, and H. M. Sandler. A brain tumor dose escalation protocol based on effective dose equivalence to prior experience. *International Journal of Radiation Oncology, Biology, Physics*, 42:137–41, 1998.
505. J. Tennison. *Beginning XSLT*. WROX Press, Birmingham, 2002.
506. L. Tesfatsion. Automatic evaluation of higher-order partial derivatives for nonlocal sensitivity analysis. In Griewank and Corliss [227], chapter 16, pages 157–165.
507. W. C. Thacker. Automatic differentiation from an oceanographer's perspective. In Griewank and Corliss [227], chapter 19, pages 191–201.
508. J. M. Thames. Synthetic calculus — A paradigm of mathematical program synthesis. In Griewank and Corliss [227], chapter 26, pages 263–272.
509. E. Tijskens, H. Ramon, and J. De Baerdemaeker. Efficient operator overloading AD for solving nonlinear PDEs. In Corliss et al. [136], chapter 19, pages 167–172.

510. E. Toro. *Shock-Capturing Methods for Free-Surface Shallow Flows*. Wiley & Sons, Chichester, UK, 2001.
511. S. Turner, editor. *CAS, CERN Accelerator School: 5th General Accelerator Physics Course*, volume 1 of *CERN Accelerator School*, Geneva, Jan. 1994. CERN.
512. J. Utke. Efficient Newton steps without Jacobians. In Berz et al. [42], chapter 22, pages 253–264.
513. J. Utke. Flattening basic blocks. In Bücker et al. [78], pages 121–133.
514. J. Utke and U. Naumann. Software technological issues in automating the semantic transformation of numerical programs. In M. Hamza, editor, *Software Engineering and Applications, Proceedings of the Seventh IASTED International Conference*, pages 417–422, Anaheim, CA, 2003. ACTA Press.
515. L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.
516. G. J. van Oldenborgh, G. Burgers, S. Venzke, C. Eckert, and R. Giering. Tracking down the delayed ENSO oscillator with an adjoint OGCM. *Monthly Weather Review*, 127:1477–1495, 1999.
517. M. C. Vanier and J. M. Bower. A comparative survey of automated parameter-search methods for compartmental neural models. *Journal of Computational Neuroscience*, 7:149–171, 1999.
518. N. V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, NY, 2000.
519. V. S. Vassiliadis, E. B. Canto, and J. R. Banga. Second-order sensitivites of general dynamic system with application to optimal control problems. *Chemical Engineering Science*, 54:3851–3860, 1999.
520. A. Verma. *Structured Automatic Differentiation*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1998.
521. A. Verma. ADMAT: Automatic differentiation in MATLAB using object oriented methods. In M. E. Henderson, C. R. Anderson, and S. L. Lyons, editors, *Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop*, pages 174–183, Philadelphia, 1999. SIAM.
522. P. A. Vidard, A. Piacentini, and F.-X. Le Dimet. Variational data analysis with control of the forecast bias. *Tellus*, 56A(3):177–188, 2004.
523. J. Vlach and K. Singhal. *Computer Methods for Circuit Analysis and Design*. Van Nostrand Reinhold, New York, NY, 2nd edition, 1994.
524. O. Vogel. Accurate gear tooth contact and sensitivity computation for hypoid bevel gears. In Corliss et al. [136], chapter 23, pages 197–204.
525. L. von Wedel. CapeML – A model exchange language for chemical process modeling. Technical report, Lehrstuhl für Prozesstechnik, RWTH Aachen University, 2002.
526. A. Walther and A. Griewank. New results on program reversals. In Corliss et al. [136], chapter 28, pages 237–243.
527. Z. Wang, K. K. Droegemeier, M. Xue, S. K. Park, J. G. Michalakes, and X. Zou. Sensitivity analysis of a 3-D compressible storm-scale model to input parameters. In *Proceedings of the International Symposium on Assimilation of Observations in Meteorology and Oceanography*, pages 437–442, Tokyo, Japan, 1995. World Meteorological Organization.

528. Z. Wang, I. M. Navon, X. Zou, and F.-X. Le Dimet. A truncated Newton optimization algorithm in meteorology applications with analytic Hessian/vector products. *Computational Optimization and Applications*, 4:241–262, 1995.
529. S. Webb. *Intensity-Modulated Radiation Therapy*. Institute of Physics Publishing, Bristol, 2000.
530. R. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7:463–464, 1964.
531. P. J. Werbos. The elements of intelligence. *Cybernetica*, 3, 1968.
532. P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioural Sciences*. PhD thesis, Harvard University, 1974. Reprinted in [543].
533. P. J. Werbos. Changes in global policy analysis procedures suggested by new methods of optimization. *Policy Analysis and Information Systems*, 3(1):27–52, 1979.
534. P. J. Werbos. Applications of advances in nonlinear sensitivity analysis. In *Systems Modeling and Optimization*, pages 762–777, New York, 1982. Springer, NY. Reprinted in [543].
535. P. J. Werbos. Solving and optimizing complex systems: Lessons from the EIA long-term energy model. In B. Lev, editor, *Energy Models and Studies*. North Holland, Amsterdam, The Netherlands, 1983.
536. P. J. Werbos. Generalized information requirements of intelligent decision-making systems. In *SUGI 11 Proceedings*, Cary NC, 1986. SAS Institute.
537. P. J. Werbos. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, 17:7–20, 1987.
538. P. J. Werbos. Backpropagation: Past and future. In *Proceedings of the IEEE International Conference on Neural Networks (IJCNN88), San Diego, CA, 1988*, volume I, pages 343–353, Long Beach, CA, 1988. IEEE Press.
539. P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1:339–356, 1988.
540. P. J. Werbos. Maximizing long-term gas industry profits in two minutes in Lotus using neural network methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(2):315–333, 1989.
541. P. J. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. Reprinted in [543].
542. P. J. Werbos. Elastic fuzzy logic: A better fit to neurocontrol and true intelligence. *Journal of Intelligent and Fuzzy Systems*, 1:365–377, 1993. Reprinted and updated in Gupta, M. (ed.): *Intelligent Control*. IEEE Press, New York, NY, 1995.
543. P. J. Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley, New York, NY, 1994.
544. P. J. Werbos. Backpropagation: Basics and new developments. In M. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*. MIT Press, Cambridge, MA, 1st edition, 1995.
545. P. J. Werbos. A brain-like design to learn optimal decision strategies in complex environments. In M. Karny, K. Warwick, and V. Kurkova, editors, *A Neural Networks Approach*. Springer, London, 1998.
546. P. J. Werbos. Stable adaptive control using new critic designs, 1998. `http://arxiv.org/abs/adap-org/9810001`.

547. P. J. Werbos. Tutorial on neurocontrol, control theory and related techniques: From backpropagation to brain-like intelligent systems, 1999. Tutorial presented at 12th International Conference on Mathematical and Computer Modelling and Scientific Computing. `http://www.iamcm.org/publications.html`.
548. P. J. Werbos. What do neural nets and quantum theory tell us about mind and reality? In K.Yasue, M.Jibu, and T. D. Senta, editors, *No Matter, Never Mind: Proceedings of Toward a Science of Consciousness: Fundamental approaches, Tokyo 1999*, pages 63–87. John Benjamins, Amsterdam, The Netherlands, 2002.
549. P. J. Werbos. From the termite mound to the stars: Meditations on discussions with Ilya Prigogine. *Problems of Nonlinear Analysis in Engineering Systems*, 19(3), 2003. Also available at `http://arxiv.org/abs/q-bio.PE/0311007`.
550. P. J. Werbos. Backwards differentiation in AD and neural nets: Past links and new opportunities. In Bücker et al. [78], pages 15–34.
551. P. J. Werbos and X. Z. Pang. Generalized maze navigation: SRN critics solve what feedforward or Hebbian nets cannot. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC96), Beijing, China, October 10–14, 1996*, volume 3, pages 1764–1769. IEEE, 1996. An earlier version appeared in the WCNN96 Proceedings from INNS.
552. P. J. Werbos and J. Titus. An empirical test of new forecasting methods derived from a theory of intelligence: The predicition of conflict in Latin America. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-8:657–666, 1978.
553. D. A. White and D. A. Sofge, editors. *The Handbook of Intelligent Control*. Van Nostrand Reinhold, New York, NY, 1992.
554. L. W. White, B. Vieux, D. Armand, and F.-X. Le Dimet. Estimation of optimal parameters for a surface hydrology model. *Advances in Water Resources*, 26(3):337–348, 2003.
555. R. D. Wilkins. Investigation of a new analytical method for numerical derivative evaluation. *Communications of the ACM*, 7:465–471, 1964.
556. World Wide Web Consortium. XSL transformations (XSLT), version 1.0. `http://www.w3.org/TR/xslt`, 1999.
557. World Wide Web Consortium. W3C math homepage. `http://www.w3.org/Math`, 2001.
558. B. Worley. Experience with the forward and reverse mode of GRESS in contaminant transport modeling and other applications. In Griewank and Corliss [227], chapter 30, pages 307–314.
559. Q. Xu. Generalized adjoint for physical processes with parameterized discontinuities — Part I: Basic issues and heuristic examples. *Journal of the Atmospheric Sciences*, 53(8):1123–1142, 1996.
560. M. Xue, K. K. Droegemeier, and V. Wong. The Advanced Regional Prediction System (ARPS) — A multiscale nonhydrostatic atmospheric simulation and prediction tool. Part I: Model dynamics and verification. *Meteorology and Atmospheric Physics*, 75(3–4):161–193, 2000.
561. M. Xue, K. K. Droegemeier, V. Wong, A. Shapiro, K. Brewster, F. Carr, Y. Liu, and D. Wang. The Advanced Regional Prediction System (ARPS) — A multi-scale nonhydrostatic atmospheric simulation and prediction tool. Part II: Model physics and applications. *Meteorology and Atmospheric Physics*, 76(1–4):143–166, 2001.
562. M. Xue, D. Wang, J. Gao, K. Brewster, and K. K. Droegemeier. The Advanced Regional Prediction System (ARPS), storm-scale numerical weather prediction

and data assimilation. *Meteorology and Atmospheric Physics*, 82(1–4):139–170, 2003.
563. Y. X. M. Xue, W. Martin, and J. Gao. Development of an adjoint for a complex atmospheric model, the ARPS, using TAF. In Bücker et al. [78], pages 263–272.
564. J. Yang. *Variational Data Assimilation for the Problems of Sediment Transport in Rivers*. PhD thesis, Université Joseph Fourier, Grenoble, 1999. (In French).
565. P. Yang, B. D. Epler, and P. K. Chaterjee. An investigation of the charge conservation problem for MOSFET circuit simulation. *IEEE Journal of Solid-State Circuits*, SC-18(1):128–139, 1983.
566. W. Yang and G. F. Corliss. Bibliography of computational differentiation. In Berz et al. [42], pages 393–418.
567. M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Discrete Mathematics*, 2(1):77–79, 1981.
568. J. Yen. Constrained equations of motion in multibody dynamics as ODEs on manifolds. *SIAM Journal on Numerical Analysis*, 30(2):553–568, 1993.
569. Y. Q. Zhu, R. Todling, J. Guo, S. E. Cohn, I. M. Navon, and Y. Yang. The Geos-3 retrospective data assimilation system: The 6-hour lag case. *Monthly Weather Review*, 131(9):2129–2150, 2003.
570. X. Zou. Tangent linear and adjoint of "on-off" processes and their feasibility for use in four-dimensional variational data assimilation. *Tellus*, 49A:3–31, 1996.
571. X. Zou, F. Vandenberghe, M. Pondeca, and Y.-H. Kuo. Introduction to adjoint techniques and the MM5 adjoint modeling system. Technical Report NCAR/TN-435-STR, National Center for Atmospheric Research, Boulder, CO, 1997.
572. M. Zupanski, D. Zupanski, D. F. Parrish, E. Rogers, and G. Dimego. Four-dimensional variational data assimilation for the blizzard of 2000. *Monthly Weather Review*, 130(8):1967–1988, 2002.

Index

Editorial Policy

§1. Volumes in the following three categories will be published in LNCSE:

i) Research monographs
ii) Lecture and seminar notes
iii) Conference proceedings

Those considering a book which might be suitable for the series are strongly advised to contact the publisher or the series editors at an early stage.

§2. Categories i) and ii). These categories will be emphasized by Lecture Notes in Computational Science and Engineering. **Submissions by interdisciplinary teams of authors are encouraged.** The goal is to report new developments – quickly, informally, and in a way that will make them accessible to non-specialists. In the evaluation of submissions timeliness of the work is an important criterion. Texts should be well-rounded, well-written and reasonably self-contained. In most cases the work will contain results of others as well as those of the author(s). In each case the author(s) should provide sufficient motivation, examples, and applications. In this respect, Ph.D. theses will usually be deemed unsuitable for the Lecture Notes series. Proposals for volumes in these categories should be submitted either to one of the series editors or to Springer-Verlag, Heidelberg, and will be refereed. A provisional judgment on the acceptability of a project can be based on partial information about the work: a detailed outline describing the contents of each chapter, the estimated length, a bibliography, and one or two sample chapters – or a first draft. A final decision whether to accept will rest on an evaluation of the completed work which should include

– at least 100 pages of text;
– a table of contents;
– an informative introduction perhaps with some historical remarks which should be accessible to readers unfamiliar with the topic treated;
– a subject index.

§3. Category iii). Conference proceedings will be considered for publication provided that they are both of exceptional interest and devoted to a single topic. One (or more) expert participants will act as the scientific editor(s) of the volume. They select the papers which are suitable for inclusion and have them individually refereed as for a journal. Papers not closely related to the central topic are to be excluded. Organizers should contact Lecture Notes in Computational Science and Engineering at the planning stage.

In exceptional cases some other multi-author-volumes may be considered in this category.

§4. Format. Only works in English are considered. They should be submitted in camera-ready form according to Springer-Verlag's specifications.
Electronic material can be included if appropriate. Please contact the publisher.
Technical instructions and/or TEX macros are available via
http://www.springer.com/sgw/cda/frontpage/0,11855,5-40017-2-71391-0,00.html
The macros can also be sent on request.

General Remarks

Lecture Notes are printed by photo-offset from the master-copy delivered in camera-ready form by the authors. For this purpose Springer-Verlag provides technical instructions for the preparation of manuscripts. See also *Editorial Policy*.

Careful preparation of manuscripts will help keep production time short and ensure a satisfactory appearance of the finished book.

The following terms and conditions hold:

Categories i), ii), and iii):
Authors receive 50 free copies of their book. No royalty is paid. Commitment to publish is made by letter of intent rather than by signing a formal contract. Springer-Verlag secures the copyright for each volume.

For conference proceedings, editors receive a total of 50 free copies of their volume for distribution to the contributing authors.

All categories:
Authors are entitled to purchase further copies of their book and other Springer mathematics books for their personal use, at a discount of 33,3 % directly from Springer-Verlag.

Addresses:

Timothy J. Barth
NASA Ames Research Center
NAS Division
Moffett Field, CA 94035, USA
e-mail: barth@nas.nasa.gov

Michael Griebel
Institut für Numerische Simulation
der Universität Bonn
Wegelerstr. 6
53115 Bonn, Germany
e-mail: griebel@ins.uni-bonn.de

David E. Keyes
Department of Applied Physics
and Applied Mathematics
Columbia University
200 S. W. Mudd Building
500 W. 120th Street
New York, NY 10027, USA
e-mail: david.keyes@columbia.edu

Risto M. Nieminen
Laboratory of Physics
Helsinki University of Technology
02150 Espoo, Finland
e-mail: rni@fyslab.hut.fi

Dirk Roose
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven-Heverlee, Belgium
e-mail: dirk.roose@cs.kuleuven.ac.be

Tamar Schlick
Department of Chemistry
Courant Institute of Mathematical
Sciences
New York University
and Howard Hughes Medical Institute
251 Mercer Street
New York, NY 10012, USA
e-mail: schlick@nyu.edu

Mathematics Editor at Springer: Martin Peters
Springer-Verlag, Mathematics Editorial IV
Tiergartenstrasse 17
D-69121 Heidelberg, Germany
Tel.: *49 (6221) 487-8185
Fax: *49 (6221) 487-8355
e-mail: martin.peters@springer.com

Lecture Notes in Computational Science and Engineering

Vol. 1 D. Funaro, *Spectral Elements for Transport-Dominated Equations.* 1997. X, 211 pp. Softcover. ISBN 3-540-62649-2

Vol. 2 H. P. Langtangen, *Computational Partial Differential Equations.* Numerical Methods and Diffpack Programming. 1999. XXIII, 682 pp. Hardcover. ISBN 3-540-65274-4

Vol. 3 W. Hackbusch, G. Wittum (eds.), *Multigrid Methods V.* Proceedings of the Fifth European Multigrid Conference held in Stuttgart, Germany, October 1-4, 1996. 1998. VIII, 334 pp. Softcover. ISBN 3-540-63133-X

Vol. 4 P. Deuflhard, J. Hermans, B. Leimkuhler, A. E. Mark, S. Reich, R. D. Skeel (eds.), *Computational Molecular Dynamics: Challenges, Methods, Ideas.* Proceedings of the 2nd International Symposium on Algorithms for Macromolecular Modelling, Berlin, May 21-24, 1997. 1998. XI, 489 pp. Softcover. ISBN 3-540-63242-5

Vol. 5 D. Kröner, M. Ohlberger, C. Rohde (eds.), *An Introduction to Recent Developments in Theory and Numerics for Conservation Laws.* Proceedings of the International School on Theory and Numerics for Conservation Laws, Freiburg / Littenweiler, October 20-24, 1997. 1998. VII, 285 pp. Softcover. ISBN 3-540-65081-4

Vol. 6 S. Turek, *Efficient Solvers for Incompressible Flow Problems.* An Algorithmic and Computational Approach. 1999. XVII, 352 pp, with CD-ROM. Hardcover. ISBN 3-540-65433-X

Vol. 7 R. von Schwerin, ***M**ulti **B**ody **S**ystem **SIM**ulation.* Numerical Methods, Algorithms, and Software. 1999. XX, 338 pp. Softcover. ISBN 3-540-65662-6

Vol. 8 H.-J. Bungartz, F. Durst, C. Zenger (eds.), *High Performance Scientific and Engineering Computing.* Proceedings of the International FORTWIHR Conference on HPSEC, Munich, March 16-18, 1998. 1999. X, 471 pp. Softcover. 3-540-65730-4

Vol. 9 T. J. Barth, H. Deconinck (eds.), *High-Order Methods for Computational Physics.* 1999. VII, 582 pp. Hardcover. 3-540-65893-9

Vol. 10 H. P. Langtangen, A. M. Bruaset, E. Quak (eds.), *Advances in Software Tools for Scientific Computing.* 2000. X, 357 pp. Softcover. 3-540-66557-9

Vol. 11 B. Cockburn, G. E. Karniadakis, C.-W. Shu (eds.), *Discontinuous Galerkin Methods.* Theory, Computation and Applications. 2000. XI, 470 pp. Hardcover. 3-540-66787-3

Vol. 12 U. van Rienen, *Numerical Methods in Computational Electrodynamics.* Linear Systems in Practical Applications. 2000. XIII, 375 pp. Softcover. 3-540-67629-5

Vol. 13 B. Engquist, L. Johnsson, M. Hammill, F. Short (eds.), *Simulation and Visualization on the Grid.* Parallelldatorcentrum Seventh Annual Conference, Stockholm, December 1999, Proceedings. 2000. XIII, 301 pp. Softcover. 3-540-67264-8

Vol. 14 E. Dick, K. Riemslagh, J. Vierendeels (eds.), *Multigrid Methods VI.* Proceedings of the Sixth European Multigrid Conference Held in Gent, Belgium, September 27-30, 1999. 2000. IX, 293 pp. Softcover. 3-540-67157-9

Vol. 15 A. Frommer, T. Lippert, B. Medeke, K. Schilling (eds.), *Numerical Challenges in Lattice Quantum Chromodynamics.* Joint Interdisciplinary Workshop of John von Neumann Institute for Computing, Jülich and Institute of Applied Computer Science, Wuppertal University, August 1999. 2000. VIII, 184 pp. Softcover. 3-540-67732-1

Vol. 16 J. Lang, *Adaptive Multilevel Solution of Nonlinear Parabolic PDE Systems.* Theory, Algorithm, and Applications. 2001. XII, 157 pp. Softcover. 3-540-67900-6

Vol. 17 B. I. Wohlmuth, *Discretization Methods and Iterative Solvers Based on Domain Decomposition.* 2001. X, 197 pp. Softcover. 3-540-41083-X

Vol. 18 U. van Rienen, M. Günther, D. Hecht (eds.), *Scientific Computing in Electrical Engineering*. Proceedings of the 3rd International Workshop, August 20-23, 2000, Warnemünde, Germany. 2001. XII, 428 pp. Softcover. 3-540-42173-4

Vol. 19 I. Babuška, P. G. Ciarlet, T. Miyoshi (eds.), *Mathematical Modeling and Numerical Simulation in Continuum Mechanics.* Proceedings of the International Symposium on Mathematical Modeling and Numerical Simulation in Continuum Mechanics, September 29 - October 3, 2000, Yamaguchi, Japan. 2002. VIII, 301 pp. Softcover. 3-540-42399-0

Vol. 20 T. J. Barth, T. Chan, R. Haimes (eds.), *Multiscale and Multiresolution Methods.* Theory and Applications. 2002. X, 389 pp. Softcover. 3-540-42420-2

Vol. 21 M. Breuer, F. Durst, C. Zenger (eds.), *High Performance Scientific and Engineering Computing.* Proceedings of the 3rd International FORTWIHR Conference on HPSEC, Erlangen, March 12-14, 2001. 2002. XIII, 408 pp. Softcover. 3-540-42946-8

Vol. 22 K. Urban, *Wavelets in Numerical Simulation.* Problem Adapted Construction and Applications. 2002. XV, 181 pp. Softcover. 3-540-43055-5

Vol. 23 L. F. Pavarino, A. Toselli (eds.), *Recent Developments in Domain Decomposition Methods.* 2002. XII, 243 pp. Softcover. 3-540-43413-5

Vol. 24 T. Schlick, H. H. Gan (eds.), *Computational Methods for Macromolecules: Challenges and Applications.* Proceedings of the 3rd International Workshop on Algorithms for Macromolecular Modeling, New York, October 12-14, 2000. 2002. IX, 504 pp. Softcover. 3-540-43756-8

Vol. 25 T. J. Barth, H. Deconinck (eds.), *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*. 2003. VII, 344 pp. Hardcover. 3-540-43758-4

Vol. 26 M. Griebel, M. A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations*. 2003. IX, 466 pp. Softcover. 3-540-43891-2

Vol. 27 S. Müller, *Adaptive Multiscale Schemes for Conservation Laws*. 2003. XIV, 181 pp. Softcover. 3-540-44325-8

Vol. 28 C. Carstensen, S. Funken, W. Hackbusch, R. H. W. Hoppe, P. Monk (eds.), *Computational Electromagnetics*. Proceedings of the GAMM Workshop on "Computational Electromagnetics", Kiel, Germany, January 26-28, 2001. 2003. X, 209 pp. Softcover. 3-540-44392-4

Vol. 29 M. A. Schweitzer, *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*. 2003. V, 194 pp. Softcover. 3-540-00351-7

Vol. 30 T. Biegler, O. Ghattas, M. Heinkenschloss, B. van Bloemen Waanders (eds.), *Large-Scale PDE-Constrained Optimization*. 2003. VI, 349 pp. Softcover.
3-540-05045-0

Vol. 31 M. Ainsworth, P. Davies, D. Duncan, P. Martin, B. Rynne (eds.), *Topics in Computational Wave Propagation*. Direct and Inverse Problems. 2003. VIII, 399 pp. Softcover. 3-540-00744-X

Vol. 32 H. Emmerich, B. Nestler, M. Schreckenberg (eds.), *Interface and Transport Dynamics.* Computational Modelling. 2003. XV, 432 pp. Hardcover. 3-540-40367-1

Vol. 33 H. P. Langtangen, A. Tveito (eds.), *Advanced Topics in Computational Partial Differential Equations.* Numerical Methods and Diffpack Programming. 2003. XIX, 658 pp. Softcover. 3-540-01438-1

Vol. 34 V. John, *Large Eddy Simulation of Turbulent Incompressible Flows.* Analytical and Numerical Results for a Class of LES Models. 2004. XII, 261 pp. Softcover. 3-540-40643-3

Vol. 35 E. Bänsch (ed.), *Challenges in Scientific Computing - CISC 2002*. Proceedings of the Conference *Challenges in Scientific Computing*, Berlin, October 2-5, 2002. 2003. VIII, 287 pp. Hardcover. 3-540-40887-8

Vol. 36 B. N. Khoromskij, G. Wittum, *Numerical Solution of Elliptic Differential Equations by Reduction to the Interface.* 2004. XI, 293 pp. Softcover. 3-540-20406-7

Vol. 37 A. Iske, *Multiresolution Methods in Scattered Data Modelling.* 2004. XII, 182 pp. Softcover. 3-540-20479-2

Vol. 38 S.-I. Niculescu, K. Gu (eds.), *Advances in Time-Delay Systems.* 2004. XIV, 446 pp. Softcover. 3-540-20890-9

Vol. 39 S. Attinger, P. Koumoutsakos (eds.), *Multiscale Modelling and Simulation.* 2004. VIII, 277 pp. Softcover. 3-540-21180-2

Vol. 40 R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Wildlund, J. Xu (eds.), *Domain Decomposition Methods in Science and Engineering.* 2005. XVIII, 690 pp. Softcover. 3-540-22523-4

Vol. 41 T. Plewa, T. Linde, V.G. Weirs (eds.), *Adaptive Mesh Refinement – Theory and Applications.* 2005. XIV, 552 pp. Softcover. 3-540-21147-0

Vol. 42 A. Schmidt, K.G. Siebert, *Design of Adaptive Finite Element Software.* The Finite Element Toolbox ALBERTA. 2005. XII, 322 pp. Hardcover. 3-540-22842-X

Vol. 43 M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations II.* 2005. XIII, 303 pp. Softcover. 3-540-23026-2

Vol. 44 B. Engquist, P. Lötstedt, O. Runborg (eds.), *Multiscale Methods in Science and Engineering.* 2005. XII, 291 pp. Softcover. 3-540-25335-1

Vol. 45 P. Benner, V. Mehrmann, D.C. Sorensen (eds.), *Dimension Reduction of Large-Scale Systems.* 2005. XII, 402 pp. Softcover. 3-540-24545-6

Vol. 46 D. Kressner (ed.), *Numerical Methods for General and Structured Eigenvalue Problems.* 2005. XIV, 258 pp. Softcover. 3-540-24546-4

Vol. 47 A. Boriçi, A. Frommer, B. Joó, A. Kennedy, B. Pendleton (eds.), *QCD and Numerical Analysis III.* 2005. XIII, 201 pp. Softcover. 3-540-21257-4

Vol. 48 F. Graziani (ed.), *Computational Methods in Transport.* 2006. VIII, 524 pp. Softcover. 3-540-28122-3

Vol. 49 B. Leimkuhler, C. Chipot, R. Elber, A. Laaksonen, A. Mark, T. Schlick, C. Schütte, R. Skeel (eds.), *New Algorithms for Macromolecular Simulation.* 2006. XVI, 376 pp. Softcover. 3-540-25542-7

Vol. 50 M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.), *Automatic Differentiation: Applications, Theory, and Implementations.* 2006. XVIII, 362 pp. Softcover. 3-540-28403-6

For further information on these books please have a look at our mathematics catalogue at the following URL: `www.springer.com/series/3527`

Texts in Computational Science and Engineering

Vol. 1 H. P. Langtangen, *Computational Partial Differential Equations.* Numerical Methods and Diffpack Programming. 2nd Edition 2003. XXVI, 855 pp. Hardcover. ISBN 3-540-43416-X

Vol. 2 A. Quarteroni, F. Saleri, *Scientific Computing with MATLAB.* 2003. IX, 257 pp. Hardcover. ISBN 3-540-44363-0

Vol. 3 H. P. Langtangen, *Python Scripting for Computational Science.* 2nd Edition 2006. XXIV, 736 pp. Hardcover. ISBN 3-540-29415-5

For further information on these books please have a look at our mathematics catalogue at the following URL: `www.springer.com/series/5151`